

Data Structures

Archived study guides

These archived materials are no longer updated. This archive was exported from a saved copy of these guides.

Export date: 2026-10-05T19:26:30.244Z

Contents

- Unit 1 – Data Structures & Algorithms: Introduction - 4 guides
- Unit 2 – Arrays and Linked Lists - 4 guides
- Unit 3 – Stacks and Queues - 3 guides
- Unit 4 – Recursion and its Applications - 4 guides
- Unit 5 – Trees and Binary Trees - 3 guides
- Unit 6 – Binary Search Trees - 3 guides
- Unit 7 – Balanced Trees (AVL and Red-Black Trees) - 3 guides
- Unit 8 – Priority Queues and Heaps - 3 guides
- Unit 9 – Hash Tables and Hash Functions - 3 guides
- Unit 10 – Graphs and their Representations - 3 guides
- Unit 11 – Graph Traversal Algorithms (BFS and DFS) - 3 guides
- Unit 12 – Minimum Spanning Trees & Shortest Paths - 3 guides
- Unit 13 – Sorting Algorithms and their Analysis - 3 guides
- Unit 14 – Searching Algorithms and their Analysis - 3 guides
- Unit 15 – Algorithm Design: Greedy, D&C, Dynamic Prog. - 4 guides

Unit 1 – Data Structures & Algorithms:

Introduction

1.1 Fundamentals of Data Structures and Abstract Data Types

Data structures are essential tools for organizing and managing information efficiently. They come in various forms, each suited for specific tasks and operations. Understanding these structures is crucial for optimizing performance in terms of time and space complexity.

Abstract data types provide a high-level view of data structures, focusing on their behavior rather than implementation. This approach allows for modular, reusable code and helps designers create clear interfaces. Linear and non-linear structures offer different ways to arrange and access data, catering to diverse needs.

Introduction to Data Structures and Abstract Data Types

Role of data structures

- Specialized formats for organizing, processing, retrieving and storing data efficiently
- Manage large amounts of data effectively
- Enable efficient access and modification of data (arrays, linked lists)
- Different data structures suited for various applications and operations
- Choice of data structure depends on data type and desired operations (search, sort, insert, delete)
- Efficient data structures optimize performance in terms of time and space complexity
- Time complexity: time taken by an algorithm to run (measured in Big O notation)
- Space complexity: memory used by an algorithm (measured in Big O notation)

Concept of abstract data types

- Mathematical models of data types that define data and operations performed on data
- Specify what the data type does, not how it does it (implementation-independent)
- Provide high-level description of data type, focusing on essential characteristics and behavior
- Important in designing data structures by providing clear and consistent interface
- Help create modular and reusable code
- Allow changing implementation without affecting users of data type
- Examples:
 - List: collection of elements with defined order
 - Stack: Last-In-First-Out (LIFO) structure
 - Queue: First-In-First-Out (FIFO) structure

Linear vs non-linear structures

- Linear data structures
 - Elements arranged sequentially
 - Each element connected to previous and next element (except first and last)
 - Examples:
 - Arrays: elements stored in contiguous memory locations
 - Linked Lists: elements linked using pointers
 - Stacks: elements added and removed from one end (top)
 - Queues: elements added from one end (rear) and removed from other end (front)
- Non-linear data structures
 - Elements not arranged sequentially
 - Each element can be connected to multiple other elements
 - Examples:
 - Trees: hierarchical structure with root node and child nodes (binary tree, AVL tree)
 - Graphs: collection of vertices (nodes) connected by edges (directed, undirected)
 - Hash Tables: elements stored using hash function for quick access (key-value pairs)

Operations on data structures

- Insertion: adding new element to data structure
- Specific insertion operation depends on data structure (push for stack, enqueue for queue)
- Deletion: removing element from data structure
- Specific deletion operation depends on data structure (pop for stack, dequeue for queue)
- Traversal: accessing each element of data structure in specific order
 - Linear traversal: visiting elements sequentially (array, linked list)
 - Hierarchical traversal: visiting elements hierarchically (tree)
 1. Depth-First Search (DFS): explore as far as possible along each branch before backtracking (preorder, inorder, postorder)
 2. Breadth-First Search (BFS): explore all neighbors at present depth before moving to next depth level (level-order)
- Search: finding specific element in data structure (linear search, binary search)
- Sort: arranging elements in specific order (ascending, descending)
- Various sorting algorithms (bubble sort, insertion sort, merge sort, quicksort)

1.2 Algorithm Analysis and Big O Notation

Algorithm analysis gives you a framework for evaluating how algorithms perform as input sizes grow. Instead of guessing which approach is faster, you can mathematically describe and compare their efficiency. This matters most when you're working with large datasets or time-sensitive applications where a poor algorithm choice can mean the difference between milliseconds and hours.

Algorithm Analysis Fundamentals

Importance of algorithm analysis

Not all algorithms that solve the same problem are equally good. Algorithm analysis lets you measure how an algorithm's running time and resource usage grow relative to input size, rather than just timing it on one machine with one dataset.

- It provides a standardized way to compare algorithms, so you can make informed choices independent of hardware or language
- It helps you identify bottlenecks and predict whether your code will still work well when the input doubles, or grows by a factor of 1000
- For large datasets or real-time systems, picking an algorithm with better scaling behavior can be the difference between a usable program and one that never finishes

Concept of time complexity

Time complexity describes how many operations an algorithm performs as a function of input size n . You're not measuring wall-clock time (that depends on your machine). Instead, you're counting the fundamental steps the algorithm takes.

This count is expressed using asymptotic notation, which captures the growth rate while ignoring constant factors and lower-order terms. The most common time complexities you'll see, ranked from fastest to slowest growth:

- Constant $O(1)$: runtime doesn't change regardless of input size
- Logarithmic $O(\log n)$: runtime grows slowly, even as input gets very large
- Linear $O(n)$: runtime grows directly in proportion to input size
- Quadratic $O(n^2)$: runtime grows with the square of input size, which gets expensive fast

To see why this matters: if you have an $O(n)$ algorithm and an $O(n^2)$ algorithm, they might feel similar at $n = 10$. But at $n = 10,000$, the linear algorithm does roughly 10,000 operations while the quadratic one does 100,000,000.

Big O Notation and Algorithm Comparison

Big O notation fundamentals

Big O notation describes the upper bound of an algorithm's growth rate. It tells you the worst-case scenario for how running time scales with input size.

The key rules for determining Big O:

1. Count the dominant term. If your algorithm does $3n^2 + 5n + 100$ operations, the n^2 term dominates as n grows large.
2. Drop constants. $O(3n^2)$ simplifies to $O(n^2)$, because constants don't affect the growth rate.
3. Drop lower-order terms. $O(n^2 + n)$ simplifies to $O(n^2)$, since n becomes negligible compared to n^2 for large n .

The common Big O classes, from most to least efficient:

- $O(1)$: Constant (e.g., accessing an array element by index)
- $O(\log n)$: Logarithmic (e.g., binary search)
- $O(n)$: Linear (e.g., scanning through a list)
- $O(n \log n)$: Linearithmic (e.g., merge sort)
- $O(n^2)$: Quadratic (e.g., nested loops over the same data)
- $O(2^n)$: Exponential (e.g., naive recursive Fibonacci)

Lower Big O classes scale better. An $O(n \log n)$ sorting algorithm will consistently outperform an $O(n^2)$ one on large inputs, even if the quadratic one is faster for very small arrays.

Time complexity comparisons

Here's how Big O plays out across common algorithm categories:

1. Searching algorithms: - Linear search $O(n)$: checks each element one by one. Works on any list, but slow for large inputs. - Binary search $O(\log n)$: repeatedly divides a sorted list in half. At each step, half the remaining elements are eliminated. For a list of 1,000,000 elements, binary search needs at most about 20 comparisons.
2. Sorting algorithms: - Bubble sort, selection sort, insertion sort $O(n^2)$: these use nested loops to compare and rearrange elements. Simple to implement, but impractical for large datasets. - Merge sort, quicksort $O(n \log n)$: these use a divide-and-conquer strategy, splitting the list into smaller pieces, sorting them, and combining the results. Note that quicksort's worst case is actually $O(n^2)$, but its average case is $O(n \log n)$.
3. Graph traversal algorithms: - Depth-first search (DFS) and breadth-first search (BFS) $O(V + E)$: both visit every vertex V and traverse every edge E in the graph. The complexity depends on the graph's structure, not just the number of nodes.
4. Dynamic programming (optimization example): - Fibonacci (naive recursive) $O(2^n)$: each call branches into two more calls, recalculating the same values over and over. At $n = 40$, this already takes over a billion operations. - Fibonacci (memoized or bottom-up) $O(n)$: by storing previously computed results, you eliminate all the redundant work. This is a classic example of how algorithmic thinking can turn an unusable solution into an efficient one.

1.3 Time and Space Complexity

Time and space complexity are crucial concepts in algorithm analysis. They measure how an algorithm's performance scales with input size, helping developers choose efficient solutions for different scenarios.

Balancing time and space complexity often involves trade-offs. Some algorithms excel in speed but use more memory, while others conserve memory at the cost of longer execution times. Understanding these trade-offs is key to optimizing algorithmic performance.

Understanding Time and Space Complexity

Time vs space complexity

- Time complexity measures the amount of time an algorithm takes to run as the input size grows (expressed using Big O notation such as $O(1)$, $O(n)$, $O(n^2)$)
- Space complexity measures the amount of memory an algorithm requires as the input size grows (also expressed using Big O notation)
- Time complexity focuses on the number of operations or steps required to solve a problem
- Space complexity considers the memory needed for input data, output data, and any additional memory used during the algorithm's execution

Complexity trade-offs in algorithms

- Balancing time and space complexity involves considering algorithms that may have excellent time complexity but require more memory (hash tables) or algorithms with lower memory requirements but longer execution times
- Trade-offs depend on specific requirements and constraints of the problem such as available memory, required response time, and input data size
- Recursion can simplify code and reduce time complexity but may require more memory due to function call overhead and stack space

Space complexity analysis

- Identifying memory usage involves considering the memory needed for input data, output data, and any additional data structures used
- Analyze the growth of memory usage as the input size increases
- Common space complexities include $O(1)$ constant space complexity where memory usage remains constant regardless of input size, $O(n)$ linear space complexity where memory usage grows linearly with input size, and $O(n^2)$ quadratic space complexity where memory usage grows quadratically with input size
- Techniques for optimizing space complexity include in-place algorithms that modify input data directly without requiring additional memory and streaming algorithms that process input data in a single pass to reduce memory requirements

Examples of algorithmic complexities

- Constant time and space complexity $O(1)$ examples: 1. Accessing an array element by index 2. Basic arithmetic operations
- Linear time complexity $O(n)$ examples: 1. Searching for an element in an unsorted array (linear search) 2. Traversing a linked list
- Quadratic time complexity $O(n^2)$ examples: 1. Nested loops, such as in a brute-force approach to compare all pairs of elements 2. Bubble sort, selection sort, and insertion sort algorithms
- Logarithmic time complexity $O(\log n)$ examples: 1. Binary search on a sorted array 2. Balanced binary search trees (AVL trees, Red-Black trees)

- Linearithmic time complexity $O(n \log n)$ examples: 1. Efficient sorting algorithms (Merge sort, Quick sort, Heap sort) 2. Divide-and-conquer algorithms (Karatsuba multiplication for large numbers)

1.4 Data Structure Selection and Trade-offs

Data Structure Selection Considerations

Choosing the right data structure is one of the most practical skills you'll develop in this course. Every data structure has strengths and weaknesses, and the goal is to match those traits to what your problem actually demands. A poor choice can turn an efficient algorithm into a slow one, even if the logic is correct.

This section covers the factors that guide your selection, the performance characteristics of common structures, and the trade-offs you'll need to weigh.

Factors in Data Structure Selection

Before picking a data structure, you need to understand your problem along several dimensions:

Problem requirements come first. What operations does your program need to perform? If you're mostly searching, that points you toward different structures than if you're mostly inserting and deleting. Think about how often each operation runs, too. An operation that happens once at startup matters less than one that runs inside a loop millions of times. Also consider hard constraints like memory limits or response-time requirements.

Data characteristics matter just as much. Ask yourself:

- How large is the dataset? Ten items? Ten million?
- Is the data static (loaded once, then only read) or dynamic (constantly changing)?
- What type of data are you storing? Are there relationships between elements, like parent-child hierarchies?

Efficiency is where you get quantitative. Evaluate the time complexity of the operations you need most, and check the space complexity to make sure your structure fits in available memory.

Ease of implementation shouldn't be ignored. A theoretically optimal structure that's error-prone to implement can cost you more time debugging than a simpler structure that's "good enough." Readable, maintainable code has real value.

Performance of Data Structures

Here's how the most common structures perform and where they shine:

Arrays - $O(1)$ access by index, which makes them ideal when you know where an element is - Great for sequential access and fixed-size collections - Insertion or deletion in the middle requires shifting elements, costing $O(n)$

Linked Lists - $O(1)$ insertion and deletion if you already have a reference to the node (finding that node first costs $O(n)$) - Well-suited for dynamic data that changes size frequently - Each node stores an extra pointer, so they use more memory per element than arrays - Random access is $O(n)$ because you must traverse from the head

Stacks and Queues - Both provide $O(1)$ insertion and removal, but only at specific ends - A stack follows LIFO (Last-In-First-Out) order. Think function call management and undo operations. - A queue follows FIFO (First-In-First-Out) order. Think task scheduling and breadth-first search.

Trees - Balanced trees (like AVL trees and B-trees) give $O(\log n)$ search, insertion, and deletion - Naturally represent hierarchical data (file systems, organizational charts) - Require extra memory for child pointers - An unbalanced binary search tree can degrade to $O(n)$ in the worst case, which is why balanced variants exist

Hash Tables - $O(1)$ average-case access, insertion, and deletion - Excellent for fast lookups and key-value storage - Worst case can degrade to $O(n)$ if many keys collide into the same bucket - Collisions are handled through techniques like chaining (storing a list at each bucket) or open addressing (probing for the next empty slot)

Trade-offs and Guidelines

Time vs. Space Complexity Trade-offs

There's almost always a tension between time and space. Faster access often costs more memory, and saving memory often means slower operations.

Concrete examples make this clearer:

- Hash tables achieve $O(1)$ average access by allocating extra space for buckets and handling collisions. You're trading memory for speed.
- Linked lists use only as much memory as they need (no wasted slots), but you pay for that with $O(n)$ random access since there's no index.
- Arrays give you $O(1)$ index-based access, but if your data is sparse (say, only 100 values spread across indices 0 to 1,000,000), most of that allocated space is wasted.

When making a selection, ask: Is time or space the bigger constraint for this problem? The answer steers you toward the right side of the trade-off.

Guidelines for Structure Suitability

Use this as a quick-reference for matching your problem's needs to a structure:

Frequent access by index or key 1. Arrays: $O(1)$ access by index 2. Hash tables: $O(1)$ average-case access by key

Frequent insertion and deletion 1. Linked lists: $O(1)$ insertion/deletion when you have a reference to the position 2. Balanced trees: $O(\log n)$ insertion and deletion while maintaining order

Hierarchical or recursive data - Trees naturally represent parent-child relationships (file directories, parse trees) - Recursive data structures fit problems that break into smaller subproblems of the same shape

Priority-based access - Heaps provide $O(1)$ access to the min or max element, with $O(\log n)$ insertion - Priority queues (often implemented with heaps) maintain elements ordered by priority

Range queries or ordered data - Balanced search trees (BSTs, AVL trees, B-trees) support efficient range queries and keep elements sorted

Unique elements and fast membership testing - Sets store unique elements and provide fast membership checks - Hash tables offer $O(1)$ average-case membership testing

Key-value pair storage - Hash tables (and the dictionaries/maps built on them) give efficient storage and retrieval by key

Stack or queue behavior - Stacks (LIFO): function call management, expression evaluation, backtracking algorithms - Queues (FIFO): task scheduling, breadth-first search, buffering

The pattern to notice: there's no single "best" data structure. The best choice depends entirely on which operations dominate your problem. Get clear on your requirements first, then let the performance characteristics guide your decision.

Unit 2 – Arrays and Linked Lists

2.1 Array Operations and Implementation

Array Fundamentals

Arrays store elements in contiguous memory locations, meaning each element sits right next to the previous one in memory. This layout is what gives arrays their biggest advantage: you can jump directly to any element using its index, without scanning through everything before it.

Static arrays have fixed sizes set at compile time, while dynamic arrays can grow or shrink at runtime. One-dimensional arrays store simple lists, and multi-dimensional arrays represent grids, matrices, or tables.

Basic Array Operations

Access is the simplest operation. Because elements are stored contiguously, the computer can calculate exactly where any element lives using the formula:

$$\text{address} = \text{base_address} + \text{index} \times \text{element_size}$$

This makes access $O(1)$, regardless of array size.

Insertion adds an element at a specific index. To make room, every element from that index onward must shift one position to the right.

For example, inserting 3 at index 2 in [1, 2, 4, 5]: 1. Shift 5 from index 3 to index 4 2. Shift 4 from index 2 to index 3 3. Place 3 at index 2 4. Result: [1, 2, 3, 4, 5]

Worst-case time complexity: $O(n)$, because inserting at index 0 requires shifting all n elements. Inserting at the end, though, is $O(1)$ since no shifting is needed.

Deletion removes an element at a specific index. Elements after the deleted one shift left to fill the gap.

For example, deleting index 2 in [1, 2, 3, 4, 5]: 1. Shift 4 from index 3 to index 2 2. Shift 5 from index 4 to index 3 3. Result: [1, 2, 4, 5]

Worst-case time complexity: $O(n)$, for the same shifting reason. Deletion at the end is $O(1)$.

Traversal visits each element sequentially, typically with a loop like `for (int i = 0; i < n; i++)`. Since every element is visited once, this is always $O(n)$.

Time Complexity Summary

Operation	Best Case	Worst Case	Notes
Access by index	$O(1)$	$O(1)$	Direct calculation from base address
Search (unsorted)	$O(1)$	$O(n)$	Linear search; element might be first or last
Search (sorted)	$O(1)$	$O(\log n)$	Binary search halves the search space each step
Insert/Delete (end)	$O(1)$	$O(1)$	No shifting required
Insert/Delete (arbitrary)	$O(1)$	$O(n)$	Shifting elements dominates the cost
Traversal	$O(n)$	$O(n)$	Must visit every element

The key tradeoff with arrays: you get fast random access but pay for it with slow insertions and deletions in the middle.

Array Types and Applications

Static vs. Dynamic Arrays

Static arrays have a fixed size determined at compile time (e.g., `int arr[10]` in C++). Memory is allocated on the stack, which is fast but inflexible. If you need more space than you declared, you're out of luck.

Dynamic arrays can resize at runtime. Languages provide built-in implementations like `vector` in C++ and `ArrayList` in Java. Memory is allocated on the heap, which allows resizing but adds some overhead.

When a dynamic array runs out of capacity, it typically: 1. Allocates a new, larger block of memory (often double the current size) 2. Copies all existing elements to the new block 3. Frees the old block

That copy step is $O(n)$, but it happens infrequently enough that appending to a dynamic array averages out to $O(1)$ per insertion. This is called amortized $O(1)$.

One-Dimensional and Multi-Dimensional Arrays

One-dimensional arrays store a flat list of elements of the same type, accessed with a single index. Example: storing student grades as `[85, 92, 78, 90]`, where `arr[2]` gives you 78.

Multi-dimensional arrays add extra dimensions. A 2D array is the most common, representing grids or matrices like a chessboard. For a 2D array `[[1, 2], [3, 4]]`, you access elements with two indices: `arr[1][0]` returns 3 (second row, first column). Higher dimensions (3D, 4D) exist but are less common.

Common Array Applications

- Searching: linear search on unsorted data, binary search on sorted data
- Finding min/max: single traversal through the array
- Sorting: algorithms like bubble sort, quicksort, and mergesort all operate on arrays
- Building other data structures: stacks, queues, and hash tables are frequently implemented with arrays as the underlying storage

2.2 Singly Linked Lists

Singly linked lists are one of the first dynamic data structures you'll encounter. Unlike arrays, which store elements in contiguous memory, a singly linked list chains together individual nodes that can live anywhere in memory. This makes insertion and deletion at the front extremely fast, but it means you lose the ability to jump directly to any element by index.

Singly Linked List Fundamentals

Structure of singly linked lists

A singly linked list is built from nodes, where each node holds two things: some data and a pointer (or reference) to the next node in the sequence. You can think of it like a scavenger hunt where each clue tells you where to find the next one.

- Each node contains a data field and a next pointer (e.g., `Node { int data; Node* next; }`)
- The last node's next pointer is set to null, which signals the end of the list
- A head pointer points to the first node and serves as your entry point to the entire list
- A tail pointer (optional) points to the last node, which lets you skip the traversal when inserting at the end

Without a tail pointer, you'd have to walk through every node just to reach the end. With one, appending becomes much cheaper.

Operations for singly linked lists

Insertion

1.

At the beginning: - Create a new node and set its next pointer to the current head - Update the head pointer to the new node - Time complexity: $O(1)$ (no traversal needed)

2.

At the end (without a tail pointer): - Traverse the list to find the last node - Create a new node and set the last node's next pointer to it - Time complexity: $O(n)$ - With a tail pointer, this drops to $O(1)$ since you can go straight to the last node

3.

At a specific position: - Traverse to the node before the desired position - Create a new node, point it to the current node's next - Update the current node's next pointer to the new node - Time complexity: $O(n)$

Deletion

1.

From the beginning: - Save a reference to the current head - Update the head pointer to `head.next` - Deallocate the old head node - Time complexity: $O(1)$

2.

From the end: - Traverse to the second-to-last node - Set its next pointer to null - Update the tail pointer (if used) to this node - Deallocate the old last node - Time complexity: $O(n)$ (even with a tail pointer, because

you need the second-to-last node)

3.

From a specific position: - Traverse to the node before the target - Update its next pointer to skip over the target node - Deallocate the deleted node - Time complexity: $O(n)$

A common mistake: forgetting to update the head or tail pointer after insertion or deletion. If you delete the only node in the list, both head and tail should become null.

Traversal starts at the head and follows next pointers until you hit null. Time complexity: $O(n)$. There's no way to go backwards in a singly linked list, so if you pass the node you need, you have to start over from the head.

Singly Linked List Analysis and Applications

Complexity of linked list operations

Operation	Time Complexity
Insert at beginning	$O(1)$
Insert at end (no tail pointer)	$O(n)$
Insert at end (with tail pointer)	$O(1)$
Insert at position	$O(n)$
Delete from beginning	$O(1)$
Delete from end	$O(n)$
Delete from position	$O(n)$
Search / Traversal	$O(n)$

Space complexity is $O(n)$ for n elements, but each node carries overhead for storing its next pointer. For small data types like integers, this overhead can be significant compared to a plain array.

Applications of singly linked lists

-

Hash table chaining: Each bucket in a hash table can be a linked list. When two keys hash to the same index (a collision), the new element is simply appended to that bucket's list. This is called separate chaining, and it's one of the most common collision resolution strategies.

-

Polynomial representation: Each term of a polynomial is stored as a node containing a coefficient and an exponent. For example, $3x^2 + 2x + 1$ would be three nodes. The linked list structure makes it straightforward to add or remove terms without shifting elements around.

-

Graph adjacency lists: Each vertex in a graph has a linked list of its neighbors. This is memory-efficient for sparse graphs (graphs with relatively few edges) compared to storing a full adjacency matrix.

-

Undo functionality: Some implementations store each user action as a node. To undo, you walk back through the list and reverse the most recent action. (In practice, a stack is more common for undo, but a linked list can serve as the underlying structure for that stack.)

2.3 Doubly Linked Lists and Circular Linked Lists

Doubly Linked Lists

Structure of doubly linked lists

A singly linked list only lets you move forward through the nodes. A doubly linked list fixes that limitation by giving each node two pointers instead of one: one pointing to the next node and one pointing to the previous node. This bidirectional linking is what makes the structure so flexible.

Each node in a doubly linked list contains three fields:

- Data storing the node's value
- Next pointer pointing to the following node
- Previous pointer pointing to the preceding node

The list itself maintains a head pointer (pointing to the first node) and a tail pointer (pointing to the last node). The head's previous pointer is null, and the tail's next pointer is null.

Because you can traverse in both directions, doubly linked lists simplify operations that are awkward in singly linked lists. For example, if you're given only a pointer to a node you want to delete, you can reach both its neighbors directly without traversing from the head. Common real-world uses include undo/redo functionality in text editors and forward/back navigation in web browsers.

Operations in doubly linked lists

Every insertion or deletion in a doubly linked list requires updating two pointers per affected node (next and previous), compared to just one in a singly linked list. Here's how each operation works:

Insertion at the beginning:

1. Create a new node
2. Set the new node's next to the current head
3. Set the new node's prev to null
4. Set the old head's prev to the new node
5. Update head to point to the new node
6. If the list was empty, also set tail to the new node

Insertion at the end:

1. Create a new node
2. Set the new node's prev to the current tail
3. Set the new node's next to null
4. Set the old tail's next to the new node

5. Update tail to point to the new node
6. If the list was empty, also set head to the new node

Insertion in the middle (after a given node A):

1. Create a new node N
2. Set N 's next to A 's next
3. Set N 's prev to A
4. Set A 's next's prev to N
5. Set A 's next to N

Deletion at the beginning:

1. Set head to the current head's next
2. Set the new head's prev to null
3. If the list is now empty, set tail to null

Deletion at the end:

1. Set tail to the current tail's prev
2. Set the new tail's next to null
3. If the list is now empty, set head to null

Deletion in the middle (removing node N):

1. Set N 's previous node's next to N 's next
2. Set N 's next node's prev to N 's prev

Traversal is straightforward: forward traversal starts at head and follows next pointers until null; backward traversal starts at tail and follows prev pointers until null. Both are $O(n)$.

Circular Linked Lists

Concept of circular linked lists

In a standard linked list, the last node's next pointer is null, marking the end. A circular linked list changes this: the last node's next pointer loops back to the first node, forming a continuous cycle with no null terminator.

There are two variants:

- Circular singly linked list: The last node's next points to the head. You can only traverse forward, but you can reach any node from any starting point by continuing around the loop.
- Circular doubly linked list: The last node's next points to the head, and the head's prev points to the last node. This gives you full bidirectional traversal with no dead ends.

Circular lists are a natural fit for problems that involve cycling through elements repeatedly. Round-robin CPU scheduling rotates through processes in a loop. Multiplayer game turns cycle through players continuously. Circular buffers use the same idea to manage fixed-size queues where old data gets

overwritten.

One thing to watch for: since there's no null to signal the end, traversal logic needs a different stopping condition. You typically save a reference to your starting node and stop when you reach it again.

Doubly vs singly linked lists

Space complexity:

- Singly linked lists use one pointer per node. Doubly linked lists use two, so they consume more memory per node.
- Circular variants don't add extra memory beyond their non-circular counterparts. The last node simply reuses its existing pointer to point to the head instead of null.

Time complexity:

Operation	Singly Linked	Doubly Linked	Circular (Singly)	Circular (Doubly)
Insert at beginning	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Insert at end	$O(n)^*$	$O(1)$	$O(1)^{**}$	$O(1)$
Delete at beginning	$O(1)$	$O(1)$	$O(1)$	$O(1)$
Delete at end	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Delete given node	$O(n)$	$O(1)$	$O(n)$	$O(1)$
Traversal	Forward only, $O(n)$	Both directions, $O(n)$	Forward only, $O(n)$	Both directions, $O(n)$

* $O(1)$ if you maintain a tail pointer, which singly linked lists sometimes do.

** Assumes you maintain a pointer to the last node, which is standard practice for circular singly linked lists.

Ease of implementation:

- Singly linked lists are the simplest since each operation only updates one pointer per node.
- Doubly linked lists require more careful pointer management (two pointers per update), but the payoff is $O(1)$ deletion of any node you already have a reference to.
- Circular linked lists add complexity around maintaining the loop. You need to handle the circular connection whenever you insert or delete, and your traversal logic must detect when you've completed a full cycle to avoid infinite loops.

2.4 Comparison of Arrays and Linked Lists

Array and Linked List Comparison

Arrays and linked lists are the two foundational data structures you'll compare everything else against. Arrays give you fast random access but come with rigid sizing and costly insertions. Linked lists offer flexible sizing and cheap insertions but force you to traverse sequentially. Understanding the trade-offs between them is the key to picking the right structure for a given problem.

Arrays vs Linked Lists

Arrays store elements in contiguous memory, which means every element sits right next to the previous one. This layout is what makes index-based access so fast.

- Access any element directly by its index in $O(1)$ time
- Size is fixed at creation. You can't grow or shrink an array without allocating a new one and copying elements over.
- Inserting or deleting in the middle is expensive. Removing the 5th element from a 100-element array means shifting the remaining 95 elements down one position, costing $O(n)$ time.

Linked lists store elements as individual nodes scattered across memory. Each node holds its data plus a pointer to the next node (and in doubly-linked lists, a pointer to the previous node too).

- Accessing the k th element requires traversing from the head, one node at a time: $O(n)$
- Size grows and shrinks dynamically. You allocate a new node only when you need one.
- Inserting or deleting a node is $O(1)$ once you already have a reference to the correct position, since you're just updating pointers. Finding that position still costs $O(n)$ if you have to search for it.
- Each node carries extra memory overhead for its pointer(s).

Time and Space Complexity Comparison

Operation	Array	Linked List
Access by index	$O(1)$	$O(n)$
Search (unsorted)	$O(n)$	$O(n)$
Search (sorted)	$O(\log n)$ (binary search)	$O(n)$ (binary search not possible)
Insert/Delete at known position	$O(n)$ (shifting required)	$O(1)$ (pointer update only)
Insert/Delete at end	$O(1)$ (if space available)	$O(1)$ (with tail pointer)
Memory layout	Contiguous, fixed allocation	Non-contiguous, per-node allocation

Space trade-offs: An array of size 100 that only holds 50 elements wastes half its allocated memory. A linked list with 50 nodes uses exactly 50 nodes' worth of memory, but each node is larger than a plain array element because it also stores pointer(s). For a singly-linked list, that's one extra pointer per node; for a doubly-linked list, it's two.

Data Structure Selection

Choose an array when:

- The data size is known in advance and doesn't change much
- You need random access (e.g., jumping directly to the 10th element)
- The data is dense and insertions/deletions in the middle are rare

Choose a linked list when:

- The data size is unknown or changes frequently
- Sequential access is sufficient (e.g., processing every element from start to finish)
- You need frequent insertions or deletions at arbitrary positions, such as maintaining a sorted list with constant additions and removals

Adapting Algorithms Between Data Structures

Sometimes you'll need to swap one structure for the other. Here's a systematic approach:

1. Identify the problem's requirements. What are the access patterns (random vs. sequential)? How often do insertions and deletions happen? What are the memory constraints?
2. Analyze the current algorithm. Which operations dominate the runtime? Where are the bottlenecks?
3. Modify the algorithm for the new structure: - Array $\rightarrow$ Linked list: Replace index-based access with pointer traversal. Handle dynamic node allocation. Swap element-shifting logic for pointer updates. - Linked list $\rightarrow$ Array: Replace traversal with direct indexing. Determine an appropriate array size upfront. Add shifting logic for insertions and deletions.
4. Evaluate the complexity trade-offs. Switching from an array to a linked list might speed up insertions from $O(n)$ to $O(1)$ but slow down access from $O(1)$ to $O(n)$. Make sure the trade-off actually helps your use case.
5. Test for correctness and benchmark performance to confirm the change was worthwhile.

Algorithmic Efficiency

How Data Structure Choice Affects Efficiency

The data structure you pick shapes the performance of every algorithm that uses it. Two things to keep in mind:

Time-space trade-off. Arrays generally give faster access but can waste memory with fixed-size allocation. Linked lists use memory more precisely but pay for it with slower traversal and extra pointer storage. The right choice depends on which resource matters more for your problem.

Asymptotic analysis with Big O notation is how you compare these trade-offs formally. Focus on the dominant term and the worst-case scenario. An algorithm running in $O(n)$ on an array will outperform one running in $O(n^2)$ on a linked list for large inputs, even if the linked list version is simpler to implement. Always consider how performance scales as input size grows.

A common mistake: assuming linked list insertion is always $O(1)$. It's $O(1)$ only if you already have a pointer to the insertion point. If you need to find that point first, the search adds $O(n)$, making the total operation $O(n)$.

Unit 3 – Stacks and Queues

3.1 Stack ADT and Applications

Stacks are one of the most fundamental data structures you'll encounter. A stack enforces a simple rule: the last element you add is the first one you remove. This Last-In-First-Out (LIFO) behavior shows up everywhere, from how your computer manages function calls to how the undo button works in a text editor.

Every primary stack operation runs in $O(1)$ time, which makes stacks extremely efficient when you only need access to the most recently added element.

Stack ADT Fundamentals

Stack Abstract Data Type

An abstract data type (ADT) defines what operations a data structure supports without specifying how they're implemented. The Stack ADT organizes elements in LIFO order, where all insertions and deletions happen at the same end, called the top of the stack.

Think of a stack of plates: you always add a new plate on top, and you always grab the top plate first. You never pull from the middle or bottom.

The five primary operations are:

1. `push(element)` — Insert an element onto the top of the stack
2. `pop()` — Remove and return the top element
3. `peek()` (also called `top()`) — View the top element without removing it
4. `isEmpty()` — Return true if the stack contains no elements
5. `size()` — Return the number of elements currently in the stack

All five operations run in $O(1)$ time because they only interact with the top of the stack. There's no traversal involved.

LIFO Principle

The LIFO principle means the last element pushed onto the stack is the first one popped off. The element sitting at the bottom was the first one inserted and will be the last one removed.

This means a stack accesses elements in the reverse order of their insertion. That reversal property is exactly what makes stacks useful for things like:

- Pressing the back button in a web browser (you return to the most recently visited page first)
- Undo operations (the most recent action gets undone first)
- Backtracking (you retrace your most recent steps before earlier ones)

Applications of Stacks

Function call management. When a function is called, the system pushes a stack frame onto the call stack. That frame holds local variables, parameters, and the return address. When the function finishes, its frame is popped off, and execution returns to the caller. This is how recursive functions unwind: each recursive call stacks a new frame, and they resolve in reverse order.

Expression evaluation. Stacks are used to evaluate arithmetic expressions in postfix (reverse Polish) notation. Operands get pushed onto the stack; when an operator is encountered, the required operands are popped, the operation is performed, and the result is pushed back. Infix and prefix evaluation also rely on stacks to handle operator precedence. Calculators use this approach internally.

Undo/redo functionality. Text editors and design tools maintain a stack of previous actions. Each undo pops the most recent action off the stack. A separate redo stack can hold undone actions so they can be reapplied.

Backtracking algorithms. Depth-first search (DFS) uses a stack to track which nodes have been visited and which paths remain to explore. When a dead end is reached, the algorithm pops back to the most recent node with unexplored neighbors.

Syntax parsing and validation. Compilers check for balanced parentheses, brackets, and braces by pushing each opening symbol onto a stack. When a closing symbol appears, the top of the stack is popped and checked for a match. If the stack is empty at the end and every symbol matched, the expression is balanced.

Time Complexity of Stack Operations

All basic stack operations run in constant time because they only access or modify the top element, with no need to traverse the rest of the structure:

Operation	Time Complexity
push(element)	$O(1)$
pop()	$O(1)$
peek() / top()	$O(1)$
isEmpty()	$O(1)$
size()	$O(1)$

The space complexity of a stack is $O(n)$, where n is the number of elements stored. The stack grows linearly as you push more elements onto it. In practice, this matters for things like deep recursion: if a recursive function calls itself thousands of times, the call stack consumes $O(n)$ memory and can overflow if n is large enough.

3.2 Queue ADT and Applications

Queue ADT and Its Applications

Queue Abstract Data Type

A queue is a collection where items are inserted at one end (the rear) and removed from the other end (the front). This makes it a First-In-First-Out (FIFO) structure: whatever goes in first comes out first, like a line at a grocery store.

The core operations of the Queue ADT are:

- `enqueue(element)` — adds an element to the rear of the queue
- `dequeue()` — removes and returns the element at the front of the queue
- `front()` (sometimes called `peek`) — returns the front element without removing it
- `isEmpty()` — returns true if the queue has no elements
- `size()` — returns the number of elements currently in the queue

All five of these operations run in $O(1)$ time, which is what makes queues so practical for high-throughput processing.

Under the hood, queues are typically implemented with either a linked list or a circular array. A standard (non-circular) array implementation can cause problems: every dequeue shifts elements or wastes space at the front. A circular array solves this by wrapping the front and rear indices around using modular arithmetic.

FIFO Principle

The FIFO principle is what distinguishes a queue from a stack. In a queue, the element that has been waiting the longest is always the next one served. In a stack, the most recently added element comes out first (LIFO).

Queue (FIFO): First element in is the first element out — like a line of people.

Stack (LIFO): Last element in is the first element out — like a stack of plates.

FIFO is the right model whenever fairness or ordering matters. Think of a ticket counter: the person who arrived first should be helped first. Any system that needs to preserve arrival order will use a queue.

Applications of Queues

Queues show up across many areas of computing. Here are the most important ones to know:

- **Task scheduling** — Operating systems use queues to manage processes waiting for CPU time. A print spooler, for example, prints documents in the order they were submitted.
- **Breadth-First Search (BFS)** — BFS explores a graph level by level. It uses a queue to track which nodes to visit next: enqueue all neighbors of the current node, then dequeue the next node to process. This guarantees you visit nodes in order of their distance from the start.
- **Message passing** — In distributed systems, message queues (like RabbitMQ or Kafka) let components communicate asynchronously. The sender enqueues messages, and the receiver dequeues them in order.
- **Event handling** — GUI frameworks maintain an event queue for user actions (key presses, mouse clicks). Events are processed in the order they occur, so the interface responds predictably.
- **Buffering** — Queues act as buffers when a producer generates data faster than a consumer can process it. Streaming video, for instance, buffers frames in a queue so playback stays smooth.

Note on cache management: some guides list cache eviction as a queue application. While a simple FIFO cache does use a queue, most real caches (like LRU caches) use more complex structures. For this course, focus on the other applications above.

Time Complexity of Queue Operations

Every standard queue operation runs in $O(1)$ time, regardless of how many elements are in the queue. Here's why each one is constant time:

Operation	Time	How It Works
enqueue	$O(1)$	Appends to the rear. In a linked list, update the tail pointer. In a circular array, place at the rear index and increment.
dequeue	$O(1)$	Removes from the front. In a linked list, update the head pointer. In a circular array, increment the front index.
front	$O(1)$	Access the element at the front pointer/index without modifying anything.
isEmpty	$O(1)$	Check whether the size counter equals zero (or whether front and rear indices indicate an empty state).
size	$O(1)$	Return a maintained counter that gets incremented on enqueue and decremented on dequeue.

The key detail: these $O(1)$ guarantees depend on the implementation. A linked list gives $O(1)$ naturally. A circular array gives $O(1)$ amortized (with occasional resizing if the array fills up). A naive array where you shift elements on every dequeue would be $O(n)$, which is why circular arrays are preferred.

3.3 Implementation of Stacks and Queues using Arrays and Linked Lists

Array-based and Linked List-based Implementations of Stacks and Queues

Stacks with arrays vs linked lists

Both array-based and linked list-based stacks support the same operations (push, pop, peek), but they differ in how they manage memory under the hood. Your choice depends on whether you know the maximum size ahead of time and how much you care about memory overhead.

Array-based stack uses a fixed-size array and a top variable that tracks the index of the topmost element. Pushing increments top and places the element; popping reads the element at top and decrements it.

- Advantages
- Straightforward implementation
- Constant time ($O(1)$) access to elements by index
- Memory-efficient since no extra space is needed for pointers
- Better cache locality because elements sit in contiguous memory
- Disadvantages
- Fixed size can cause stack overflow if the array fills up

- Wastes memory if you allocate a large array but only use a fraction of it
- Resizing (if supported) requires copying all elements to a new array, which costs $O(n)$

Linked list-based stack uses a singly linked list where the head of the list acts as the top of the stack. Pushing inserts a new node at the head; popping removes the head node.

- Advantages
- Dynamic size, so the stack grows and shrinks as needed
- No overflow (as long as the system has free memory)
- No wasted capacity since each node is allocated only when needed
- Disadvantages
- Each node carries extra memory overhead for its pointer (typically 4 or 8 bytes per node)
- Nodes are scattered in memory, so cache performance is worse than arrays
- Slightly more complex to implement due to pointer management

Queues with arrays vs linked lists

Queues add at the rear and remove from the front (FIFO). The tricky part with an array-based queue is that naive implementations waste space as the front index advances. That's why array-based queues almost always use a circular array design.

Array-based queue (circular) uses a fixed-size array with two indices, front and rear, that wrap around using modular arithmetic. For example, the next position after index i is $(i + 1) \bmod n$, where n is the array capacity. This prevents the "used-up front slots" problem.

- Advantages
- Straightforward once you understand the wrap-around logic
- Constant time ($O(1)$) enqueue and dequeue
- Memory-efficient with no pointer overhead
- Disadvantages
- Fixed size can cause queue overflow if the array fills up
- You need to decide the capacity upfront (or implement costly resizing)
- Distinguishing between "full" and "empty" requires care (common approach: track a size counter, or keep one slot unused)

Linked list-based queue uses a singly linked list with a front pointer (for dequeue) and a rear pointer (for enqueue). Enqueue appends a new node at rear; dequeue removes the node at front.

- Advantages
- Dynamic size with no overflow concern
- Memory is allocated per element, so nothing is wasted
- No wrap-around logic needed

- Disadvantages
- Extra memory for pointers in every node
- Worse cache locality than arrays
- Must handle the edge case where front and rear both point to the same node (single-element queue)

Performance of stack and queue implementations

Both implementations achieve the same time complexity for core operations, so the real differences come down to memory behavior and constant factors.

Operation	Array-based	Linked list-based
Push / Enqueue	$O(1)$	$O(1)$
Pop / Dequeue	$O(1)$	$O(1)$
Peek / Front	$O(1)$	$O(1)$
Space	$O(n)$ fixed (capacity)	$O(n)$ dynamic (elements + pointers)

A few things worth noting:

- Array-based push is $O(1)$ amortized if you support dynamic resizing (doubling the array when full). Individual resizes cost $O(n)$, but averaged over many operations it stays $O(1)$.
- Linked list operations have slightly higher constant overhead due to memory allocation on each push/enqueue and deallocation on each pop/dequeue.
- Array-based structures tend to perform better in practice because of CPU cache effects, even though the Big-O is identical.

Error handling in stacks and queues

Robust implementations need to handle edge cases gracefully. Here are the main ones to watch for:

- Underflow (empty stack/queue): Always check if the structure is empty before calling pop or dequeue. If it's empty, throw an exception or return a sentinel value depending on your language and design.
- Overflow (array-based only): Check if the structure is full before push or enqueue. Either throw an exception, or resize the underlying array if your implementation supports it.
- Null elements: Decide upfront whether your stack or queue allows null values. If it does, you can't use null as a "not found" return value for pop/dequeue, so you'll need exceptions or wrapper types instead.
- Concurrent access: If multiple threads share a stack or queue, you need synchronization (locks, semaphores, or lock-free designs) to prevent race conditions. Without it, two threads could pop the same element or corrupt internal pointers.

Unit 4 – Recursion and its Applications

4.1 Principles of Recursion

Introduction to Recursion

Recursion is a programming technique where a function solves a problem by calling itself with a smaller or simpler input. It's one of the core ideas in computer science because many data structures (trees, graphs, nested lists) and algorithms (sorting, searching, backtracking) are naturally recursive. Once you understand how recursion works, you'll see it everywhere.

Components of Recursion

Every recursive function has two essential parts: a base case and a recursive case.

The base case is the condition that stops the recursion. It handles the simplest version of the problem and returns a result directly, without making another function call. Without a base case, the function would call itself forever and eventually crash with a stack overflow.

The recursive case is where the function calls itself with a modified input that moves closer to the base case. Each call shrinks the problem until it's simple enough for the base case to handle.

Here's how these parts look in a classic example, the factorial function:

```
def factorial(n):  
    if n == 0:      # base case  
        return 1  
    else:          # recursive case  
        return n * factorial(n - 1)
```

- The base case: $n = 0$ returns 1, since $0! = 1$.
- The recursive case: for any $n > 0$, the function computes $n \times (n - 1)!$ by calling itself with $n - 1$.

Each recursive call brings n one step closer to 0, guaranteeing the base case will eventually be reached.

Execution Flow in Recursive Functions

To understand what actually happens when a recursive function runs, you need to understand the call stack. The call stack is a data structure the runtime uses to keep track of active function calls.

Each time a function is called, a new stack frame is pushed onto the call stack. That frame holds the function's local variables and parameters. When the function returns, its frame is popped off the stack, and execution resumes in the calling function.

Here's a step-by-step trace of `factorial(3)`:

1. `factorial(3)` is called. A frame is pushed with $n = 3$. It needs the result of `factorial(2)`.
2. `factorial(2)` is called. A frame is pushed with $n = 2$. It needs the result of `factorial(1)`.
3. `factorial(1)` is called. A frame is pushed with $n = 1$. It needs the result of `factorial(0)`.
4. `factorial(0)` is called. A frame is pushed with $n = 0$. This is the base case, so it returns 1.

Now the stack unwinds as each frame computes its result:

- factorial(0) returns 1
- factorial(1) returns $1 \times 1 = 1$
- factorial(2) returns $2 \times 1 = 2$
- factorial(3) returns $3 \times 2 = 6$

The final result is 6. Notice that the call stack grew four frames deep before any values were returned. Tracing through the stack like this is the best way to debug and reason about recursive code.

Recursion vs. Iteration

Any problem solvable with recursion can also be solved with iteration (loops), and vice versa. The question is which approach fits the problem better.

Recursion strengths: - More intuitive for problems that have a naturally recursive structure (tree traversal, nested data) - Often produces shorter, more readable code for these problems - Mirrors the mathematical definition of many functions (factorial, Fibonacci)

Recursion drawbacks: - Higher memory usage because every recursive call adds a frame to the call stack - Risk of stack overflow if the recursion goes too deep (e.g., factorial(100000) in Python will crash) - Function call overhead makes it slower than iteration in many cases

Iteration strengths: - More memory-efficient since it doesn't build up a call stack - No risk of stack overflow - Generally faster due to less overhead per step

Iteration drawbacks: - Can be harder to write and read for problems with inherent recursive structure - May require you to manually manage a stack data structure to simulate recursion

Some recursive solutions can be converted to iterative ones. Tail recursion is a special case where the recursive call is the very last operation in the function, which some languages (not Python or Java) can optimize into a loop automatically. You can also convert recursion to iteration by maintaining your own explicit stack.

The choice depends on the problem structure, the language you're using, and whether performance or clarity matters more in that context.

Applications of Recursive Solutions

Recursion shows up across many categories of problems in data structures and algorithms.

Problems with inherent recursive structure: - Traversing or searching trees (binary trees, file system directories) - Exploring all combinations or permutations of a set - Divide-and-conquer algorithms like Merge Sort and Quick Sort, which split the input, recurse on each half, and combine results

Problems divisible into identical subproblems: - Computing factorials, Fibonacci numbers, and other mathematical sequences - Generating all valid combinations of parentheses - The Tower of Hanoi puzzle, where moving n disks requires solving the problem for $n - 1$ disks first

Backtracking problems, where you explore possible paths and abandon invalid ones: - Solving a Sudoku puzzle by trying digits and backtracking when a conflict arises - Finding all paths through a maze - The N-Queens problem (placing N non-attacking queens on an $N \times N$ chessboard)

Problems that rely on maintaining a history of states: - Depth-First Search (DFS) in graphs, which naturally uses the call stack to remember where to backtrack - Evaluating nested arithmetic expressions with parentheses

4.2 Recursive Problem-Solving Techniques

Recursive Problem-Solving Techniques

Recursion solves complex problems by breaking them into smaller subproblems and calling the same function with reduced input until hitting a base case. Mastering these techniques is essential because they underpin major algorithms (merge sort, quick sort, binary search) and are the natural way to traverse trees and graphs.

Recursive Solutions for Classic Problems

Each classic recursive problem follows the same pattern: define a base case that stops the recursion, and a recursive case that moves toward it.

Factorial

The factorial of n is defined as $n \times (n - 1)!$, with the base case $0! = 1$. Each call multiplies n by the result of calling itself with $n - 1$.

$$\bullet \text{ factorial}(5) = 5 \times \text{factorial}(4) = 5 \times 4 \times \text{factorial}(3) = \dots = 120$$

The recursion "unwinds" once it hits the base case, and the multiplications resolve on the way back up the call stack.

Fibonacci Sequence

The Fibonacci sequence is defined as $F_n = F_{n-1} + F_{n-2}$, with base cases $F_0 = 0$ and $F_1 = 1$. Each call branches into two recursive calls, which is what makes naive Fibonacci so expensive (more on that in the complexity section).

$$\bullet \text{ fib}(5) = \text{fib}(4) + \text{fib}(3) = (\text{fib}(3) + \text{fib}(2)) + (\text{fib}(2) + \text{fib}(1)) = \dots = 5$$

Notice that $\text{fib}(2)$ and $\text{fib}(3)$ get computed multiple times. This redundancy is why the naive recursive Fibonacci has exponential time complexity, and why memoization or dynamic programming is often used to fix it.

Binary Search

Binary search works on a sorted array by repeatedly cutting the search space in half.

1. Compare the target to the middle element.
2. If they match, return the index (base case: element found).
3. If the target is smaller, recurse on the left half. If larger, recurse on the right half.
4. If $\text{low} > \text{high}$, the element isn't in the array (base case: search space exhausted).

Example: Searching for 42 in [1, 5, 17, 42, 98]. The middle element is 17, so you recurse on [42, 98]. The middle is now 42, and you've found the target.

Divide-and-Conquer in Recursion

Divide-and-conquer is a general strategy with three steps:

1. Divide the problem into smaller subproblems.
2. Conquer each subproblem recursively (base cases handle the smallest subproblems directly).

3. Combine the subproblem solutions into a solution for the original problem.

The "combine" step is what distinguishes different divide-and-conquer algorithms from each other.

Merge Sort

1. Divide the array into two halves.
2. Recursively sort each half (base case: a subarray of size 1 is already sorted).
3. Merge the two sorted halves by comparing their elements one by one and placing them in order.

The key work happens in the merge step. You walk through both sorted halves with two pointers, always picking the smaller element, which produces a fully sorted result.

Quick Sort

1. Choose a pivot element (common choices: last element, first element, or a random element).
2. Partition the array so that all elements smaller than the pivot come before it, and all larger elements come after it.
3. Recursively sort the subarrays on either side of the pivot (base case: subarrays of size 0 or 1).

Unlike merge sort, quick sort does its heavy lifting in the partition step rather than a merge step. The pivot ends up in its final sorted position after partitioning.

Recursion for Data Structure Traversal

Trees and graphs have a naturally recursive structure: each node connects to other nodes, which connect to more nodes. Recursion maps directly onto this.

Tree Traversal

For a binary tree, there are three standard recursive traversals. They differ only in when you process the current node relative to its children:

- Preorder: Visit root → recurse left → recurse right
- Inorder: Recurse left → visit root → recurse right
- Postorder: Recurse left → recurse right → visit root

For a tree with root 1, left subtree [2, 4, 5], and right subtree [3, 6, 7]: - Preorder yields: [1, 2, 4, 5, 3, 6, 7] - Inorder yields: [4, 2, 5, 1, 6, 3, 7] - Postorder yields: [4, 5, 2, 6, 7, 3, 1]

Inorder traversal of a binary search tree gives you the elements in sorted order, which is a useful property to remember.

Graph Traversal

- Depth-first search (DFS) recursively explores as far as possible along one path before backtracking. The recursive function takes the graph, the current node, and a set of visited nodes to prevent infinite loops in graphs with cycles.
- Breadth-first search (BFS) explores all neighbors at the current depth before moving deeper. BFS is typically implemented with a queue and iteration rather than recursion, though a recursive adaptation is possible.

Example: In a graph where A connects to B and C, B connects to D, DFS starting from A might visit [A, B, D, C] (going deep along $A \rightarrow B \rightarrow D$ before backtracking to C).

Complexity Analysis of Recursive Algorithms

Time Complexity

To analyze a recursive algorithm, count how many recursive calls are made and how much work each call does.

- Factorial: n recursive calls, each doing $O(1)$ work $\rightarrow O(n)$
- Naive Fibonacci: The call tree branches twice at each level, leading to $O(2^n)$ time. This is why the naive version is impractical for large n .
- Binary search: Each call halves the input $\rightarrow O(\log n)$
- Merge sort: Splits into two halves at each level, with $O(n)$ merge work per level $\rightarrow O(n \log n)$

The Master Theorem is a formal tool for solving these kinds of recurrence relations, which you'll likely encounter if your course covers recurrences in depth.

Space Complexity

Recursive algorithms use stack space proportional to the maximum depth of the call stack.

- Factorial: The call stack goes n levels deep $\rightarrow O(n)$ space
- Binary search: The stack goes $\log n$ levels deep $\rightarrow O(\log n)$ space
- Naive Fibonacci: The maximum call depth is $n \rightarrow O(n)$ space (not $O(2^n)$, because branches resolve before new ones open)

Tail recursion is a special case where the recursive call is the very last operation in the function. Some compilers and languages can optimize tail-recursive functions to reuse the same stack frame, effectively turning the recursion into a loop and reducing space to $O(1)$. Standard factorial is not tail-recursive (it multiplies after the recursive call returns), but it can be rewritten with an accumulator parameter to become tail-recursive.

4.3 Tail Recursion and Optimization

Tail recursion is a powerful technique that optimizes recursive functions by making the recursive call the final operation. It allows for more efficient memory usage and can prevent stack overflow errors in deep recursions.

By converting recursive functions to tail-recursive form, programmers can improve performance and write cleaner code. However, it's important to consider limitations like language support and debugging challenges when implementing tail recursion.

Tail Recursion

Conversion to tail-recursive form

- Tail recursion performs the recursive call as the last operation in the function without any additional computations or modifications to the result
- Convert recursive functions to tail-recursive form by:

- Ensuring the recursive call is the last operation performed
- Performing any necessary computations or operations before the recursive call
- Passing intermediate results as arguments to the recursive call, allowing it to directly return the final result
- Example: Converting a recursive factorial function to tail-recursive form
- Original recursive function: `def factorial(n): if n == 0: return 1 else: return n * factorial(n - 1)`
- Tail-recursive version: `def factorial_tail(n, acc=1): if n == 0: return acc else: return factorial_tail(n - 1, n * acc)`

Benefits of tail recursion

- Tail recursion enables optimization by the compiler or interpreter known as tail call optimization (TCO)
- TCO eliminates the need for additional stack frames for each recursive call by reusing the current stack frame
- Reduces memory overhead and prevents stack overflow errors in deep recursions
- Benefits include:
 - Improved memory efficiency with constant stack space usage regardless of recursion depth
 - Enhanced performance by eliminating overhead of creating and managing multiple stack frames
 - Cleaner and more readable code by allowing a more declarative and expressive programming style

Implementation and performance comparison

- Implement tail-recursive solutions by:
 - Identifying the base case(s) that terminate the recursion
 - Determining the recursive case(s) and ensuring the recursive call is the last operation
 - Passing necessary accumulators or intermediate results as arguments to the recursive call
- Compare performance of tail-recursive and non-tail-recursive versions by:
 - Measuring execution time for various input sizes
 - Analyzing memory usage and stack space consumption
 - Considering the maximum depth of recursion supported
- Example: Comparing tail-recursive and non-tail-recursive Fibonacci implementations
 - Non-tail-recursive version: `def fibonacci(n): if n <= 1: return n else: return fibonacci(n - 1) + fibonacci(n - 2)`
 - Tail-recursive version: `def fibonacci_tail(n, a=0, b=1): if n == 0: return a else: return fibonacci_tail(n - 1, b, a + b)`
 - The non-tail-recursive version has exponential time complexity and may lead to stack overflow for large values of n , while the tail-recursive version has linear time complexity and can handle larger values efficiently

Limitations of tail recursion

- Tail recursion has limitations and may not be applicable in certain scenarios:
- Not all recursive problems can be easily converted to tail-recursive form (tree traversals, divide-and-conquer algorithms)
- Relies on support for tail call optimization (TCO) by the language or compiler, which may not be available in all cases
- Debugging and stack trace analysis can be more challenging due to optimized stack frames
- Consider the following when deciding to use tail recursion:
 - Nature of the problem and its suitability for tail recursion
 - Support for TCO in the target programming language or compiler
 - Readability and maintainability of the code
 - Need for debugging and stack trace analysis during development

4.4 Applications of Recursion in Data Structures

Recursive Algorithms for Tree Traversal

Recursion is a natural fit for data structures that are themselves defined recursively. A tree, for instance, is a node connected to smaller trees. A linked list is a node connected to a smaller linked list. This self-similar structure is exactly why recursive algorithms work so well here: the structure of the code mirrors the structure of the data.

That said, recursive solutions come with real trade-offs in time and space complexity, particularly around call stack usage. This section covers the major applications of recursion in trees, graphs, and linked lists, along with how to analyze their efficiency.

Recursive algorithms for tree traversal

Tree traversals visit every node in a tree exactly once. The three classic orderings differ only in when you process the current node relative to its children:

- Preorder (root-left-right): Visit the root first, then recursively traverse the left subtree, then the right subtree. Useful for creating a copy of the tree or generating a prefix expression.
- Inorder (left-root-right): Recursively traverse the left subtree, visit the root, then traverse the right subtree. On a binary search tree (BST), this visits nodes in sorted (ascending) order.
- Postorder (left-right-root): Recursively traverse the left subtree, then the right subtree, then visit the root. This is the right choice when you need to process children before their parent, such as when deleting or freeing nodes.

All three traversals follow the same recursive pattern. The only thing that changes is where the "visit" step happens relative to the two recursive calls.

Recursion in tree and graph problems

Height calculation of a binary tree

The height of a tree is the number of edges on the longest path from the root to a leaf. Recursion handles this cleanly because the height of any node depends on the heights of its subtrees:

1. If the node is null (empty tree), return -1 (or 0 if you define height as the number of nodes on the longest path rather than edges; be consistent with your course's convention).
2. Recursively calculate the height of the left subtree.
3. Recursively calculate the height of the right subtree.
4. Return $\max(\text{leftHeight}, \text{rightHeight}) + 1$.

This runs in $O(n)$ time since every node is visited once.

Shortest path finding using recursive DFS

Recursive depth-first search can find paths in a graph by exploring as deep as possible before backtracking:

- From the starting vertex, recursively explore each unvisited neighbor, building up the current path.
- When you reach the target vertex, compare the current path length to the shortest found so far and update if shorter.
- Pruning optimization: If the current path length already exceeds the best known shortest path, stop exploring that branch early.

Note that DFS is not the most efficient way to find shortest paths in general. BFS finds shortest paths in unweighted graphs in $O(V + E)$ time, and algorithms like Dijkstra's handle weighted graphs better. Recursive DFS for shortest paths is mainly useful as a learning exercise or in constrained scenarios.

Recursion for data structure implementation

Recursive linked list operations

Since a linked list is a node followed by a smaller linked list, many operations translate directly into recursive solutions:

- Insertion: Recurse through the list until you reach the desired position, then create and link the new node.
- Deletion: Recurse until you find the target node, then update the previous node's pointer to skip over it.
- Reversal: Recurse to the end of the list, then on each return, flip the next pointer so each node points back to the one before it. The original tail becomes the new head (returned up through every call).

Recursive construction of a binary tree from a sorted array

Building a balanced BST from a sorted array is a classic divide-and-conquer application:

1. If the subarray is empty, return null (base case).
2. Find the middle element and create a new node with that value (this becomes the root of the current subtree).
3. Recursively build the left subtree from the left half of the subarray (elements before the middle).
4. Recursively build the right subtree from the right half (elements after the middle).

Because you split the array in half each time, the resulting tree is balanced, and the algorithm runs in $O(n)$ time.

Efficiency of recursive algorithms

Time complexity analysis

To determine the time complexity of a recursive algorithm:

1. Write the recurrence relation that describes the running time. For example, binary tree traversal gives $T(n) = 2T(n/2) + O(1)$.
2. Solve the recurrence using the Master Theorem, recursion tree method, or substitution method.
3. Express the result in Big O notation (e.g., $O(n)$, $O(n \log n)$).

Space complexity analysis

Recursive algorithms use stack space proportional to the maximum depth of recursion:

- Each recursive call adds a frame to the call stack, consuming memory for local variables, parameters, and the return address.
- The maximum recursion depth determines the stack space. For a balanced binary tree, that's $O(\log n)$. For a skewed tree or a linked list, it's $O(n)$.
- Deep recursion on large inputs can cause a stack overflow if the recursion depth exceeds the system's stack size limit.

When analyzing space complexity, account for both the call stack and any additional data structures your algorithm uses. The total space is the sum of both.

Unit 5 – Trees and Binary Trees

5.1 Tree Terminology and Properties

Tree Fundamentals

Trees are the go-to data structure for representing hierarchical relationships. A tree is made up of nodes connected by edges, where each node stores data and references to other nodes. If you've ever looked at a file system, an org chart, or an HTML document, you've seen a tree structure in action.

This section covers the core terminology and properties you'll need before working with binary trees, BSTs, and tree algorithms.

Fundamental Concepts of Trees

A tree is a hierarchical, acyclic data structure. "Acyclic" means there are no cycles: you can't start at a node, follow edges, and end up back where you started. Edges are directed, always pointing from a parent down to a child.

Here's the key vocabulary for the parts of a tree:

- Node: the basic unit in a tree, storing data and references to other nodes
- Root node: the topmost node in the tree; it has no parent
- Parent node: a node that directly connects to and sits above another node
- Child node: a node that directly connects to and sits below another node
- Sibling nodes: nodes that share the same parent
- Leaf (external node): a node with zero children
- Internal node: a node with at least one child
- Edge: a connection between two nodes, representing a parent-child relationship
- Subtree: any node in the tree together with all of its descendants; every node is the root of its own subtree

One property worth remembering: a tree with n nodes always has exactly $n - 1$ edges. This follows directly from the fact that every node except the root has exactly one incoming edge from its parent.

Properties and Characteristics of Trees

These numeric properties show up constantly in algorithm analysis, so make sure you can distinguish them:

- Depth of a node: the number of edges from the root down to that node
- Nodes at the same depth are said to be on the same level
- The root is at level 0, its children at level 1, and so on
- Height of a node: the number of edges on the longest path from that node down to a leaf
- A leaf node has height 0

- Height of a tree: the height of the root node, which equals the maximum depth of any node in the tree
- Degree of a node: the number of children that node has
- Degree of a tree: the maximum degree of any node in the tree

Watch out for an easy mix-up: depth is measured from the root down to a node, while height is measured from a node down to its deepest leaf. They go in the same direction (toward leaves), but depth is relative to the root and height is relative to the node itself.

Tree Types and Relationships

Types of Trees

Binary trees restrict each node to at most two children, called the left child and right child. Two important variants:

- Strict (proper) binary tree: every node has either 0 or 2 children. No node has exactly one child.
- Complete binary tree: every level is fully filled except possibly the last, and the last level has all its nodes packed as far left as possible. This is the shape you see in a binary heap.

A binary search tree (BST) is a binary tree that enforces an ordering rule:

1. Every node in the left subtree has a key less than the current node's key.
2. Every node in the right subtree has a key greater than the current node's key.
3. Both the left and right subtrees are themselves valid BSTs.

This ordering is what makes search efficient: at each node, you can eliminate an entire subtree.

A balanced tree keeps the height difference between the left and right subtrees of any node to at most one. AVL trees and Red-Black trees are classic examples. Balancing guarantees $O(\log n)$ time for search, insertion, and deletion, whereas an unbalanced BST can degrade to $O(n)$ in the worst case (think of inserting sorted data into a plain BST, which produces a straight chain of nodes).

Relationships Between Tree Nodes

- Path: a sequence of nodes connected by edges from an ancestor down to a descendant. The path length is the number of edges in that sequence.
- Distance: the number of edges on the (unique) path between two nodes.
- Lowest Common Ancestor (LCA): the deepest node that is an ancestor of both given nodes. The LCA is useful for computing the distance between two nodes: find the LCA, then add the distance from the LCA to each node.

Traversal means visiting every node in the tree exactly once, in a defined order. For binary trees, the three standard depth-first traversals are:

1. Preorder (Root → Left → Right): visit the current node first, then recurse into the left subtree, then the right subtree.
2. Inorder (Left → Root → Right): recurse into the left subtree, visit the current node, then recurse into the right subtree. For a BST, inorder traversal visits nodes in sorted (ascending) order.

3. Postorder (Left → Right → Root): recurse into the left subtree, then the right subtree, then visit the current node. This is useful when you need to process children before their parent (e.g., deleting a tree or evaluating an expression tree).

5.2 Binary Tree Representation and Traversals

Binary Tree Representation

Binary trees organize data in a hierarchy where each node has at most two children (left and right). This structure is the foundation for binary search trees, heaps, and expression trees, so understanding how to represent and traverse them is essential for nearly everything else in this unit.

Linked vs. Array Representation

Linked representation is the more common approach. Each node is an object (or struct) containing three things: the data, a reference to the left child, and a reference to the right child. If a child doesn't exist, that reference is null.

- Nodes are dynamically allocated, so the tree can grow or shrink as needed
- Inserting or deleting a node only requires updating a few references
- Each node carries extra memory overhead for storing two pointers

Array representation stores the entire tree in a single array using index math to find relatives:

- For a node at index i :
- Left child is at $2i + 1$
- Right child is at $2i + 2$
- Parent is at $\lfloor (i - 1)/2 \rfloor$
- The root always sits at index 0

Array representation works well for complete or nearly complete binary trees (like heaps), where most positions are filled. For sparse trees with many missing nodes, you end up wasting array slots on empty positions.

When to use which: Linked representation is the default choice for general binary trees. Array representation is preferred for heaps and other complete trees where the index math gives you fast parent/child lookups without pointer overhead.

Binary Tree Traversals

Traversal means visiting every node in the tree exactly once in a specific order. The three classic depth-first traversals differ only in when you process the current node relative to its children.

The Three Depth-First Traversals

Preorder (NLR: Node, Left, Right)

1. Process the current node
2. Recursively traverse the left subtree
3. Recursively traverse the right subtree

Preorder visits the root before anything else, so it's useful for copying a tree or generating a prefix expression from an expression tree.

Inorder (LNR: Left, Node, Right)

1. Recursively traverse the left subtree
2. Process the current node
3. Recursively traverse the right subtree

For a binary search tree, inorder traversal produces elements in sorted (non-decreasing) order. This is the traversal you'll reach for most often when working with BSTs.

Postorder (LRN: Left, Right, Node)

1. Recursively traverse the left subtree
2. Recursively traverse the right subtree
3. Process the current node

Postorder processes children before their parent, making it the right choice for deleting a tree (you free children before the parent) or evaluating a postfix expression.

Quick example: Given a tree with root A, left child B, right child C: - Preorder: A, B, C - Inorder: B, A, C - Postorder: B, C, A

Recursive vs. Iterative Implementation

Recursive algorithms are the natural fit for tree traversals because trees are recursive structures.

- Base case: if the node is null, return (do nothing)
- Recursive case: process the node and recurse on left/right children in the appropriate order
- Clean and easy to read, but each recursive call adds a frame to the call stack

Iterative algorithms replace the implicit call stack with an explicit stack (or queue) data structure. These are trickier to write but avoid stack overflow on very deep trees.

- Preorder (iterative): Push the root onto a stack. Pop a node, process it, then push its right child followed by its left child. (Right goes first so left gets popped first.)
- Inorder (iterative): Start at the root and push all left descendants onto the stack. Pop a node, process it, then move to its right child and repeat.
- Postorder (iterative): The trickiest of the three. One common approach uses two stacks: run a modified preorder (node, right, left) pushing results onto a second stack, then pop the second stack to get postorder.

Applications of Tree Traversals

Evaluating expression trees Arithmetic expressions like $(3 + 5) \times 2$ can be stored as binary trees where operators are internal nodes and operands are leaves. A postorder traversal evaluates the expression bottom-up: compute each subtree's value before applying the parent operator.

Serialization and deserialization To save a tree to a file or send it over a network, you convert it to a linear format (a string or array) using preorder or postorder traversal, with a special marker (like # or null) for

missing children. To reconstruct the tree, you read the sequence back in the same traversal order.

Computing tree properties - Height: Use postorder traversal. The height of a node is $\max(\text{height of left subtree, height of right subtree}) + 1$. A null node has height -1 (or 0 , depending on your convention). - Balance check: A tree is balanced if, for every node, the left and right subtree heights differ by at most 1 . You can check this during a postorder height computation. - BST validation: An inorder traversal of a valid BST produces a strictly sorted sequence. If any element is out of order, the tree isn't a valid BST. - Lowest common ancestor (LCA): Recurse through the tree. If the current node matches either target, return it. Otherwise, search both subtrees. If both subtrees return non-null results, the current node is the LCA.

5.3 Tree Applications and Implementations

Tree Data Structures

Tree-based data structures

A Binary Search Tree (BST) is built from nodes that each have at most two children: left and right. The core rule is simple: every value in the left subtree is less than the node, and every value in the right subtree is greater. This ordering property is what makes searching efficient, since you can eliminate half the tree at each step.

AVL Trees solve a major BST weakness. A regular BST can become lopsided (imagine inserting $1, 2, 3, 4, 5$ in order), which kills performance. AVL trees are self-balancing BSTs that track a balance factor for each node, defined as the height of the left subtree minus the height of the right subtree. Whenever an insertion or deletion causes any node's balance factor to fall outside the range $[-1, 1]$, the tree performs rotations to restore balance:

- Left rotation: fixes right-heavy imbalance
- Right rotation: fixes left-heavy imbalance
- Left-right rotation: fixes left-child-right-heavy imbalance (double rotation)
- Right-left rotation: fixes right-child-left-heavy imbalance (double rotation)

Heaps are complete binary trees that satisfy the heap property. In a Max Heap, every node's value is greater than or equal to its children's values, so the maximum sits at the root. In a Min Heap, every node's value is less than or equal to its children's values, so the minimum sits at the root.

Heaps are commonly implemented using an array rather than linked nodes. The parent-child relationships are maintained purely through index math:

- Parent of node at index i : $\lfloor (i - 1)/2 \rfloor$
- Left child: $2i + 1$
- Right child: $2i + 2$

This array-based layout is memory-efficient and cache-friendly since no pointers are needed.

Efficiency of tree operations

Operation	BST (Average)	BST (Worst)	AVL Tree	Heap
Search	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$
Insertion	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Deletion	$O(\log n)$	$O(n)$	$O(\log n)$	$O(\log n)$
Find min/max	$O(n)$	$O(n)$	$O(\log n)$	$O(1)$

The BST worst case of $O(n)$ happens when the tree degenerates into a linked list (all nodes on one side). That's exactly the problem AVL trees fix by guaranteeing balance.

Heaps give you $O(1)$ access to the min or max (it's always at the root), but they're not designed for general search. The heap property only guarantees a parent-child ordering, not a left-right ordering like a BST, so you can't prune your search path.

- Heapify up (used during insertion): place the new element at the end of the array, then swap it upward with its parent until the heap property is restored.
- Heapify down (used during root deletion): replace the root with the last element, then swap it downward with the appropriate child until the heap property is restored.

Applications of tree structures

Hierarchical Data Representation is the most intuitive use of trees. File systems model directories as internal nodes and files as leaves. XML/HTML documents nest elements inside each other, forming a tree. Organization charts map reporting relationships the same way. Trees naturally capture any data with parent-child relationships.

Expression Evaluation uses Binary Expression Trees. Leaf nodes hold operands (numbers or variables), and internal nodes hold operators (+, -, *, /). For example, the expression $(3 + 5) * 2$ becomes a tree with * at the root, + as the left child (with leaves 3 and 5), and 2 as the right child. Evaluating the tree with a post-order traversal* processes operands before their operator, which naturally respects order of operations.

Decision Making uses Decision Trees where each internal node represents a condition (e.g., "Is temperature > 30°?") and each leaf represents an outcome or classification. You traverse from the root, following branches based on your answers, until you reach a leaf. These are foundational in machine learning for classification tasks.

Implementation of tree algorithms

Traversal algorithms define the order in which you visit every node. Each has distinct use cases:

1. Pre-order (root, left, right): useful for copying a tree or generating a prefix expression
2. In-order (left, root, right): on a BST, this visits nodes in sorted ascending order
3. Post-order (left, right, root): useful for deleting a tree (children before parent) or evaluating expression trees
4. Level-order (breadth-first): visits nodes level by level using a queue; useful for finding the shortest path or printing the tree by depth

Pre-order, in-order, and post-order are typically implemented with recursion (or an explicit stack). Level-order uses a queue: enqueue the root, then repeatedly dequeue a node, process it, and enqueue its children.

Balancing algorithms in AVL trees trigger after every insertion or deletion. The process works like this:

1. Perform the standard BST insertion or deletion
2. Walk back up the tree to the root, updating each ancestor's balance factor
3. At the first node where the balance factor becomes +2 or -2, determine the rotation case
4. Apply the appropriate rotation (single or double) to restore balance

Memory considerations matter in pointer-based tree implementations. Each node is dynamically allocated and holds pointers to its children (and sometimes its parent). When deleting nodes, you need to deallocate memory to prevent leaks. For performance-critical applications, memory pools or custom allocators can reduce allocation overhead and fragmentation compared to calling malloc/new for every individual node.

Unit 6 – Binary Search Trees

6.1 BST Properties and Operations

Binary Search Tree Fundamentals

Binary search trees (BSTs) organize data into a node-based hierarchy where every element has a defined position based on its value. This structure lets you find, add, or remove elements quickly by making simple comparisons at each level, rather than scanning through everything.

The key difference between a BST and a plain binary tree is the ordering rule that every node must follow. In a balanced BST, this ordering gives you $O(\log n)$ performance for core operations. But if the tree becomes unbalanced, performance can degrade all the way to $O(n)$.

BST Properties

A valid BST must satisfy all of the following:

- Left subtree rule: Every node in a node's left subtree has a key less than that node's key.
- Right subtree rule: Every node in a node's right subtree has a key greater than that node's key.
- Recursive structure: Both the left and right subtrees must themselves be valid BSTs. This isn't just about immediate children; the property must hold for every descendant.

Each node stores a key (its data), a reference to a left child, and a reference to a right child. Some implementations also include a parent reference to simplify certain operations like deletion.

BSTs vs Regular Binary Trees

Regular binary trees (like heaps or expression trees) impose no ordering relationship between parent and child keys. A heap, for example, enforces a vertical rule (parent $\geq$ children in a max-heap) but has no left-vs-right ordering.

BSTs enforce the left-less-than, right-greater-than property, which is what makes efficient searching possible. The tradeoff is that regular binary trees can be kept structurally balanced (complete, full, or perfect) by design, while a BST's shape depends entirely on the order elements are inserted. Insert the values 1, 2, 3, 4, 5 in that order, and your BST degenerates into a linked list.

BST Operations and Complexity

Search

Searching a BST follows a straightforward comparison-and-branch pattern:

1. Start at the root node.
2. Compare the search key to the current node's key. - If they're equal, you've found it. Done. - If the search key is less, move to the left child. - If the search key is greater, move to the right child.
3. Repeat step 2. If you reach a null reference (no child where you need to go), the value isn't in the tree.

Each comparison eliminates an entire subtree from consideration. In a balanced tree with 1,000 nodes, you'd make roughly 10 comparisons instead of 1,000.

Insertion

Insertion works almost identically to search. You're looking for the spot where the new value would be if it existed:

1. Starting at the root, compare the new key with the current node's key.
2. If the new key is less, move left. If greater, move right.
3. Repeat until you hit a null reference. That null spot is where the new node belongs.
4. Create a new node and attach it there.

The new node is always inserted as a leaf. This is simple but means the tree's shape depends on insertion order.

Deletion

Deletion is the trickiest operation because removing a node from the middle of the tree requires maintaining the BST property. There are three cases:

- Case 1 (leaf node): Simply remove it. No children to worry about.
- Case 2 (one child): Replace the deleted node with its single child. The BST property is preserved because that child's entire subtree was already on the correct side.
- Case 3 (two children): You can't just remove the node without breaking the tree structure. Instead:
 1. Find the inorder successor, which is the smallest key in the right subtree. (You find it by going right once, then left as far as possible.)
 2. Copy the inorder successor's key into the node you want to delete.
 3. Delete the inorder successor from the right subtree. Since the inorder successor has at most one child (a right child), this reduces to Case 1 or Case 2.

You could also use the inorder predecessor (largest key in the left subtree) instead of the inorder successor. Both approaches are valid.

Time Complexity

All three core operations depend on the height of the tree, since you traverse from root toward a leaf.

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

In a balanced BST, the height is approximately $\log_2 n$, so each step roughly halves the remaining nodes. In the worst case, the tree is completely skewed (every node has only one child), making the height equal to n . At that point, the BST offers no advantage over a linked list.

This worst-case behavior is exactly why self-balancing BST variants (AVL trees, red-black trees) exist: they guarantee $O(\log n)$ height regardless of insertion order.

6.2 BST Implementation and Analysis

Binary Search Tree (BST) Implementation and Analysis

Binary Search Trees (BSTs) organize data in a hierarchical structure that enables efficient searching, insertion, and deletion. Each node has at most two children, with the left subtree containing smaller values and the right subtree containing larger values. This ordering property is what makes BSTs powerful: you can eliminate half the remaining nodes at each step, much like binary search on a sorted array.

BST operations run in $O(\log n)$ time when the tree is balanced, but can degrade to $O(n)$ when the tree becomes skewed. Understanding why that happens and how to implement BST operations correctly is central to this unit.

Implementation of BST

A BST is built from nodes, and the tree itself is managed by a class that holds a reference to the root.

Node structure: - Key stores the node's value (integer, string, or any comparable data type) - Left pointer references the left child (values smaller than the key) - Right pointer references the right child (values larger than the key)

BST class: - Root is a pointer to the root node of the tree. An empty tree has a null root.

Insert Operation

Insertion places a new key in the correct position so the BST property is preserved.

1. Start at the root.
2. Compare the new key to the current node's key.
3. If the new key is less, move to the left child. If greater, move to the right child.
4. Repeat until you reach a null pointer.
5. Create a new node and attach it at that null position.

For example, inserting 5 into a BST with root 8 and left child 3: you compare 5 to 8 (go left), then compare 5 to 3 (go right), and 3's right child is null, so 5 becomes 3's right child.

Search Operation

Searching follows the same traversal logic as insertion.

1. Start at the root.
2. Compare the target key to the current node's key.
3. If they match, return the node.
4. If the target is less, move left. If greater, move right.
5. If you reach a null pointer, the key is not in the tree. Return null.

Delete Operation

Deletion is the trickiest operation because removing a node must preserve the BST property. There are three cases:

1. No children (leaf node): Simply remove the node by setting its parent's pointer to null.
2. One child: Replace the node with its single child by updating the parent's pointer to skip over the deleted node.
3. Two children: Find the inorder successor (the smallest node in the right subtree). Copy the successor's key into the node being deleted, then delete the successor node. The successor is guaranteed to have at most one child (no left child), so deleting it falls into case 1 or 2.

The inorder successor works because it's the next-largest value in the tree, so swapping it in preserves the BST ordering. You could also use the inorder predecessor (largest node in the left subtree) with the same logic.

Complexity Analysis of BST Operations

Time complexity:

All three core operations (insert, search, delete) have a time complexity of $O(h)$, where h is the height of the tree. The height determines how many comparisons you make in the worst case.

- Balanced BST: $h = \log_2 n$, so operations run in $O(\log n)$. A tree with 1,000,000 nodes would need at most ~20 comparisons.
- Skewed BST: $h = n$, so operations run in $O(n)$. This happens when you insert already-sorted data (e.g., 1, 2, 3, 4, 5), which produces a tree that looks like a linked list.

Space complexity:

- Storing the tree itself requires $O(n)$ space for n nodes.
- Recursive implementations add $O(h)$ space on the call stack. In a skewed tree, that's $O(n)$ stack frames, which could cause a stack overflow for large inputs. Iterative implementations avoid this overhead.

BST vs Other Data Structures

Arrays and linked lists: - Searching an unsorted array or linked list takes $O(n)$. A balanced BST cuts that to $O(\log n)$. - Insertion at the end of an array or linked list is $O(1)$, but inserting in sorted order costs $O(n)$ for arrays (shifting elements). BSTs handle sorted insertion naturally. - BSTs use more memory per element than arrays because each node stores two pointers in addition to the key.

Hash tables: - Hash tables offer $O(1)$ average time for search, insert, and delete, which is faster than a BST's $O(\log n)$. - However, BSTs maintain key ordering. You can do an inorder traversal to get all elements in sorted order, or efficiently answer range queries (e.g., "find all keys between 10 and 50"). Hash tables cannot do this.

Balanced BSTs (AVL trees, Red-Black trees): - These are BSTs with extra rules that prevent skewing. They guarantee $O(\log n)$ for all operations, eliminating the worst-case $O(n)$ problem. - The tradeoff is added complexity: AVL trees store a balance factor at each node and perform rotations on insert/delete. Red-Black trees store a color bit and have their own rebalancing rules. Both use slightly more space and time per operation to maintain balance.

Advantages and Disadvantages of BSTs

Advantages: - $O(\log n)$ average-case search, insert, and delete, which is much better than the $O(n)$ of arrays and linked lists - Elements stay ordered by key, enabling inorder traversal (sorted output), range queries, and efficient min/max lookups - Dynamic size with no need to pre-allocate memory like arrays

Disadvantages: - Performance depends entirely on tree shape. Inserting sorted data without balancing produces $O(n)$ operations, which is no better than a linked list. - Slower than hash tables when you don't need ordering - Higher memory overhead than arrays due to storing two pointers per node

When to use a BST: - You need data in sorted order (e.g., maintaining a sorted list of usernames) - You need range queries (e.g., finding all students with grades between 80 and 90) - You need efficient min/max lookups (e.g., a priority-based system) - You need a symbol table where ordered iteration matters (e.g., a compiler's symbol table)

When to consider alternatives: - Order doesn't matter and you want the fastest lookups possible: use a hash table (e.g., implementing a cache or dictionary) - Data is static and won't change: use a sorted array with binary search, which avoids pointer overhead - You need guaranteed $O(\log n)$ and can't risk skewed input: use a balanced BST like an AVL tree or Red-Black tree

6.3 Self-Balancing BSTs Introduction

Introduction to Self-Balancing BSTs

A standard BST can become lopsided depending on the order you insert elements, which kills its performance. Self-balancing BSTs fix this by automatically restructuring themselves after insertions and deletions, guaranteeing $O(\log n)$ time complexity for core operations.

The Problem with Unbalanced BSTs

To understand why self-balancing trees matter, consider what happens when you insert already-sorted data (say, 1, 2, 3, 4, 5) into a plain BST. Each new node becomes the right child of the previous one, and you end up with a structure that looks and behaves exactly like a linked list. The tree's height becomes n instead of the ideal $\log n$.

This means:

- Search degrades from $O(\log n)$ to $O(n)$
- Insertion degrades from $O(\log n)$ to $O(n)$
- Deletion degrades from $O(\log n)$ to $O(n)$

At that point, you've lost every advantage a BST is supposed to give you.

Unbalanced trees typically result from:

- Inserting elements in sorted or nearly sorted order (ascending or descending)
- Repeated deletions from one side of the tree, causing lopsided structure

What Self-Balancing Means

A self-balancing BST automatically adjusts its structure after every insertion or deletion so that the height stays proportional to $\log n$. "Balance" here means minimizing the height difference between the left and right subtrees of every node. The result is consistent $O(\log n)$ performance regardless of what order you insert or delete elements.

Types and Principles of Self-Balancing BSTs

AVL Trees

AVL trees (named after Adelson-Velsky and Landis) track a balance factor at each node, defined as:

Balance Factor = $\text{height}(\text{left subtree}) - \text{height}(\text{right subtree})$

The rule is simple: every node's balance factor must be -1 , 0 , or 1 . If an insertion or deletion causes any node's balance factor to fall outside that range, the tree performs rotations to fix it.

AVL trees use four rotation types:

1. Right rotation — fixes a left-heavy imbalance (left child's left subtree is too tall)
2. Left rotation — fixes a right-heavy imbalance (right child's right subtree is too tall)
3. Left-right rotation — a left rotation on the left child, then a right rotation on the node (fixes a left-heavy node where the left child is right-heavy)
4. Right-left rotation — a right rotation on the right child, then a left rotation on the node (fixes a right-heavy node where the right child is left-heavy)

AVL trees are strictly balanced, so lookups are very fast. The tradeoff is that insertions and deletions may require more rotations compared to other self-balancing trees.

Red-Black Trees

Red-Black trees take a more relaxed approach to balance. Each node is colored either red or black, and the tree enforces these properties:

1. The root is always black.
2. No red node can have a red child (no two consecutive red nodes on any path).
3. Every path from a given node down to any of its NULL leaves contains the same number of black nodes (this count is called the black-height).

After an insertion or deletion violates these rules, the tree restores them using a combination of rotations and color flips (swapping a node's color between red and black).

Because Red-Black trees allow slightly more imbalance than AVL trees, they do fewer restructuring operations on average during insertions and deletions. That's why many standard library implementations (like Java's `TreeMap` and C++'s `std::map`) use Red-Black trees under the hood.

Balancing Mechanisms: Core Principles

All self-balancing BSTs share a few key ideas, even though the specific rules differ:

- Height control — The goal is always to keep tree height close to $\log n$. AVL trees enforce this strictly; Red-Black trees enforce it loosely (the height is at most $2\log_2(n + 1)$).
- Rotations — The fundamental tool for restructuring. A left rotation moves a node's right child up to take its place; a right rotation moves the left child up. These preserve BST ordering while changing the tree's shape.
- Local restructuring — Rebalancing happens locally, near the point of insertion or deletion, and propagates upward only as needed. You don't rebuild the whole tree.

AVL vs. Red-Black at a glance: AVL trees give faster lookups (stricter balance, shorter height). Red-Black trees give faster insertions/deletions (fewer rotations on average). Choose based on whether your workload is read-heavy or write-heavy.

Unit 7 – Balanced Trees (AVL and Red-Black Trees)

7.1 AVL Trees: Properties and Operations

AVL trees are self-balancing binary search trees that automatically maintain a balanced structure after every insertion and deletion. This balance guarantee is what makes them useful: while a plain BST can degrade to $O(n)$ height if you insert sorted data, an AVL tree always stays at $O(\log n)$ height, keeping search, insertion, and deletion fast.

AVL Tree Properties

An AVL tree is a BST with one extra rule: for every node in the tree, the heights of its left and right subtrees can differ by at most 1. This is called the height-balance property.

- Balance factor of a node is defined as $height(left\ subtree) - height(right\ subtree)$
- For a tree to be AVL-valid, every node's balance factor must be -1 , 0 , or 1
- Stated formally: $|height(left\ subtree) - height(right\ subtree)| \leq 1$ for all nodes

Why does this matter? This constraint keeps the tree's overall height at $O(\log n)$ for n nodes. You can prove this using Fibonacci-like minimum AVL trees: the minimum number of nodes in an AVL tree of height h satisfies $N(h) = N(h-1) + N(h-2) + 1$, which grows exponentially. That means height grows only logarithmically with n .

A null (empty) subtree has height -1 by convention, and a leaf node has height 0 . Getting this convention right matters when you're computing balance factors during implementation.

AVL Tree Operations

Insertion

Insertion is a two-phase process:

1. Insert like a normal BST. Walk down the tree comparing keys, and place the new node at the appropriate leaf position.
2. Walk back up to the root, updating heights and checking the balance factor at each ancestor node.
3. If any node's balance factor becomes -2 or 2 , the tree is unbalanced at that node. Perform the appropriate rotation (see below) to fix it.

One key detail: after an insertion, you only ever need one rotation (single or double) to restore balance. Once you fix the first unbalanced ancestor, all ancestors above it will also be balanced. This is not true for deletion.

Deletion

Deletion works similarly but with an important difference:

1. Delete like a normal BST. If the node has two children, replace it with its in-order successor (or predecessor), then delete that successor node.

2. Walk back up from the deleted node's parent to the root, updating heights and checking balance factors.
3. If any node is unbalanced, perform the appropriate rotation.

Unlike insertion, a single deletion can require multiple rotations along the path to the root. Each rotation can reduce the height of a subtree, which may unbalance an ancestor further up.

Time Complexity

All three core operations run in $O(\log n)$ worst-case time:

- Search: Standard BST search. Since the height is $O(\log n)$, you visit at most $O(\log n)$ nodes.
- Insertion: The BST insertion walk is $O(\log n)$. Rebalancing requires at most one rotation, which is $O(1)$. Total: $O(\log n)$.
- Deletion: The BST deletion walk is $O(\log n)$. Rebalancing may require up to $O(\log n)$ rotations in the worst case, but each rotation is $O(1)$, so the total is still $O(\log n)$.

Rotations for AVL Rebalancing

Rotations are the mechanism that restores balance. There are four cases, determined by where the imbalance is and which direction the heavy subtree leans. Think of it as: first identify the unbalanced node, then look at its heavy child's balance factor to pick the right rotation.

Single Rotations:

- Right Rotation (RR): The node has balance factor $+2$ and its left child has balance factor $+1$ or 0 (left-left case). The left child becomes the new subtree root, and the original node becomes its right child.
- Left Rotation (LL): The node has balance factor -2 and its right child has balance factor -1 or 0 (right-right case). The right child becomes the new subtree root, and the original node becomes its left child.

Double Rotations:

•

Left-Right Rotation (LR): The node has balance factor $+2$ and its left child has balance factor -1 (left-right case). 1. Perform a left rotation on the left child. 2. Perform a right rotation on the unbalanced node.

•

Right-Left Rotation (RL): The node has balance factor -2 and its right child has balance factor $+1$ (right-left case). 1. Perform a right rotation on the right child. 2. Perform a left rotation on the unbalanced node.

A quick way to remember: if the imbalance and the heavy grandchild are on the same side (left-left or right-right), you need a single rotation. If they're on opposite sides (left-right or right-left), you need a double rotation that first straightens the path, then rotates.

7.2 Red-Black Trees: Properties and Operations

Red-Black Tree Properties

Red-Black trees are self-balancing binary search trees that attach a color (red or black) to every node. This coloring scheme enforces a set of invariants that keep the tree roughly balanced, guaranteeing $O(\log n)$ time for search, insertion, and deletion. They're used heavily in practice: the C++ `std::map`, Java's `TreeMap`, and the Linux kernel's scheduler all rely on Red-Black trees under the hood.

Compared to AVL trees, Red-Black trees allow slightly less strict balancing, which means they can be faster for workloads with many insertions and deletions (fewer rotations on average). The tradeoff is that lookups can be marginally slower because the tree can be a bit taller.

The Five Properties

Every valid Red-Black tree satisfies all five of these properties simultaneously:

1. Node coloring: Every node is colored either red or black.
2. Root property: The root is always black.
3. Leaf property: All leaf nodes (NIL/null pointers) are considered black. These aren't "real" nodes with data; they're sentinel placeholders that simplify the logic.
4. Red property: A red node cannot have a red child. Put another way, no two red nodes can appear consecutively on any path. This is sometimes called the no double-red rule.
5. Black-height property: Every path from a given node down to any of its descendant NIL leaves passes through the same number of black nodes. This count is called the node's black-height.

Property 5 is the one that actually enforces balance. Combined with property 4, it guarantees that the longest root-to-leaf path is at most twice the length of the shortest. Here's why: the shortest possible path is all black nodes, and the longest path alternates red and black. Since both paths have the same number of black nodes, the longest path can only squeeze in one red node between each pair of black nodes, making it at most $2 \times$ the shortest.

This means a Red-Black tree with n nodes has a height of at most $2\log_2(n + 1)$, keeping all operations logarithmic.

Red-Black Tree Operations

Insertion

New nodes are always inserted as red. Why red? Inserting a black node would immediately violate the black-height property (property 5) on every path through that node. A red insertion only risks violating the red property (property 4), and only if the parent is also red.

Step-by-step insertion process:

1. Perform a standard BST insertion. Color the new node red.
2. If the new node is the root, recolor it black. Done.
3. If the parent is black, no property is violated. Done.
4. If the parent is red, you have a double-red violation. Now check the uncle (the parent's sibling):

Case A: Uncle is red (recoloring case) 1. Recolor the parent and uncle to black. 2. Recolor the grandparent to red. 3. Move your focus up to the grandparent and repeat from step 2 (the grandparent might now cause a new double-red violation with its parent).

Case B: Uncle is black or NIL (rotation case)

The specific rotation depends on the shape of the path from grandparent to parent to new node:

Configuration	Fix
Left-Left (LL): Parent is left child, new node is left child	Right-rotate grandparent, swap colors of parent and grandparent
Left-Right (LR): Parent is left child, new node is right child	Left-rotate parent (converts to LL case), then apply LL fix
Right-Right (RR): Parent is right child, new node is right child	Left-rotate grandparent, swap colors of parent and grandparent
Right-Left (RL): Parent is right child, new node is left child	Right-rotate parent (converts to RR case), then apply RR fix

After a rotation case, the fix is local and doesn't propagate further up the tree. The recoloring case (Case A) is the one that can ripple upward to the root.

Deletion

Deletion is the trickiest operation. The high-level idea: remove the node using standard BST deletion, then fix any property violations that result.

Step-by-step deletion process:

1. Find the node to delete. If it has two children, replace its value with its in-order successor (smallest node in the right subtree) or in-order predecessor (largest in the left subtree), then delete that successor/predecessor node instead. This reduces the problem to deleting a node with at most one child.
2. Let the node actually being removed be V , and its child (which replaces it) be U .

Simple cases: - If V is red, just remove it. No properties are violated since removing a red node doesn't change any path's black-height. - If V is black but U is red, replace V with U and recolor U black. This preserves the black-height.

Hard case: both V and U are black (or U is NIL)

This creates a double-black at U 's position, meaning that path is now one black node short. Fixing this requires examining the sibling of the double-black node and its children. The cases involve combinations of:

- Sibling is red: rotate to convert into a case where the sibling is black.
- Sibling is black with at least one red child: perform a rotation and recoloring to push an extra black node into the deficient path.
- Sibling is black with two black children: recolor the sibling red, pushing the double-black problem up to the parent. If the parent is red, recolor it black and you're done. If the parent is black, it becomes the new double-black and you repeat.

The exact rotation direction depends on whether the sibling is a left or right child, mirroring the insertion cases. In the worst case, the fix propagates up to the root, but it still takes at most $O(\log n)$ steps.

Deletion has more cases than insertion, and most courses don't expect you to memorize every sub-case perfectly. Focus on understanding the strategy: you're redistributing black nodes across paths to restore the black-height property, using rotations and recoloring as your tools.

Time Complexity

Operation	Worst Case	Why
Search	$O(\log n)$	Tree height is at most $2\log_2(n + 1)$, so the longest search path is logarithmic
Insertion	$O(\log n)$	BST insertion takes $O(\log n)$; fix-up involves at most $O(\log n)$ recolorings and at most 2 rotations
Deletion	$O(\log n)$	BST deletion takes $O(\log n)$; fix-up involves at most $O(\log n)$ recolorings and at most 3 rotations

The constant number of rotations per insertion/deletion is a practical advantage over AVL trees, which may require $O(\log n)$ rotations during deletion. Recolorings are cheap (just flipping a bit), so even when they propagate to the root, the real cost stays low.

7.3 Comparison of AVL and Red-Black Trees

Properties and Balancing Mechanisms

Properties of AVL vs Red-Black Trees

Both AVL and Red-Black trees are self-balancing binary search trees, but they take fundamentally different approaches to staying balanced.

AVL trees enforce a strict height invariant. For every node, the balance factor (height of left subtree minus height of right subtree) must be -1 , 0 , or 1 . Whenever an insertion or deletion violates this, the tree immediately rebalances through rotations.

Red-Black trees enforce balance through a set of color-based rules:

- Every node is colored red or black.
- The root is always black.
- No red node can have a red child (the "no double-red" rule).
- Every path from the root to a NULL leaf passes through the same number of black nodes (the black-height property).

These rules guarantee that the longest path in the tree is at most twice the length of the shortest path, bounding the height to at most $2\log_2(n + 1)$. That's looser than AVL's guarantee, but still $O(\log n)$.

Trade-offs in Tree Balancing

Because AVL trees keep a tighter balance, they tend to be shorter for the same number of nodes. This makes search slightly faster in practice. The cost is more rotations during insertions and deletions. AVL rebalancing uses four rotation types: left rotation, right rotation, left-right (double) rotation, and right-left (double) rotation.

Red-Black trees allow more slack in their structure, so they need fewer structural changes per update. Rebalancing after an insertion requires at most 2 rotations, and after a deletion at most 3 rotations. The rest of the fix-up work is just recoloring nodes, which is cheap. This makes Red-Black trees faster for workloads dominated by insertions and deletions.

Both structures guarantee $O(\log n)$ worst-case time for search, insertion, and deletion. The difference is in the constant factors: AVL trees have a smaller constant for search, while Red-Black trees have a smaller constant for modifications.

Performance and Trade-offs

Scenarios for Tree Selection

Choosing between these two structures comes down to your workload pattern:

- AVL trees are the better pick when searches heavily outnumber insertions and deletions. Think of read-heavy applications like in-memory dictionaries or lookup tables in real-time systems where you need the tightest possible search times.
- Red-Black trees are the better pick when you have frequent insertions and deletions mixed with searches. This is why many standard library implementations use Red-Black trees (e.g., `std::map` in C++, `TreeMap` in Java). They handle update-heavy workloads with less overhead from rotations.

On memory, the difference is small. AVL nodes typically store a balance factor or height (an integer), while Red-Black nodes store a color bit. In practice, the color bit can be packed into an existing pointer field, so Red-Black trees can use marginally less extra space per node. This rarely drives the decision, though.

Implementation and Performance Analysis

AVL implementation centers on maintaining and checking the balance factor after every modification:

1. Insert or delete a node using standard BST logic.
2. Walk back up the tree toward the root, updating the balance factor at each ancestor:
 $\text{balanceFactor} = \text{height}(\text{leftSubtree}) - \text{height}(\text{rightSubtree})$
3. If any node's balance factor becomes $+2$ or -2 , apply the appropriate rotation (left, right, left-right, or right-left) to restore balance.

Red-Black implementation focuses on restoring color properties after modifications:

1. Insert a new node colored red (this preserves the black-height property automatically).
2. If the new node's parent is also red, you have a double-red violation. Fix it by either recoloring the parent, uncle, and grandparent, or performing 1-2 rotations, depending on the configuration.
3. Deletion fix-ups follow a similar pattern but are more involved, handling cases based on the sibling node's color and its children's colors.

Empirical results consistently match the theoretical expectations. In benchmarks across varying dataset sizes:

- AVL trees perform search operations slightly faster due to their shorter height.
- Red-Black trees complete sequences of mixed insertions and deletions faster due to fewer rotations per operation.

- Both scale as $O(\log n)$, so the performance gap between them stays proportionally small as n grows.

The bottom line: if your application is search-dominated, lean toward AVL. If it's update-heavy or you want a well-tested general-purpose balanced BST, Red-Black trees are the standard choice.

Unit 8 – Priority Queues and Heaps

8.1 Priority Queue ADT

Priority Queue Fundamentals

A priority queue is an abstract data type that manages elements based on assigned priorities rather than insertion order. Where a regular queue follows First-In-First-Out (FIFO), a priority queue always serves the highest-priority element first. This makes it foundational for algorithms where processing order depends on some ranking, not just when something arrived.

Priority Queue ADT and Operations

The priority queue ADT models a collection where every element has an associated priority value. You can think of it as a queue that constantly reorders itself so the most important item is always at the front.

Core operations:

- `Insert(element, priority)` — adds an element with its associated priority to the queue
- `ExtractMax()` / `ExtractMin()` — removes and returns the element with the highest (max-heap) or lowest (min-heap) priority. Which one you use depends on whether you're working with a max-priority queue or a min-priority queue.
- `Peek()` — returns the highest- or lowest-priority element without removing it
- `IsEmpty()` — returns true if the queue contains no elements
- `Size()` — returns the number of elements currently in the queue

The distinction between max and min variants matters. A max-priority queue extracts the largest key first (useful for task scheduling by importance). A min-priority queue extracts the smallest key first (useful in Dijkstra's algorithm, where you want the vertex with the shortest tentative distance).

Priority vs. Regular Queues

A regular queue follows FIFO: elements are processed in the exact order they were added. There's no concept of priority at all.

A priority queue processes elements based on their priority value. A high-priority element added last will still be served before a low-priority element added first. When two elements share the same priority, they're typically processed in FIFO order among themselves (though this isn't guaranteed by all implementations).

The key difference: a regular queue cares about when you arrived. A priority queue cares about how important you are.

Real-World Applications

Priority queues show up in many core algorithms and systems:

- OS task scheduling — The operating system assigns priority levels to processes. Higher-priority tasks (like handling user input) get CPU time before lower-priority background tasks.
- Event-driven simulations — Events are stored with timestamps as priorities. The simulation always processes the event with the earliest timestamp next, using a min-priority queue.

- Dijkstra's shortest path algorithm — At each step, the algorithm extracts the unvisited vertex with the smallest tentative distance. A min-priority queue makes this extraction $O(\log n)$ instead of $O(n)$.
- Huffman coding — Builds an optimal prefix code for data compression by repeatedly extracting the two lowest-frequency nodes and merging them. A min-priority queue drives this process.
- Prim's minimum spanning tree — Selects the next edge with the minimum weight connecting the tree to a new vertex, again relying on a min-priority queue for efficient extraction.

Time Complexity Across Implementations

How you implement a priority queue dramatically affects performance. Here's the comparison:

Binary heap implementation (the standard choice):

1. Insert: $O(\log n)$
2. ExtractMax/ExtractMin: $O(\log n)$
3. Peek: $O(1)$

Sorted array implementation:

1. Insert: $O(n)$ — you need to shift elements to maintain sorted order
2. ExtractMax/ExtractMin: $O(1)$ — the extreme element is already at one end
3. Peek: $O(1)$

Unsorted array implementation:

1. Insert: $O(1)$ — just append to the end
2. ExtractMax/ExtractMin: $O(n)$ — requires a linear scan to find the extreme element
3. Peek: $O(n)$ — also requires a linear scan

Regular queue (array or linked list):

1. Enqueue: $O(1)$
2. Dequeue: $O(1)$
3. Front: $O(1)$

The binary heap is the go-to implementation because it balances both insert and extract at $O(\log n)$. A sorted array gives you fast extraction but slow insertion. An unsorted array gives you fast insertion but slow extraction. The heap avoids that tradeoff by keeping a partially ordered structure, which is enough to always know the max or min without fully sorting everything.

8.2 Heap Data Structure and Operations

Heap Data Structure

A binary heap is a tree-based data structure that keeps elements in a partially sorted order, guaranteeing fast access to either the minimum or maximum element. Heaps are the go-to backing structure for priority queues and play a central role in algorithms like heapsort and Dijkstra's shortest path.

Structure and Properties of Binary Heaps

A binary heap must satisfy two properties:

1. Complete binary tree: Every level is fully filled except possibly the last, and the last level is filled left to right. This shape guarantee is what makes the array-based implementation work so cleanly.
2. Heap ordering property: Every parent node has a specific relationship with its children (defined by whether it's a min-heap or max-heap).

One thing that trips people up: there's no ordering between sibling nodes. Just because the left child is smaller than the right child in one subtree doesn't mean that holds everywhere. The only guarantee is the parent-child relationship.

Min-Heaps vs Max-Heaps

- Min-heap: Every parent is less than or equal to its children. The root always holds the smallest value. This is the standard choice when you need to repeatedly extract the minimum, like in task scheduling where the lowest-priority (or earliest-deadline) task should come out first.
- Max-heap: Every parent is greater than or equal to its children. The root always holds the largest value. Useful when you need the maximum quickly, such as in event-driven simulations where the highest-priority event fires next.

The operations work identically for both; the only difference is the direction of the comparison (less-than vs. greater-than).

Array-Based Implementation of Heaps

Because a binary heap is always a complete binary tree, you can store it in a flat array with no pointers at all. The tree structure is encoded entirely through index arithmetic.

For a node at index i (using 0-based indexing):

Relationship	Index Formula
Left child	$2i + 1$
Right child	$2i + 2$
Parent	$\lfloor (i - 1)/2 \rfloor$

The root sits at index 0. Elements fill the array left to right, level by level. For example, a min-heap containing [2, 5, 3, 8, 7] maps to this tree:

```
      2      index 0
     /\
    5 3      indices 1, 2
   /\
  8 7      indices 3, 4
```

Node 5 is at index 1. Its left child is at $2(1) + 1 = 3$ (which is 8), and its parent is at $\lfloor (1 - 1)/2 \rfloor = 0$ (which is 2). This direct index calculation is what makes heap operations fast with no pointer chasing.

Heap Operations

Insertion (Heapify-Up)

To insert a new element into a min-heap:

1. Place the new element at the end of the array (the next open position in the last level).
2. Compare it with its parent at index $\lfloor (i - 1)/2 \rfloor$.
3. If the new element is smaller than its parent, swap them.
4. Repeat steps 2-3, moving upward, until the element is greater than or equal to its parent, or it becomes the root.

Walkthrough: Insert 1 into the min-heap [2, 5, 3, 8, 7].

- Place 1 at index 5. Parent is at index $\lfloor (5 - 1)/2 \rfloor = 2$, which holds 3. Since $1 < 3$, swap. Array: [2, 5, 1, 8, 7, 3].
- Now 1 is at index 2. Parent is at index $\lfloor (2 - 1)/2 \rfloor = 0$, which holds 2. Since $1 < 2$, swap. Array: [1, 5, 2, 8, 7, 3].
- Now 1 is at index 0 (the root). Done.

Deletion (Heapify-Down)

Deletion always removes the root (the min or max element). To delete from a min-heap:

1. Replace the root with the last element in the array, then shrink the array by one.
2. Compare the new root with both of its children.
3. Swap it with the smaller child if the heap property is violated. (In a max-heap, you'd swap with the larger child.)
4. Repeat steps 2-3, moving downward, until the element is smaller than or equal to both children, or it reaches a leaf.

Walkthrough: Delete the root from [1, 5, 2, 8, 7, 3].

- Remove 1, move last element (3) to the root. Array: [3, 5, 2, 8, 7].
- Compare 3 with children: left child 5 (index 1), right child 2 (index 2). Smaller child is 2. Since $3 > 2$, swap. Array: [2, 5, 3, 8, 7].
- Now 3 is at index 2. Its left child would be at index $2(2) + 1 = 5$, which is out of bounds. 3 is a leaf. Done.

Peek

Peek simply returns the element at index 0 without modifying anything.

Time Complexity of Heap Operations

Operation	Time Complexity	Why
Insertion	$O(\log n)$	Heapify-up traverses at most the height of the tree
Deletion	$O(\log n)$	Heapify-down traverses at most the height of the tree
Peek	$O(1)$	Direct array access at index 0
Build heap	$O(n)$	More efficient than n individual insertions

The height of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$, which is where the $O(\log n)$ bound comes from.

Build heap deserves a closer look. You might expect $O(n \log n)$ since you're calling heapify-down on multiple nodes. But most nodes are near the bottom of the tree and only need to sift down a short distance. Mathematically, the work sums to $O(n)$. The key detail: you start heapifying from the last non-leaf node (index $\lfloor n/2 \rfloor - 1$) and work backward to the root, calling heapify-down on each. This bottom-up approach is what makes it linear.

8.3 Heap Implementation and Applications

Heap Data Structure

Heaps are data structures built to handle priority-based operations efficiently. Their core strength is finding and removing the highest (or lowest) priority element in $O(\log n)$ time, which makes them the go-to choice for priority queues, scheduling systems, and sorting.

This section covers how to build heaps from arrays, how to use them for finding k-th elements and sorting, and when heaps are (and aren't) the right tool.

Building Heaps from Arrays

There are two distinct approaches to building a heap, and they have different time complexities. This is a common exam topic, so make sure you understand the difference.

Approach 1: Bottom-up heapify (the faster way)

The heapify procedure converts an existing array into a heap in-place. It works by treating the array as a complete binary tree and fixing heap violations from the bottom up.

1. Start at the last non-leaf node (index $\lfloor n/2 \rfloor - 1$ in a zero-indexed array).
2. For each node, compare it with its children and swap with the larger child (max-heap) or smaller child (min-heap) if the heap property is violated.
3. After a swap, continue sifting down the swapped node until it lands in a valid position.
4. Move to the next node toward the root and repeat.

The time complexity is $O(n)$, not $O(n \log n)$. This surprises a lot of people. The reason is that most nodes are near the bottom of the tree and only need to sift down a short distance. A formal proof uses the fact that the sum of heights across all nodes is $O(n)$.

Approach 2: Repeated insertion (the slower way)

You can also build a heap by starting with an empty heap and inserting elements one at a time. Each insertion places the new element at the next available position and then sifts it up to restore the heap property.

- Each insertion takes $O(\log n)$ in the worst case.
- Over n elements, this gives $O(n \log n)$ total.

Bottom-up heapify is $O(n)$. Repeated insertion is $O(n \log n)$. If you're asked to "build a heap" on an exam, the $O(n)$ bottom-up method is typically what's expected.

Applications of Heaps

Finding the k -th largest element

A min-heap of size k can find the k -th largest element in an unsorted array without fully sorting it.

1. Insert the first k elements into a min-heap.
2. For each remaining element in the array, compare it to the root (the current minimum in the heap).
3. If the element is larger than the root, remove the root and insert the new element. This keeps the heap holding the k largest elements seen so far.
4. After processing all elements, the root of the min-heap is the k -th largest.

Time complexity: $O(n \log k)$, since each heap operation on a size- k heap costs $O(\log k)$, and you process n elements. When k is much smaller than n , this is significantly faster than sorting the whole array.

Finding the k -th smallest element works the same way but uses a max-heap of size k . You replace the root whenever you encounter a smaller element. The time complexity is still $O(n \log k)$.

Heaps in Sorting Algorithms

Heap Sort

Heap Sort uses a max-heap to sort an array in ascending order. It works in two phases:

1. Build a max-heap from the input array using bottom-up heapify ($O(n)$).
 2. Extract elements one by one: - Swap the root (the maximum) with the last element in the unsorted portion. - Shrink the heap size by one (the swapped element is now in its final sorted position). - Sift the new root down to restore the heap property.
 3. Repeat step 2 until the heap is empty.
- Time complexity: $O(n \log n)$ in all cases (best, average, worst).
 - Space complexity: $O(1)$ auxiliary space since it sorts in-place.
 - Heap Sort is not stable: equal elements may change their relative order.

Priority Queues

A priority queue is an abstract data type where each element has an associated priority, and you always retrieve the element with the highest (or lowest) priority first. Heaps are the standard implementation.

- Min-heap backs a minimum priority queue (lowest value = highest priority).

- Max-heap backs a maximum priority queue (highest value = highest priority).
- Insertion: $O(\log n)$
- Extract-min/max: $O(\log n)$
- Peek at min/max: $O(1)$

Pros and Cons of Heaps

Advantages:

- $O(\log n)$ insertion and extraction of the min/max element.
- $O(1)$ peek at the min/max.
- Heap Sort is in-place with guaranteed $O(n \log n)$ worst-case performance.
- Simple array-based implementation with no pointers or extra memory overhead.

Disadvantages:

- Searching for an arbitrary element is $O(n)$ since the heap only guarantees a partial ordering (parent vs. children), not a full one.
- Updating an element's priority (decrease-key or increase-key) requires knowing the element's position. Without an auxiliary index, finding it costs $O(n)$.
- Heap Sort is not stable, so equal elements may be reordered.
- In practice, heaps have worse cache performance than algorithms like quicksort due to the non-sequential memory access pattern of sifting through tree levels.

When to choose a heap:

- Priority-based access is the primary operation (scheduling, event-driven simulations, Dijkstra's algorithm).
- You need guaranteed $O(n \log n)$ worst-case sorting and can't afford quicksort's $O(n^2)$ worst case.

When something else fits better:

- If you need to search for specific elements frequently, a balanced BST gives you $O(\log n)$ search.
- If stable sorting matters (preserving the order of equal elements), use Merge Sort instead of Heap Sort.
- If your data changes frequently and you need ordered traversal, a BST or balanced tree is more appropriate.

Unit 9 – Hash Tables and Hash Functions

9.1 Hash Function Design and Properties

Hash Function Fundamentals

Hash functions map arbitrary-sized data to fixed-size values, enabling the constant-time lookups that make hash tables so fast. Understanding how to design a good hash function is the foundation for everything else in this unit, because a poorly chosen hash function can degrade a hash table from $O(1)$ average-case operations all the way down to $O(n)$.

Purpose of Hash Functions

A hash function takes a key of any size and produces a fixed-size integer called a hash code. That hash code determines where the key-value pair gets stored in the table's underlying array.

This is what makes hash tables useful for structures like dictionaries and sets: instead of searching through every element, you compute the hash code, jump straight to the right slot, and get average-case $O(1)$ for insertion, deletion, and lookup.

Beyond hash tables, hash functions show up in database indexing, caching layers, checksums for data integrity, and cryptographic applications. The design goals differ across these contexts, but the core idea is the same: turn data into a compact, fixed-size representation.

Uniformity and Efficiency

Two properties matter most when evaluating a hash function: uniformity and efficiency.

Uniformity means the hash function distributes keys evenly across the output range. When distribution is uneven, multiple keys land in the same slot, causing collisions. A collision is when two different keys produce the same hash code. More collisions means more time spent resolving them, which erodes the constant-time guarantee.

You can assess uniformity using statistical tools like the chi-squared test (comparing observed vs. expected slot occupancy) or by monitoring the load factor (number of stored elements divided by table size).

Efficiency means the hash code is fast to compute. If computing the hash takes longer than the lookup it enables, you've defeated the purpose. The ideal is $O(1)$ time and minimal extra space per computation.

Here are three common hash function families and how they balance these properties:

- Division method: $h(k) = k \bmod m$. Simple and fast, but if m has small prime factors or is a power of 2, keys with patterns in their lower bits will cluster into the same slots. Choosing m as a prime not close to a power of 2 helps.
- Multiplication method: $h(k) = \lfloor m(kA \bmod 1) \rfloor$, where A is a constant between 0 and 1. This gives better distribution than division and is less sensitive to the choice of m . Knuth suggested $A \approx 0.6180339887$ (the golden ratio conjugate) as a good default.
- Universal hashing: Instead of picking one fixed function, you randomly select from a family of hash functions at runtime. This provides theoretical worst-case guarantees on collision probability, regardless of the input distribution. The trade-off is slightly more setup overhead.

Designing and Optimizing Hash Functions

Designing for Different Data Types

Different data types need different hashing strategies.

Integers are the simplest case. Modulo-based methods like $h(k) = k \bmod m$ work directly. You can also use bit-level operations (XOR, shifts, rotations) to scramble the bits and break up patterns in the input. For example, many practical hash functions XOR a key with a right-shifted version of itself before taking the modulus, which helps spread out clustered values.

Strings require more thought because they're variable-length sequences of characters. Two common approaches:

- Polynomial rolling hash: Treat each character as a coefficient of a polynomial. For a string $s_0s_1\dots s_{n-1}$, compute $h = s_0 \cdot p^{n-1} + s_1 \cdot p^{n-2} + \dots + s_{n-1}$, where p is a small prime (like 31 or 37). This is fast to compute incrementally and produces good distribution.
- CRC (Cyclic Redundancy Check): Computes the remainder of a polynomial division over the bit representation. More commonly used for error detection, but sometimes applied in hash table contexts.

Composite objects (like a struct with multiple fields) are hashed by combining the hash codes of their individual fields. A typical approach: multiply a running hash by a prime number, then XOR or add the next field's hash code. This prevents different field orderings from producing the same result.

When choosing a hash function, consider:

- The expected key distribution. If keys are nearly sequential integers, a simple modulus on a prime works fine. If keys are skewed or patterned, you need a function that actively scrambles those patterns.
- Your collision resolution scheme. Chaining is more tolerant of collisions than open addressing, so open addressing benefits more from a higher-quality hash function.
- Security requirements. For untrusted input (like user-supplied keys in a web server), use cryptographic hash functions like SHA-256 to prevent adversarial collision attacks. These are much slower but resistant to deliberate exploitation.

Complexity vs. Performance Trade-offs

There's a real tension between hash function quality and speed.

A simple function like $h(k) = k \bmod m$ computes in nanoseconds but may produce uneven distributions for certain inputs. A more sophisticated function (multiple rounds of bit mixing, or universal hashing) distributes keys more evenly but takes longer per call. Since the hash function runs on every insertion and lookup, even small per-call costs add up.

Your collision resolution strategy also factors into this decision:

1. Chaining stores colliding elements in linked lists (or other structures) at each slot. It tolerates higher collision rates gracefully, so you can afford a simpler, faster hash function. The trade-off is extra memory for the list pointers.
2. Open addressing probes alternative slots within the array when a collision occurs. Clusters of occupied slots slow down probing, so you need a hash function with strong uniformity to avoid clustering. Linear probing is especially sensitive to this.

To find the right balance for a specific application, consider:

- The expected number of elements and the table's load factor (keeping it below 0.7 is a common guideline for open addressing)
- Whether keys come from a known, predictable distribution or from arbitrary/adversarial sources
- Actual benchmarking. Profile different hash functions with realistic data. Theoretical uniformity doesn't always predict real-world performance, because cache behavior and branch prediction also matter.

9.2 Collision Resolution Techniques

Hash tables are powerful data structures for fast key-value lookups. However, collisions occur when multiple keys map to the same index. This intro explores collision resolution techniques, crucial for maintaining hash table efficiency.

We'll cover chaining and open addressing methods, examining their pros and cons. We'll also discuss load factor's impact on performance and how to choose the best strategy for your specific use case.

Hash Table Collision Resolution

Collisions in hash tables

- Two or more keys generate the same hash value and map to the same index (bucket)
- Inevitable due to the pigeonhole principle: n items, m slots, $n > m$, at least one slot contains multiple items
- Degrade hash table performance by requiring additional work to resolve collisions
- Collision frequency increases with more elements, reducing hash table efficiency

Collision resolution techniques

- Chaining (separate chaining) stores colliding elements in a linked list at each hash table index
 - Allows unlimited elements, time complexity $O(1 + \alpha)$, α = load factor (n/m)
 - Open addressing stores colliding elements in alternative hash table slots
 - Probing sequence finds the next available slot
1. Linear probing: search next slot linearly $(h(k) + i) \bmod m$, $i = 0, 1, 2, \dots$
 2. Quadratic probing: quadratic function for next slot $(h(k) + i^2) \bmod m$, $i = 0, 1, 2, \dots$
 3. Double hashing: second hash function determines probing sequence $h_2(k) = r - (k \bmod r)$, $r = \text{prime} < \text{table size}$
- Requires load factor < 1 to ensure empty slots for collision resolution

Load factor and efficiency

- Load factor (α) = number of elements (n) / hash table size (m)
- Affects collision probability and hash table efficiency
- Higher load factor $\rightarrow$ more collisions, expected collisions for a key proportional to α
- Lower load factor ($\alpha \leq 0.5$) fewer collisions but more space

- Higher load factor ($\alpha \geq 0.7$) more collisions but efficient space utilization
- Practical load factor 0.5 to 0.75 balances space and time efficiency

Choosing resolution strategies

- Consider expected elements, key distribution, memory constraints, performance requirements
- Chaining suits unknown/growing element count, ample memory, frequent insertions/deletions
- Open addressing suits known element count $\leq$ table size, limited memory, faster search/retrieval
- Hybrid approaches combine chaining and open addressing to balance advantages/disadvantages

9.3 Hash Table Implementation and Analysis

Hash Table Implementation

Hash tables map keys to array indices using hash functions, giving you average-case $O(1)$ lookups, insertions, and deletions. That speed advantage makes them one of the most widely used data structures in practice. The tradeoff: you need to handle collisions (when two keys land on the same index) and manage the table's size as it fills up.

Hash table implementation with arrays

A hash table is backed by an array. A hash function takes a key and returns an index into that array. The quality of the hash function directly determines how evenly keys spread across the table.

Common hash functions:

- Modulo hash function: $hash(key) = key \bmod m$, where m is the table size. Simple and fast. Choosing a prime number for m helps distribute keys more evenly.
- Multiplication hash function: $hash(key) = \lfloor m \cdot (k \cdot A \bmod 1) \rfloor$, where A is a constant between 0 and 1. Knuth suggests $A \approx 0.6180339887$ (the golden ratio conjugate). This method is less sensitive to the choice of m .
- Universal hashing: Randomly select a hash function from a designed family of functions at runtime. This gives provably low expected collisions regardless of the input distribution.

Collision resolution is needed because two different keys can hash to the same index. There are two main strategies:

Separate chaining stores a linked list (or other collection) at each index. When a collision occurs, the new key is appended to the list at that index. Lookups walk the list to find the target key. The lists grow dynamically, so the table never technically "fills up," but long chains hurt performance.

Open addressing stores all entries directly in the array. When a collision occurs, you probe for the next available slot:

1. Linear probing: Check index $h + 1$, then $h + 2$, and so on (wrapping around). Simple, but tends to create clusters of occupied slots, which slows future insertions and lookups.
2. Quadratic probing: Check index $h + 1^2$, then $h + 2^2$, then $h + 3^2$, etc. Reduces clustering compared to linear probing, but can fail to find an empty slot if the table is more than half full (depending on table size).

3. Double hashing: Use a second hash function $h_2(key)$ to determine the step size. Probe sequence: $h + i \cdot h_2(key)$ for $i = 0, 1, 2, \dots$. This produces more uniform probe sequences and avoids clustering.

Time complexity of hash operations

Average case: $O(1)$ for insertion, deletion, and search. This assumes a good hash function that distributes keys uniformly and a load factor that stays reasonable (typically $\alpha < 0.75$).

Worst case: $O(n)$ for all three operations. This happens when every key hashes to the same index. With chaining, you'd be searching a single linked list of length n . With open addressing, you'd probe through nearly every slot.

The load factor α controls how full the table is:

$$\alpha = \frac{n}{m}$$

where n is the number of stored keys and m is the table size. As α increases, collisions become more frequent. For separate chaining, α can exceed 1 (more keys than slots), but performance degrades. For open addressing, α must stay below 1, and performance drops sharply as it approaches 1. A common threshold is $\alpha = 0.75$ for triggering a resize.

Hash Table Optimization and Applications

Rehashing and dynamic resizing

When the load factor crosses a threshold, the table needs to grow. This process is called rehashing:

1. Allocate a new array, typically double the current size.
2. Recompute the hash for every existing key using the new table size (since m changed, all indices change).
3. Insert each key into the new array.
4. Deallocate the old array.

Rehashing is $O(n)$ since you touch every key. But it happens infrequently enough that the amortized cost of insertion remains $O(1)$. Think of it this way: you do n cheap insertions, then one expensive $O(n)$ rehash, which averages out to $O(1)$ per insertion.

Some implementations also shrink the table (e.g., halve it when α drops below 0.25) to avoid wasting memory after many deletions.

Hash tables vs other data structures

	Hash Table	Array	Balanced BST (AVL, Red-Black)
Search	$O(1)$ avg, $O(n)$ worst	$O(n)$ (unsorted), $O(\log n)$ (sorted)	$O(\log n)$ worst
Insert	$O(1)$ avg, $O(n)$ worst	$O(n)$ (shifting)	$O(\log n)$ worst
Delete	$O(1)$ avg, $O(n)$ worst	$O(n)$ (shifting)	$O(\log n)$ worst
Ordered iteration	Not supported	By index only	Yes (in-order traversal)
Range queries	Not efficient	Not efficient	Efficient

The key takeaway: hash tables win on raw speed for individual key lookups. Balanced BSTs win when you need sorted order or range queries (e.g., "find all keys between 50 and 100"). Arrays are best when you're accessing elements by numeric index.

Applications of hash tables

Caching: Store recently or frequently accessed data in a hash table keyed by the request (URL, query, etc.). Instead of hitting a database or making a network call, you check the cache first. If the key exists, you return the cached result in $O(1)$.

Indexing: Build a lookup structure that maps search terms to their locations. For example, an inverted index maps each word in a document collection to the list of documents containing that word. Search engines rely heavily on this pattern.

Counting frequencies: Use a hash table where keys are elements and values are counts. To count word frequencies in a text:

1. For each word, check if it exists in the table.
2. If yes, increment its count.
3. If no, insert it with a count of 1.

This runs in $O(n)$ for n words, since each lookup and update is $O(1)$ on average. The same pattern applies to detecting duplicates, computing histograms, and grouping data by category.

Unit 10 – Graphs and their Representations

10.1 Graph Terminology and Properties

Graph Fundamentals

Graphs model relationships between entities. A graph consists of vertices (also called nodes) and edges that connect them. This structure is flexible enough to represent social networks, road systems, web pages, task dependencies, and much more.

Understanding graph terminology is the foundation for every graph algorithm you'll encounter later. This section covers the core vocabulary: vertices, edges, degree, and the major graph types.

Graph Fundamentals

Vertices and Edges

A graph has two building blocks:

- Vertices (nodes) represent entities or objects.
- Edges represent relationships or connections between vertices.

Edges come in several flavors depending on the problem you're modeling:

- Weighted edges carry a numerical value representing cost, distance, or strength of a connection (e.g., road distance between two cities).
- Unweighted edges simply indicate that a connection exists, with no associated value.
- Directed edges point from one vertex to another, like a one-way street.
- Undirected edges have no direction and represent a two-way connection.

A single graph can be both directed and weighted, or undirected and unweighted. These properties are independent of each other.

Directed vs. Undirected Graphs

Directed graphs (digraphs) have edges with a specific direction, drawn as arrows. The order of the two vertices matters: an edge from A to B is not the same as an edge from B to A.

Real-world examples: - Social media followers: an edge from user A to user B means A follows B, but B doesn't necessarily follow A. - Web page links: an edge from page A to page B means A contains a hyperlink to B. - Task dependencies: an edge from task A to task B means A must be completed before B can start.

Undirected graphs have edges with no direction. If an edge exists between A and B, you can traverse it both ways.

Real-world examples: - Friendships on a social network: if A is friends with B, then B is friends with A. - Road networks (assuming two-way roads): an edge between two cities means you can drive in either

direction. - Collaboration networks: an edge between two researchers means they've co-authored a paper.

Vertex Properties

Degree

The degree of a vertex measures how many edges are connected to it.

In an undirected graph, the degree of a vertex is simply the number of edges incident to it. For example, if vertex C connects to vertices A, B, and D, then C has degree 3.

In a directed graph, degree splits into two counts:

- In-degree: the number of edges pointing into the vertex.
- Out-degree: the number of edges pointing out of the vertex.

The total degree of a vertex in a directed graph equals its in-degree plus its out-degree.

One useful fact: the Handshaking Lemma states that in any undirected graph, the sum of all vertex degrees equals twice the number of edges ($\sum \deg(v) = 2|E|$). This makes sense because each edge contributes 1 to the degree of each of its two endpoints.

Classification of Graph Types

Complete Graphs

A complete graph has an edge between every pair of distinct vertices. If a complete graph has n vertices, the number of edges is:

$$\frac{n(n-1)}{2}$$

For example, a complete graph on 5 vertices has $\frac{5 \times 4}{2} = 10$ edges. Complete graphs are often denoted K_n (so K_5 is the complete graph on 5 vertices).

Bipartite Graphs

A bipartite graph has its vertices split into two disjoint sets, and every edge connects a vertex in one set to a vertex in the other set. No edges exist between vertices within the same set.

Think of it this way: you can color every vertex either red or blue such that no edge connects two vertices of the same color.

Common applications: - Job assignment: one set is workers, the other is tasks, and edges represent which workers can do which tasks. - Course enrollment: one set is students, the other is courses, and edges represent who is enrolled in what.

Connected Graphs

An undirected graph is connected if there's a path between every pair of vertices. Starting from any vertex, you can reach every other vertex by following edges.

A graph that isn't connected is called disconnected and breaks into two or more connected components, each of which is a maximal connected subgraph. For directed graphs, the analogous (and stricter) concept is strongly connected, meaning there's a directed path from every vertex to every other vertex.

10.2 Graph Representation Methods

Graph Representation Methods

Graphs model relationships between entities, and how you store a graph in memory directly affects the speed and memory cost of every operation you run on it. This section covers the four main representation methods, their trade-offs, and how to pick the right one for a given problem.

Adjacency matrices for graph representation

An adjacency matrix stores a graph as a 2D array of size $V \times V$, where V is the number of vertices. Each entry $matrix[i][j]$ is 1 if there's an edge from vertex i to vertex j , and 0 otherwise.

For weighted graphs, you store the edge weight instead of 1, and use a sentinel value (like 0 or ∞) for missing edges.

A few properties to keep in mind:

- Undirected graphs produce symmetric matrices, since an edge between i and j means $matrix[i][j] = matrix[j][i] = 1$. Think of a friendship graph: if Alice is friends with Bob, Bob is friends with Alice.
- Directed graphs can have asymmetric matrices. A hyperlink from page A to page B doesn't mean page B links back to A.
- Space complexity is always $O(V^2)$, no matter how many edges exist. That's fine for dense graphs (like a fully connected social network), but wasteful for sparse ones (like a road network where most cities connect to only a handful of neighbors).

The big advantage of adjacency matrices is constant-time edge lookup: checking whether an edge exists between any two vertices is just an array access at $matrix[i][j]$, which takes $O(1)$.

Adjacency lists vs matrices

An adjacency list uses an array of size V , where each slot holds a list (linked list, dynamic array, etc.) of that vertex's neighbors.

How edges are stored depends on the graph type:

- In an undirected graph, each edge appears twice. An edge between vertices 2 and 5 shows up in vertex 2's list and vertex 5's list.
- In a directed graph, each edge appears once, in the source vertex's list. An edge from vertex 3 to vertex 7 only appears in vertex 3's list.

The space complexity is $O(V + E)$, which is much better than $O(V^2)$ when the graph is sparse.

Time complexity comparison for common operations:

Operation	Adjacency List	Adjacency Matrix
Add an edge	$O(1)$	$O(1)$
Remove an edge	$O(\deg(v))$	$O(1)$
Check if edge exists	$O(\deg(v))$	$O(1)$
Iterate over neighbors of V	$O(\deg(v))$	$O(V)$

A correction from the original: removing or checking an edge in an adjacency list costs $O(\deg(v))$ (the degree of the vertex, i.e., the length of that vertex's neighbor list), not $O(E)$. You only scan one vertex's list, not every edge in the graph.

That last row matters a lot. Iterating over a vertex's neighbors is one of the most common operations in graph algorithms (BFS, DFS, Dijkstra's). With an adjacency list, you touch only the actual neighbors. With a matrix, you scan the entire row of V entries, most of which may be zeros in a sparse graph.

Edge lists and incidence matrices

Edge lists are the simplest representation. You store an array of edge pairs, like $[(1, 2), (2, 3), (1, 4)]$. For weighted graphs, each entry becomes a triple: (u, v, weight) .

- Space complexity: $O(E)$
- Great for algorithms that process edges one at a time, like Kruskal's minimum spanning tree algorithm, which sorts all edges by weight.
- Poor for adjacency queries. Checking whether a specific edge exists or finding all neighbors of a vertex requires scanning the entire list, taking $O(E)$ time.

Incidence matrices use a 2D array of size $V \times E$. Each column represents an edge, and each row represents a vertex. Entry $matrix[i][j] = 1$ if vertex i is an endpoint of edge j .

- Space complexity: $O(V \times E)$, which can be very large.
- Useful in specialized contexts like network flow problems or algebraic graph theory, where you need to reason about the relationship between vertices and specific edges.
- Checking if two specific vertices share an edge is slow, since you'd need to scan columns looking for a column where both vertices have a 1.

In practice, edge lists and incidence matrices are less commonly used than adjacency lists and matrices, but they show up in specific algorithmic contexts.

Selecting graph representation methods

Picking the right representation comes down to three factors: graph density, which operations you'll perform most, and memory budget.

Graph density: - Dense graphs (edge count close to V^2) pair well with adjacency matrices. A complete graph on 1,000 vertices has about 500,000 edges, and the matrix uses $1,000 \times 1,000 = 1,000,000$ entries, which isn't much overhead. - Sparse graphs (edge count close to V) pair well with adjacency lists or edge lists. A tree with 1,000 vertices has only 999 edges, so an adjacency list uses far less memory than a million-entry matrix.

Operation frequency: - If you frequently check whether a specific edge exists, adjacency matrices give you $O(1)$ lookups. - If you frequently iterate over neighbors (as in BFS/DFS), adjacency lists are faster because you skip non-neighbors entirely. - If you need to process all edges in bulk (sorting them, filtering them), an edge list keeps things simple.

Space constraints: - Adjacency lists at $O(V + E)$ and edge lists at $O(E)$ are the most memory-efficient options for sparse graphs. - Adjacency matrices always cost $O(V^2)$ regardless of edge count.

Problem-specific needs: - Algorithms like Kruskal's that sort or iterate over all edges work naturally with edge lists. - Dynamic graphs with frequent insertions and deletions favor adjacency lists. - Problems requiring fast pairwise adjacency checks (like verifying whether a graph is complete) favor adjacency

matrices.

10.3 Graph ADT and Basic Operations

Graphs are powerful data structures that model relationships between entities. They consist of vertices and edges, representing connections in various real-world scenarios like social networks, transportation systems, and computer networks.

Graph ADTs provide a blueprint for implementing graphs, offering methods for vertex and edge manipulation, traversal, and analysis. Understanding graph operations and their complexities is crucial for efficient problem-solving in computer science and beyond.

Graph ADT and Basic Operations

Graph ADT implementation

- Graph ADT represents a collection of vertices (nodes) and edges (connections between vertices)
- Can be directed (edges have a specific direction) or undirected (edges have no specific direction)
- Directed graphs (digraphs) have edges with a specific direction (one-way streets)
- Undirected graphs have edges with no specific direction (two-way streets)
- Can be weighted (edges have associated weights) or unweighted (edges have no associated weights)
- Weighted graphs have edges with associated numerical values (road networks with distances)
- Unweighted graphs have edges without any associated values (social networks)
- Object-oriented implementation involves defining Vertex, Edge, and Graph classes
- Vertex class represents each vertex in the graph
- Contains data associated with the vertex (label, value, etc.)
- May include methods for adding/removing edges connected to the vertex
- Edge class represents each edge in the graph
- Contains references to the vertices it connects (endpoints)
- May include a weight value for weighted graphs (cost, distance, etc.)
- Graph class manages the collection of vertices and edges
- Contains methods for adding/removing vertices and edges
- Provides methods for graph traversal (DFS, BFS) and other graph operations (shortest path, connectivity)

Vertex and edge manipulation

- Adding a vertex creates a new instance of the Vertex class and adds it to the graph's collection of vertices
- Time complexity: $O(1)$ as it involves a simple insertion operation
- Removing a vertex removes it from the graph's collection of vertices and all edges connected to it

- Time complexity: $O(E)$, where E is the number of edges in the graph, as it requires updating all connected edges
- Adding an edge creates a new instance of the Edge class, adds it to the graph's collection of edges, and updates the adjacency lists or matrices of the connected vertices
- Time complexity: $O(1)$ as it involves a simple insertion operation
- Removing an edge removes it from the graph's collection of edges and updates the adjacency lists or matrices of the connected vertices
- Time complexity: $O(1)$ as it involves a simple deletion operation

Graph traversal techniques

- Depth-first search (DFS) explores as far as possible along each branch before backtracking
- Implemented using a stack data structure (recursion or explicit stack)
- Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges
- Space complexity: $O(V)$ for the stack to keep track of visited vertices
- Applications: finding connected components, topological sorting, solving mazes
- Breadth-first search (BFS) explores all neighboring vertices before moving to the next level
- Implemented using a queue data structure
- Time complexity: $O(V + E)$, where V is the number of vertices and E is the number of edges
- Space complexity: $O(V)$ for the queue to keep track of visited vertices
- Applications: shortest path algorithms (Dijkstra's algorithm), web crawlers

Complexity of graph operations

- Adjacency list representation
- Space complexity: $O(V + E)$ as it stores each vertex and edge separately
- Adding/removing a vertex: $O(1)$ as it involves a simple insertion/deletion operation
- Adding/removing an edge: $O(1)$ as it involves updating the adjacency lists of the connected vertices
- DFS and BFS: $O(V + E)$ time complexity as it visits each vertex and edge once
- Adjacency matrix representation
- Space complexity: $O(V^2)$ as it stores a matrix of size $V \times V$
- Adding/removing a vertex: $O(V^2)$ due to matrix resizing and updating all entries
- Adding/removing an edge: $O(1)$ as it involves updating a single entry in the matrix
- DFS and BFS: $O(V^2)$ time complexity as it requires iterating through the entire matrix
- Factors affecting the choice of representation
- Density of the graph (ratio of edges to vertices)
- Adjacency list is more efficient for sparse graphs (few edges relative to vertices)

- Adjacency matrix is more efficient for dense graphs (many edges relative to vertices)
- Frequency of graph operations (additions, removals, traversals)
- Adjacency list is more efficient for frequent vertex/edge additions and removals
- Adjacency matrix is more efficient for frequent edge existence checks and updates
- Space constraints of the system
- Adjacency list requires less space for sparse graphs
- Adjacency matrix requires more space but provides faster access for dense graphs

Unit 11 – Graph Traversal Algorithms (BFS and DFS)

11.1 Breadth-First Search (BFS) Algorithm

Breadth-First Search (BFS) Algorithm

Breadth-First Search (BFS) is a graph traversal algorithm that explores a graph level by level, visiting all neighbors of a vertex before moving deeper. It relies on a queue to process vertices in first-in, first-out (FIFO) order, which naturally produces this layer-by-layer behavior.

BFS is especially useful for finding the shortest path in unweighted graphs, identifying connected components, and checking whether a graph is bipartite. It runs in $O(V + E)$ time and $O(V)$ space, making it efficient for most graph problems you'll encounter.

Characteristics of BFS

- Level-by-level exploration: BFS visits every vertex at distance k from the start before visiting any vertex at distance $k + 1$. This is what makes it "breadth-first" rather than "depth-first."
- Shortest path guarantee: In an unweighted graph, the first time BFS reaches a vertex is always via the shortest path (fewest edges). This property does not hold for weighted graphs.
- Queue-driven order: A queue enforces FIFO processing. The vertex that was discovered earliest gets explored next, which is exactly what produces the level-order behavior.
- Visited tracking: BFS marks vertices as visited when they are enqueued, not when they are dequeued. This prevents the same vertex from being added to the queue multiple times.

Queue-Based BFS Implementation

Here's how BFS works step by step:

1. Pick a starting vertex. Enqueue it and mark it as visited.
2. While the queue is not empty: - Dequeue the front vertex (call it current). - Process current (print it, record its distance, etc.). - For each neighbor of current that hasn't been visited, enqueue that neighbor and mark it as visited.
3. When the queue empties, every reachable vertex has been explored.

A common mistake is marking a vertex as visited only when you dequeue it. If you do that, the same vertex can get enqueued multiple times by different neighbors, wasting time and potentially producing incorrect results. Always mark visited on enqueue.

Pseudocode:

```
function BFS(graph, startVertex):  
    queue = new Queue()  
    visited = new Set()  
  
    queue.enqueue(startVertex)  
    visited.add(startVertex)
```

```

while queue is not empty:
    currentVertex = queue.dequeue()
    process(currentVertex)

    for each neighbor of currentVertex:
        if neighbor is not in visited:
            queue.enqueue(neighbor)
            visited.add(neighbor)

```

Quick trace example: Consider a graph where vertex A connects to B and C, B connects to D, and C connects to D.

- Start: enqueue A → queue: [A], visited: {A}
- Dequeue A, enqueue B and C → queue: [B, C], visited: {A, B, C}
- Dequeue B, enqueue D (C is already visited) → queue: [C, D], visited: {A, B, C, D}
- Dequeue C, D is already visited → queue: [D]
- Dequeue D, no unvisited neighbors → queue: []
- Done. Traversal order: A, B, C, D

Complexity Analysis

Time complexity: $O(V + E)$

Every vertex is enqueued and dequeued exactly once ($O(V)$ work). For each vertex, BFS iterates over its adjacency list to check neighbors. Across all vertices, this totals $O(E)$ for a directed graph or $O(2E)$ for an undirected graph (each edge appears in two adjacency lists), which simplifies to $O(E)$. Combined: $O(V + E)$.

For a connected graph, $E \geq V - 1$, so the time complexity simplifies to $O(E)$.

Space complexity: $O(V)$

- The visited set stores up to V entries.
- The queue can hold up to $O(V)$ vertices in the worst case. For example, in a star graph (one center vertex connected to all others), all $V - 1$ neighbors enter the queue at once.

Applications of BFS

Finding the shortest path in an unweighted graph

Because BFS discovers vertices level by level, the first time it reaches any vertex is via the minimum number of edges. To actually recover shortest distances:

1. Initialize a distance array with all values set to ∞ (or -1). Set $\text{distance}[\text{start}] = 0$.
2. When you enqueue a neighbor, set $\text{distance}[\text{neighbor}] = \text{distance}[\text{current}] + 1$.
3. After BFS completes, the distance array holds the shortest path length from the start to every reachable vertex.

To reconstruct the actual path, also store a parent/predecessor for each vertex and trace back from the destination to the start.

Identifying connected components in an undirected graph

1. Loop through all vertices. When you find one that hasn't been visited, start a BFS from it.
2. Every vertex reached during that BFS belongs to the same connected component.
3. Repeat until all vertices have been visited. The number of times you started a new BFS equals the number of connected components.

Checking if a graph is bipartite

A graph is bipartite if you can color every vertex with one of two colors such that no two adjacent vertices share the same color. BFS handles this naturally:

1. Assign the start vertex color 0. Enqueue it.
2. For each neighbor of the current vertex, if unvisited, assign it the opposite color and enqueue it. If it's already visited and has the same color as the current vertex, the graph is not bipartite.
3. If BFS finishes without finding a conflict, the graph is bipartite.

This works because BFS alternates between levels, and in a bipartite graph, adjacent vertices always belong to different levels (different color groups).

11.2 Depth-First Search (DFS) Algorithm

Depth-First Search (DFS) Algorithm

Depth-First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. Where BFS fans out level by level, DFS dives deep down one path, only turning around when it hits a dead end. This makes it naturally suited for problems like cycle detection, topological sorting, and pathfinding through maze-like structures.

DFS can be implemented recursively or iteratively using a stack. Either way, the time complexity is $O(V + E)$, where V is the number of vertices and E is the number of edges.

DFS Algorithm Fundamentals

The core idea of DFS: pick a starting vertex, visit it, then immediately move to one of its unvisited neighbors. Keep going deeper until you reach a vertex with no unvisited neighbors, then backtrack to the most recent vertex that still has unexplored paths.

- Works on both directed graphs (edges have a direction) and undirected graphs (edges go both ways)
- Uses a stack data structure to track which vertex to visit next. The recursive version uses the call stack implicitly; the iterative version uses an explicit stack.
- Requires a visited set (or boolean array) to avoid revisiting vertices and getting stuck in infinite loops

Think of it like navigating a maze: you pick a direction and keep walking until you hit a wall, then retrace your steps to the last intersection where you had an untried path.

Stack-Based and Recursive DFS Implementation

Recursive implementation:

1. Mark the current vertex as visited.
2. For each unvisited neighbor of the current vertex, recursively call DFS on that neighbor.
3. The base case is implicit: when a vertex has no unvisited neighbors, the function returns and the call stack handles backtracking for you.

```
DFS(vertex, visited):
    visited.add(vertex)
    for each neighbor of vertex:
        if neighbor not in visited:
            DFS(neighbor, visited)
```

Iterative implementation using an explicit stack:

1. Create a stack and push the starting vertex onto it.
2. While the stack is not empty:
3. Pop a vertex from the top of the stack.
4. If that vertex has not been visited, mark it as visited.
5. Push all of its unvisited neighbors onto the stack.

One thing to watch: the iterative version may visit neighbors in a different order than the recursive version, since pushing all neighbors onto the stack reverses the order they get processed. If traversal order matters, push neighbors in reverse order.

Time and Space Complexity of DFS

Time complexity: $O(V + E)$

Every vertex is visited at most once, and every edge is examined at most once (twice for undirected graphs, but that's still proportional to E). This is the same time complexity as BFS.

Space complexity: $O(V)$

- The visited structure takes $O(V)$ space.
- In the worst case, the stack (explicit or call stack) can hold up to V vertices. This happens when the graph is a long chain with no branching, so DFS goes all the way deep before any backtracking.
- For the recursive version, a very deep graph can cause a stack overflow if V is large. The iterative version avoids this by using heap-allocated memory for the explicit stack.

Applications of DFS in Graph Problems

Cycle detection is one of the most common uses. During DFS on an undirected graph, if you encounter an edge leading to a vertex that's already been visited (and it's not the vertex you just came from), a cycle exists. For directed graphs, cycle detection uses a "recursion stack" or coloring scheme to track vertices currently being explored.

Topological sorting applies to directed acyclic graphs (DAGs). You run DFS and record vertices in reverse order of completion. The result is a valid ordering where every edge points from an earlier vertex to a later one. Course prerequisite chains are a classic example: if Course A must come before Course B, A appears earlier in the topological order.

Finding connected components: In an undirected graph, running DFS from an unvisited vertex discovers all vertices in that component. Repeat from the next unvisited vertex to find the next component. For directed graphs, algorithms like Kosaraju's or Tarjan's use DFS to find strongly connected components, which are maximal subsets of vertices where every vertex is reachable from every other vertex.

Pathfinding and exhaustive search: DFS naturally explores all possible paths from a source, making it useful for problems that require examining every configuration. Maze solving, puzzle states (like solving a Sudoku board), and generating all permutations are typical examples. Note that DFS does not guarantee the shortest path, so use BFS when shortest paths matter.

11.3 Applications of BFS and DFS

Graph Traversal Applications

BFS and DFS aren't just abstract algorithms you learn for exams. They're the backbone of real systems: search engines, GPS apps, social networks, and operating systems all rely on graph traversal to solve practical problems. Knowing which traversal to use and why is what separates understanding the algorithms from actually applying them.

Real-World Applications of BFS and DFS

Social network analysis is one of the most intuitive applications. A social network is naturally a graph: users are nodes, and friendships or follows are edges.

- BFS finds the shortest connection path between two users. LinkedIn's "2nd degree connection" feature works this way: start from your node, traverse level by level, and see how many hops it takes to reach someone.
- DFS can identify clusters or communities by exploring deeply through connected groups of users who interact frequently.
- Friend recommendation engines use graph proximity: if you and another user share many neighbors, the system suggests a connection.

Web crawling and indexing treats the internet as a directed graph where pages are nodes and hyperlinks are edges.

- A crawler starts at a seed URL and follows links to discover new pages. BFS-style crawling prioritizes pages close to the seed (good for indexing a single site's structure), while DFS-style crawling dives deep along link chains.
- Crawlers also detect broken links (edges leading to nonexistent nodes) and analyze site structure for SEO.

Pathfinding in navigation and games relies heavily on BFS for shortest-path problems.

- In an unweighted grid (like a simple maze), BFS guarantees the shortest path from start to goal. Game AI for grid-based movement often uses exactly this.
- DFS is useful when you need to explore all possible paths, such as generating every route through a maze or enumerating solutions in a puzzle.

Resource allocation and scheduling maps tasks and dependencies as a directed graph.

- Topological sorting (a DFS application) determines a valid execution order for tasks with dependencies, which is how build systems and project schedulers work.

- Operating systems use graph traversal on resource-allocation graphs to detect deadlocks: if DFS finds a cycle, a deadlock exists.

Graph Connectivity Problems

Many important problems boil down to asking "what's connected to what?" in a graph.

- Finding connected components: Run BFS or DFS from an unvisited node, mark everything reachable, then repeat for the next unvisited node. Each traversal discovers one connected component. This detects network partitions in distributed systems or groups related data points in clustering tasks.
- Reachability: To check whether node A can reach node B, run BFS or DFS from A. If B gets visited, a path exists. This is fundamental for verifying network reliability.
- Bridges and articulation points: These are edges or nodes whose removal disconnects the graph. A modified DFS (using discovery and low-link values) identifies them efficiently. Network engineers use this to find single points of failure.
- Flood fill: The paint bucket tool in image editors is literally BFS/DFS. Starting from a pixel, the algorithm visits all connected pixels of the same color and recolors them. Same idea applies to region labeling in image segmentation.

BFS and DFS in Diverse Domains

Recommendation systems model users and items as a bipartite graph. Traversing from a user node through item nodes to other users who liked the same items is the basis of collaborative filtering. The closer two users are in the graph, the more relevant their preferences are to each other.

Network analysis and optimization uses traversal to find structural properties:

- Influential nodes (hubs with high connectivity) can be identified by examining traversal patterns and node degrees.
- Bottleneck detection finds nodes or edges whose removal would most disrupt flow, which matters for both computer networks and transportation systems.

Web search and ranking builds on graph structure. Google's PageRank algorithm treats the web as a directed graph and assigns importance scores based on link structure. While PageRank itself uses iterative matrix methods rather than plain BFS/DFS, the underlying web graph is first discovered through crawling, which is a traversal problem.

BFS vs. DFS: Use Cases and Trade-Offs

Understanding when to pick one over the other comes down to what you're optimizing for.

	BFS	DFS
Traversal order	Level by level (all neighbors first)	As deep as possible before backtracking
Data structure	Queue	Stack (or recursion)
Shortest path	Guarantees shortest path in unweighted graphs	Does not guarantee shortest path
Memory usage	Stores all nodes at the current frontier level, which can be large for wide graphs	Stores only nodes along the current path, so memory usage is proportional to the maximum depth

	BFS	DFS
Time complexity	$O(V + E)$	$O(V + E)$
Best for	Shortest paths, nearest neighbors, level-order processing	Cycle detection, topological sorting, exploring all paths, connected components

A few things to note about trade-offs:

- Both have the same time complexity of $O(V + E)$, so the choice isn't about speed. It's about what kind of answer you need and how much memory you can afford.
- BFS memory can blow up on wide graphs. If a node has thousands of neighbors, the queue grows fast. DFS only needs stack space proportional to the longest path, which is often much smaller.
- DFS pairs naturally with recursion, making it simpler to implement for problems like topological sort or finding strongly connected components. BFS requires an explicit queue.
- A search, often mentioned alongside BFS, is actually a weighted* shortest-path algorithm that uses a priority queue and a heuristic. It's related to BFS in spirit but is a distinct algorithm.

Quick decision rule: If you need the shortest or closest answer, reach for BFS. If you need to explore exhaustively or detect structure like cycles and components, reach for DFS.

Unit 12 – Minimum Spanning Trees & Shortest Paths

12.1 Minimum Spanning Tree Algorithms (Prim's and Kruskal's)

Minimum Spanning Tree Algorithms

A Minimum Spanning Tree (MST) is the cheapest way to connect every node in a weighted, undirected graph without forming any cycles. MST algorithms show up constantly in network design problems: laying cable between buildings, connecting routers, or planning road systems where you want full connectivity at minimum cost.

This section covers the two classic MST algorithms, Prim's and Kruskal's, along with the properties that make MSTs work.

Minimum Spanning Tree Algorithms

Minimum spanning tree properties

An MST is a subset of edges from a weighted, connected, undirected graph that connects all vertices with the lowest possible total edge weight. Because it's a tree, it contains no cycles, and it always has exactly $V - 1$ edges for V vertices.

Whether a graph has one unique MST or several valid ones depends on the edge weights:

- Distinct edge weights guarantee a unique MST.
- Duplicate edge weights can produce multiple valid MSTs (each with the same total weight, but using different edges).

Two properties are especially useful for proving correctness of MST algorithms:

- Cut property: Take any cut (a partition of vertices into two groups). The minimum-weight edge crossing that cut must be in the MST. This is why greedy choices work: picking the lightest crossing edge is always safe.
- Cycle property: For any cycle in the graph, the heaviest edge in that cycle is not in the MST. If it were included, you could swap it out for a cheaper edge and reduce total weight.

Steps of Prim's algorithm

Prim's grows the MST outward from a single starting vertex, always attaching the nearest reachable vertex. Think of it as expanding a connected region one vertex at a time.

1. Initialize an empty set to hold MST vertices. Assign a key value to every vertex: set the starting vertex's key to 0 and all others to infinity.
2. While there are vertices not yet in the MST:
3. Select the vertex with the smallest key value that isn't in the MST yet.
4. Add that vertex to the MST set.

5. For each neighbor of the newly added vertex, check: is the edge weight to that neighbor less than the neighbor's current key? If so, update the neighbor's key to the smaller weight (and record which edge produced it).
6. Stop when all vertices are in the MST. The recorded edges form the tree.

The key values act as "best known cost to reach this vertex from the MST so far." A priority queue (typically a binary heap) makes step 2.i efficient by quickly extracting the minimum-key vertex.

Steps of Kruskal's algorithm

Kruskal's takes a different approach: instead of growing from one vertex, it considers all edges globally, cheapest first, and merges components together.

1. Sort all edges in non-decreasing order of weight.
2. Initialize an empty edge set for the MST. Create a disjoint-set (union-find) data structure where each vertex starts in its own component.
3. Iterate through the sorted edges, from lightest to heaviest:
4. For the current edge (u, v) , use Find to check whether u and v are in the same component.
5. If they're in different components, add the edge to the MST and call Union to merge their components.
6. If they're already in the same component, skip the edge (adding it would create a cycle).
7. Stop once the MST contains $V - 1$ edges.

Cycle detection here is handled entirely by the union-find structure. Two vertices in the same set are already connected, so any edge between them would close a cycle.

Prim's vs Kruskal's algorithms

Both algorithms are greedy: they make locally optimal choices that lead to a globally optimal MST. When all edge weights are distinct, they produce the exact same tree.

Time complexity

Algorithm	Complexity	Breakdown
Prim's (binary heap)	$O(E \log V)$	Each edge relaxation and vertex extraction costs $O(\log V)$
Kruskal's	$O(E \log E)$	Dominated by sorting; union-find operations run in $O(\alpha(V))$ per call, which is nearly constant

Since $E \leq V^2$, $\log E$ is at most $2 \log V$, so both algorithms have the same asymptotic order.

When to use which

- Prim's works well on dense graphs (many edges) because it processes vertices via a priority queue and pairs naturally with an adjacency matrix. It doesn't need to sort all edges upfront.
- Kruskal's tends to be faster in practice on sparse graphs (few edges) because sorting a small edge list is cheap, and the union-find operations are nearly constant time.

If you're choosing between them on an exam, a good rule of thumb: dense graph $\rightarrow$ Prim's, sparse graph $\rightarrow$ Kruskal's. If edge weights are already sorted, Kruskal's gets even cheaper since you skip the $O(E \log E)$ sorting step.

12.2 Shortest Path Algorithms (Dijkstra's and Bellman-Ford)

Shortest Path Algorithms

Shortest path algorithms solve a fundamental graph problem: given a weighted graph, what's the cheapest way to get from one vertex to another? "Cheapest" here means the path whose edge weights sum to the smallest total.

Two algorithms dominate this space. Dijkstra's algorithm is faster but only works when all edge weights are non-negative. Bellman-Ford is slower but handles negative edge weights and can detect negative cycles. Knowing when to use each one is just as important as knowing how they work.

The Shortest Path Problem

In a weighted graph, each edge has a numeric weight representing some cost: distance in miles, latency in milliseconds, price in dollars, etc. The shortest path between two vertices is the path where the sum of these weights is minimized. This isn't necessarily the path with the fewest edges.

Common applications:

- Navigation systems (Google Maps, Waze): finding the fastest or shortest driving route between two locations
- Network routing: protocols like OSPF use shortest path computations to route packets efficiently across the internet
- Supply chain optimization: minimizing transportation cost or delivery time across a logistics network
- Social networks: measuring degrees of separation between users or finding connection paths for recommendations

Process of Dijkstra's Algorithm

Dijkstra's works by greedily expanding outward from the source, always processing the closest unvisited vertex next. It maintains a running estimate of the shortest distance to every vertex and progressively locks in final answers.

1. Initialize distances. Set the source vertex's distance to 0. Set every other vertex's distance to ∞ .
2. Insert the source into a priority queue (min-heap), keyed by distance.
3. Extract the minimum. Pull the vertex with the smallest distance from the priority queue. Call it u .
4. Relax each neighbor. For every neighbor v of u , compute the tentative distance: $\text{dist}(u) + w(u, v)$. If this value is less than v 's current distance, update v 's distance and record u as v 's predecessor. Then insert or update v in the priority queue.
5. Repeat steps 3–4 until the priority queue is empty.

Once the queue is empty, you have the shortest path distances from the source to all reachable vertices. To reconstruct the actual path, trace back through the predecessor pointers.

Why does the greedy approach work here? When all weights are non-negative, the vertex with the smallest tentative distance can never be improved later. That's why Dijkstra's processes it and never revisits it. If negative weights exist, this guarantee breaks, and the algorithm can produce wrong answers.

Process of Bellman-Ford Algorithm

Bellman-Ford takes a different approach: instead of greedily picking the next vertex, it repeatedly relaxes every edge in the graph. It's slower, but this brute-force relaxation handles negative edge weights correctly.

1. Initialize distances. Set the source vertex's distance to 0 and all others to ∞ .
2. Relax all edges $|V| - 1$ times. Loop $|V| - 1$ iterations. In each iteration, go through every edge (u, v) in the graph. If $\text{dist}(u) + w(u, v) < \text{dist}(v)$, update $\text{dist}(v)$ and set u as v 's predecessor.
3. Check for negative cycles. Do one more pass over all edges. If any distance can still be reduced, a negative cycle exists, meaning you can keep looping around it to decrease the total weight forever. In that case, shortest paths are undefined.
4. If no distance was updated in step 3, the algorithm has found the correct shortest paths from the source to all reachable vertices.

Why $|V| - 1$ iterations? The longest possible shortest path in a graph with $|V|$ vertices uses at most $|V| - 1$ edges. Each iteration guarantees that paths using one more edge are correctly computed. So after $|V| - 1$ rounds, all shortest paths are finalized.

Complexities of Shortest Path Algorithms

	Dijkstra's	Bellman-Ford
Time complexity	$O((V + E)\log V)$ with a min-heap	$O(V \cdot E)$
Negative edge weights	Cannot handle them	Handles them correctly
Negative cycle detection	No	Yes
Best used when	All edge weights are non-negative	Negative weights are possible, or you need cycle detection

Dijkstra's gets its time complexity from extracting each vertex from the heap ($O(|V|\log|V|)$) and updating neighbors through edge relaxations ($O(|E|\log|V|)$). With a simple array instead of a heap, the complexity becomes $O(|V|^2)$, which can actually be better for dense graphs.

Bellman-Ford performs $|V| - 1$ passes over all $|E|$ edges, giving $O(|V| \cdot |E|)$. This is noticeably slower than Dijkstra's on large graphs with non-negative weights, so you should default to Dijkstra's when negative weights aren't a concern.

One limitation to keep in mind: Bellman-Ford can only detect negative cycles that are reachable from the source vertex. If a negative cycle exists in a disconnected part of the graph, the standard algorithm won't flag it.

12.3 Applications of MST and Shortest Path Algorithms

Applications of MST and Shortest Path Algorithms

Minimum Spanning Trees (MSTs) and Shortest Path Algorithms solve two fundamental network problems: connecting everything at minimum total cost, and finding the cheapest route between specific points. These two goals sound similar but lead to very different solutions. An MST connects all nodes with the least total edge weight, while a shortest path algorithm finds the best route between a source and destination (or all pairs). Understanding where each one applies is critical for choosing the right tool.

Applications of Minimum Spanning Tree Algorithms

Real-world Applications of MSTs

The core idea behind every MST application is the same: you need to connect a set of points, and you want to do it as cheaply as possible without creating unnecessary connections.

Network design and infrastructure is the most natural fit for MSTs. You're physically wiring or piping things together, and every extra meter of material costs money.

- **Telecommunications:** A company laying fiber-optic cable between cities wants to connect all cities with the minimum total cable length. The MST of the city graph (where edge weights are distances or costs) gives exactly that layout.
- **Electrical grids:** Power companies use MST-style optimization to connect power stations to consumers while minimizing the total length of transmission lines.
- **Water supply networks:** Designing pipeline layouts (aqueducts, municipal water systems) to reach all consumers with minimum construction cost follows the same logic.

Clustering and data analysis is a less obvious but powerful application. If you build an MST over data points (using distance or dissimilarity as edge weights), then removing the heaviest edges splits the data into clusters.

- **Image segmentation:** Treat each pixel as a node and connect neighboring pixels with edges weighted by color/intensity difference. Building an MST and cutting its heaviest edges partitions the image into visually distinct regions.
- **Hierarchical clustering:** In machine learning, you can build a dendrogram (tree-like clustering structure) by constructing an MST over data points and progressively removing edges from heaviest to lightest. This produces clusters at multiple levels of granularity.

Approximation algorithms for NP-hard problems use MSTs as a starting point for problems that are too hard to solve exactly.

- **Traveling Salesman Problem (TSP):** Finding the shortest tour visiting all cities exactly once is NP-hard. But you can get a tour guaranteed to be at most 2x the optimal cost by building an MST, then doing a preorder traversal of the tree. This works because the MST cost is always a lower bound on the optimal TSP tour.

Problem-Solving with Network Algorithms

- **Transportation networks:** Road or rail systems connecting cities can use MSTs to minimize total construction cost. This differs from shortest path problems because the goal is building the entire network, not routing through it.

- Supply chain design: MSTs help determine how to connect warehouses to retail locations with minimum total shipping infrastructure cost. They can also guide decisions about where to place distribution centers.
- Network reliability: A pure MST has no redundancy; removing any single edge disconnects the graph. In practice, engineers add backup edges beyond the MST to provide fault tolerance. MSTs help identify which additional edges provide the most benefit, since you can analyze which edge removals would be most damaging.

Efficiency of MST Algorithms

Different MST algorithms suit different graph structures:

- Kruskal's algorithm sorts all edges by weight, then greedily adds them if they don't create a cycle (using a Union-Find structure). Best for sparse graphs where E is much smaller than V^2 .
-

Time complexity: $O(E \log E)$, which is equivalent to $O(E \log V)$ since $E \leq V^2$

•

Prim's algorithm grows the MST from a starting vertex, always adding the cheapest edge that connects a tree vertex to a non-tree vertex. Best for dense graphs where E is close to V^2 .

•

Time complexity: $O(E \log V)$ with a binary heap, or $O(E + V \log V)$ with a Fibonacci heap

•

Borůvka's algorithm works in rounds: each connected component finds its cheapest outgoing edge, and all those edges are added simultaneously. This parallel-friendly structure makes it well suited for distributed or parallel implementations.

- Time complexity: $O(E \log V)$

All three algorithms produce the same MST (assuming unique edge weights). The choice between them comes down to graph density and whether you need parallelism.

Applications of Shortest Path Algorithms

Shortest Path Algorithms in Practice

Where MSTs answer "how do I connect everything cheaply?", shortest path algorithms answer "what's the best route from A to B?" This makes them relevant whenever you're routing through an existing network rather than building one.

Transportation and logistics are the most visible applications:

- GPS navigation: Services like Google Maps run shortest path algorithms (often A* variants) on road network graphs where edge weights represent travel time or distance. Real-time traffic data adjusts the weights dynamically.
- Delivery route optimization: Companies like UPS use shortest path computations as building blocks for planning delivery sequences that minimize fuel consumption and total travel time.

Network routing determines how data moves across the internet:

- OSPF (Open Shortest Path First): Each router builds a graph of the network and runs Dijkstra's algorithm to compute the shortest path to every other router. Routes update when the network topology changes.
- BGP (Border Gateway Protocol): Operates at a higher level between autonomous systems, using path-vector routing rather than pure shortest path, but shortest path concepts underpin the design.
- Quality of Service (QoS) routing: For applications like VoIP or video streaming, edge weights might represent latency or packet loss rather than distance, and the algorithm finds paths meeting specific performance thresholds.

Robotics and path planning apply shortest path algorithms in physical space:

- Autonomous vehicles plan routes in real-time, treating road segments as edges with weights that account for distance, speed limits, and obstacles.
- Industrial robots on assembly lines use shortest path planning to move between workstations efficiently while avoiding collisions.

Social networks and recommendation systems model relationships as graphs:

- Friend suggestions: Platforms like Facebook find short paths between users through mutual connections. A short path between two unconnected users suggests they might know each other.
- Product recommendations: Representing users and products as nodes in a bipartite graph, shortest path distances can capture similarity and drive recommendations.

Problem-Solving with Network Algorithms

- Traffic flow optimization: Shortest path algorithms help model how drivers choose routes. City planners use this to identify bottlenecks, optimize traffic signal timing, and predict how road closures will redirect traffic.
- Resource allocation in distributed systems: In cloud computing, tasks need to be assigned to servers. Shortest path algorithms help minimize communication overhead between dependent tasks by finding efficient mappings across the network.
- Multi-robot coordination: In warehouse automation (think Amazon fulfillment centers), each robot computes shortest paths while accounting for other robots' positions to avoid collisions and deadlocks.
- Geographic Information Systems (GIS): Emergency dispatch systems (911, ambulance routing) run shortest path queries on road networks to find the fastest response routes. These queries need to return results in milliseconds.

Efficiency of Path-Finding Methods

Each shortest path algorithm handles different graph properties:

- Dijkstra's algorithm finds shortest paths from a single source to all other vertices, but requires non-negative edge weights. It greedily selects the closest unvisited vertex and relaxes its neighbors.
-

Time complexity: $O((V + E)\log V)$ with a binary heap, or $O(E + V\log V)$ with a Fibonacci heap

•

Bellman-Ford algorithm handles negative edge weights (but not negative cycles). It relaxes all edges $V - 1$ times. If any distance still decreases on a V -th pass, a negative cycle exists.

-

Time complexity: $O(VE)$, which is slower than Dijkstra's but necessary when negative weights are present

-

Floyd-Warshall algorithm computes shortest paths between all pairs of vertices using dynamic programming. For each intermediate vertex k , it checks whether routing through k improves the path between every pair (i, j) .

-

Time complexity: $O(V^3)$, which is practical for small-to-medium graphs or when you need many pairwise distances

-

A* search algorithm finds the shortest path between a specific source and goal, using a heuristic function $h(n)$ that estimates the remaining cost to the goal. It explores fewer nodes than Dijkstra's when the heuristic is good.

- The heuristic must be admissible (never overestimates) to guarantee optimality. If it's also consistent (satisfies the triangle inequality), A* won't revisit nodes.
- Time complexity depends on heuristic quality; with a perfect heuristic it's nearly linear, with a trivial heuristic ($h = 0$) it reduces to Dijkstra's.

Choosing the right algorithm: Use Dijkstra's for single-source with non-negative weights. Use Bellman-Ford if negative weights exist. Use Floyd-Warshall when you need all-pairs distances on a smaller graph. Use A* when you have a single target and a good heuristic estimate.

Unit 13 – Sorting Algorithms and their Analysis

13.1 Comparison-Based Sorting Algorithms

Comparison-based sorting algorithms organize data by repeatedly comparing pairs of elements to determine their order. They range from simple $O(n^2)$ methods that work well on small inputs to efficient $O(n \log n)$ algorithms designed for large datasets. Knowing how each one works, and when to reach for it, is a core skill in data structures.

Comparison-Based Sorting Algorithms

Implementation of basic sorting algorithms

These three algorithms are straightforward to code and reason about, but they all degrade to $O(n^2)$ time on large or poorly ordered inputs.

Bubble Sort

Bubble Sort walks through the array comparing adjacent elements and swapping them if they're out of order. After each full pass, the largest unsorted element "bubbles" to its final position at the end.

- Repeat passes over the array until no swaps occur in a complete pass.
- Time complexity: $O(n^2)$ worst/average case (reverse-ordered or random input), $O(n)$ best case (already sorted, with an early-termination check).
- Space complexity: $O(1)$ since only a temporary swap variable is needed.

Selection Sort

Selection Sort splits the array into a sorted portion (initially empty) and an unsorted portion. On each iteration, it scans the unsorted portion for the minimum element and swaps it into the next position of the sorted portion.

- The sorted region grows by one element per pass; after $n - 1$ passes the array is fully sorted.
- Time complexity: $O(n^2)$ in all cases. Even if the array is already sorted, it still scans for the minimum every time.
- Space complexity: $O(1)$.

Insertion Sort

Insertion Sort treats the first element as a one-element sorted subarray. It then takes the next unsorted element and shifts larger sorted elements to the right until it finds the correct insertion point.

- Time complexity: $O(n^2)$ worst/average case, $O(n)$ best case (already sorted, since each element only needs one comparison).
- Space complexity: $O(1)$.
- Particularly fast on nearly sorted data because few shifts are needed. The number of shifts equals the number of inversions (pairs of elements that are out of order).

Application of efficient sorting techniques

Both Merge Sort and Quick Sort use a divide-and-conquer strategy: break the problem into smaller subproblems, solve them recursively, and combine the results.

Merge Sort

1. Divide: Split the array into two halves.
 2. Recurse: Recursively sort each half until you reach subarrays of size 1 (base case).
 3. Merge: Combine two sorted halves into one sorted array by comparing elements from the front of each half and placing the smaller one next.
- Time complexity: $O(n \log n)$ in all cases. The array is halved $\log n$ times, and each level of recursion does $O(n)$ work during the merge step.
 - Space complexity: $O(n)$ because the merge step requires a temporary array to hold the merged result.
 - Stable: equal elements keep their original relative order.

Quick Sort

1. Choose a pivot: Select an element (common choices: last element, first element, median-of-three, or random).
 2. Partition: Rearrange the array so that all elements less than the pivot come before it and all elements greater come after it. The pivot is now in its final sorted position.
 3. Recurse: Recursively sort the subarray before the pivot and the subarray after the pivot. Base case: subarray of size 0 or 1.
- Time complexity: $O(n \log n)$ best/average case (balanced partitions), $O(n^2)$ worst case (extremely unbalanced partitions, e.g., pivot is always the min or max).
 - Space complexity: $O(\log n)$ for the recursive call stack in the average case. The sort is in-place since it doesn't allocate a separate output array.
 - Unstable: equal elements may be reordered during partitioning.
 - Worst-case behavior is rare in practice when you use randomized pivot selection or the median-of-three heuristic.

Complexity analysis of comparison sorts

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

One important theoretical result: no comparison-based sort can do better than $O(n \log n)$ in the worst case. This lower bound comes from the fact that there are $n!$ possible orderings, and each comparison

eliminates at most half of them, requiring at least $\log_2(n!)$ comparisons, which is $\Theta(n \log n)$.

Suitability of sorting algorithms

Choosing the right sort depends on input size, how sorted the data already is, memory constraints, and whether stability matters.

- Bubble Sort, Selection Sort, Insertion Sort
- Best for small arrays (roughly $n < 50$ –100) where the overhead of recursive algorithms isn't worth it.
- Insertion Sort is the standout of the three: it runs in $O(n)$ on nearly sorted data and has low constant factors. Many real-world libraries use it as a subroutine inside Quick Sort for small partitions (e.g., Java's `Arrays.sort` switches to Insertion Sort below a threshold).
-

Selection Sort's one advantage: it performs at most $n - 1$ swaps, which matters if writes are expensive.

-

Merge Sort

- Guarantees $O(n \log n)$ regardless of input order, so it's a safe default for large datasets.
- Stable, making it the go-to when you need to preserve the relative order of equal elements (e.g., sorting records by one field after already sorting by another).
-

The downside is $O(n)$ extra memory. In memory-constrained environments, this can be a dealbreaker.

-

Quick Sort

- Generally the fastest in practice for large arrays due to good cache locality and small constant factors, despite sharing the same $O(n \log n)$ average-case complexity as Merge Sort.
- In-place (only $O(\log n)$ stack space), so it's preferred when memory is tight.
- Unstable, so avoid it when relative order of equal elements matters.
- Vulnerable to $O(n^2)$ worst case on already-sorted or nearly-sorted input if you use a naive pivot (like always picking the first or last element). Randomized pivot selection or median-of-three effectively eliminates this risk.

13.2 Non-Comparison Sorting Algorithms

Non-Comparison Sorting Algorithms

Non-comparison sorting algorithms take a fundamentally different approach from algorithms like merge sort or quicksort. Instead of deciding order by comparing pairs of elements, they exploit the structure of the data itself (its digits, its value range, its distribution) to place elements directly into their correct positions. This lets them break through the $O(n \log n)$ lower bound that applies to all comparison-based sorts, achieving linear time in favorable conditions.

These algorithms work best on integer arrays or data that can be mapped to integers within a known range. Understanding when and why they outperform comparison sorts is the core takeaway of this

section.

Principles of Non-Comparison Sorting

Counting Sort works by tallying how many times each value appears, then using those counts to compute where each element belongs in the output.

- Assumes input elements are integers in a specific range $[0, k]$
- Builds a histogram: a count array where index i stores how many times value i appears
- Transforms the count array into a prefix sum array, so each entry tells you how many elements are less than or equal to that value
- Uses the prefix sums to place each element directly into its correct output position, with no comparisons needed
- Works well when the range k is small relative to n (e.g., exam scores 0–100, ages 0–120)

Radix Sort sorts numbers digit by digit, starting from the least significant digit and working toward the most significant.

- Each pass sorts the array by one digit position using a stable subroutine (typically counting sort). Stability is critical here: it preserves the ordering established by previous passes, so earlier digit sorts aren't undone.
- After processing all d digit positions, the array is fully sorted.
- Particularly effective for fixed-width data like zip codes, phone numbers, or fixed-length strings.

Bucket Sort distributes elements across a set of "buckets" that each cover a subrange of the input, sorts each bucket individually, then concatenates the results.

- Each bucket represents a subinterval of the input range (e.g., prices $[0, 10)$, $[10, 20)$, etc.)
- Elements are assigned to buckets based on their value, grouping similar values together
- Each bucket is sorted independently, often with insertion sort (which is fast on small or nearly-sorted data)
- The sorted buckets are concatenated in order to produce the final output
- Performs best when input is uniformly distributed over the range (e.g., floating-point numbers between 0 and 1), because that keeps bucket sizes roughly equal

Implementation for Integer Arrays

Counting Sort Implementation

1. Find the maximum value k in the input array.
2. Create a count array of size $k + 1$, initialized to zero.
3. Iterate through the input array and increment $\text{count}[\text{element}]$ for each element.
4. Compute the prefix sum of the count array: each entry now stores the number of elements less than or equal to that index. This tells you the last valid position for that value in the output.
5. Build the sorted output array by traversing the input in reverse order (this preserves stability). For each element, place it at position $\text{count}[\text{element}] - 1$ in the output, then decrement that count.

6. Copy the output array back to the original array.

Traversing in reverse during step 5 is what makes counting sort stable. If you go forward instead, equal elements end up in reversed relative order.

Radix Sort Implementation

1. Find the maximum element and determine how many digits d it has.
2. For each digit position, starting from the least significant: - Run counting sort on the array using only the current digit as the key (extract the digit with division and modulo operations). - Because counting sort is stable, the relative order from previous passes is preserved.
3. After all d passes, the array is sorted.

Bucket Sort Implementation

1. Choose the number of buckets. A common choice is n buckets for n elements, though $\sqrt{n}$ is also used.
2. Determine the range of the input and assign each element to a bucket based on its value (e.g., $\text{bucket_index} = \text{floor}(n * \text{element} / \text{max_value})$).
3. Sort each bucket individually using a simple algorithm like insertion sort.
4. Concatenate all buckets in order to produce the final sorted array.

Complexity of Non-Comparison Algorithms

Counting Sort

- Time: $O(n + k)$, where n is the number of elements and k is the range of input values. You iterate through the input once and through the count array once.
- Space: $O(n + k)$ for the count array (size $k + 1$) and the output array (size n).
- The catch: if k is much larger than n , this becomes wasteful. Sorting 10 elements in the range $[0, 1,000,000]$ means allocating a million-entry count array.

Radix Sort

- Time: $O(d \cdot (n + k))$, where d is the number of digits and k is the radix (base). For decimal digits, $k = 10$; for bytes, $k = 256$.
- Space: $O(n + k)$ for the counting sort subroutine's temporary arrays.
- Since d and k are often treated as constants (e.g., sorting 32-bit integers in base 256 gives $d = 4$, $k = 256$), radix sort effectively runs in $O(n)$ for fixed-width data.

Bucket Sort

- Time: $O(n + k)$ on average, where k is the number of buckets. This average case assumes uniform distribution, which keeps each bucket at roughly $O(1)$ elements. More precisely, sorting bucket i takes $O(n_i \log n_i)$ if using a comparison sort, but with uniform distribution and $k = n$ buckets, each n_i is $O(1)$.
- Worst case: $O(n^2)$ if all elements land in the same bucket and you're using insertion sort inside it.
- Space: $O(n + k)$ for the bucket structures and output.

Non-Comparison vs. Comparison-Based Sorting

Advantages

- Can achieve $O(n)$ time (or close to it), beating the $O(n \log n)$ lower bound that applies to all comparison-based sorts
- Counting sort is extremely fast when the value range k is small relative to n
- Radix sort handles large datasets of fixed-width integers or strings very efficiently
- Counting sort and radix sort are stable, which matters when you need to preserve the relative order of equal elements

Limitations

- You need to know (or compute) the range of input values ahead of time
- Higher space complexity than in-place comparison sorts like heapsort or quicksort, since they require auxiliary arrays or bucket structures
- Not directly applicable to arbitrary data types (floats, objects with custom orderings) without first mapping values to integers
- Performance degrades badly when $k \gg n$: counting sort wastes space, and bucket sort produces unbalanced buckets
- None of these algorithms sort in-place in the traditional sense, which can be a constraint in memory-limited environments

Rule of thumb: Use non-comparison sorts when your data is integer-valued (or can be treated that way), the range is manageable, and you need speed over minimal memory usage. Fall back to comparison-based sorts for general-purpose or memory-constrained scenarios.

13.3 Sorting Algorithm Analysis and Trade-offs

Sorting Algorithm Analysis

Sorting algorithms sit at the core of data structures coursework because the choice of algorithm directly affects program performance. Each algorithm carries its own time complexity, space complexity, and behavioral quirks, so understanding the trade-offs lets you pick the right tool for a given situation.

Time and Space Complexity of Sorting Algorithms

The table below summarizes the complexities you need to know. After the table, you'll find notes on why these numbers look the way they do.

Algorithm	Best	Average	Worst	Space
Bubble Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
Insertion Sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$
Quick Sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$

A few things worth noticing:

- Bubble Sort and Insertion Sort hit $O(n)$ in the best case because they can detect that the array is already sorted and stop early. Selection Sort can't do this; it always scans for the minimum, giving $O(n^2)$ regardless.
- Merge Sort guarantees $O(n \log n)$ every time, but it pays for that consistency with $O(n)$ extra space for the temporary arrays used during merging.
- Quick Sort averages $O(n \log n)$, yet degrades to $O(n^2)$ when pivot selection is poor (e.g., always picking the smallest or largest element in an already-sorted array). Its $O(\log n)$ space comes from the recursion stack.
- Heap Sort matches Merge Sort's $O(n \log n)$ guarantee while using only $O(1)$ extra space, which sounds ideal. The catch is that it tends to be slower in practice due to poor cache locality compared to Quick Sort.

Stability and Adaptability in Sorting

Stability means the algorithm preserves the relative order of elements that compare as equal. This matters when you're sorting by one field but want to keep a previous ordering on another field intact.

- Stable: Bubble Sort, Insertion Sort, Merge Sort
- Unstable: Selection Sort, Quick Sort, Heap Sort

For example, if you sort a list of students by grade and two students both have a 90, a stable sort keeps them in whatever order they were in before the sort. An unstable sort might swap them.

Adaptability means the algorithm runs faster when the input is already partially sorted.

- Adaptive: Bubble Sort, Insertion Sort. Both benefit from existing order because they do fewer swaps and comparisons on nearly-sorted data.
- Non-adaptive: Selection Sort, Merge Sort, Quick Sort, Heap Sort. These perform roughly the same amount of work no matter how the input starts.

Selection Criteria for Sorting Algorithms

Choosing an algorithm depends on the constraints of your specific problem. Here are the main factors:

- Small input sizes: Insertion Sort is typically the best choice. Its low overhead (no recursion, no extra arrays) makes it faster than $O(n \log n)$ algorithms for small n , often around $n < 20-50$.
- Nearly sorted input: Bubble Sort and Insertion Sort both adapt well here, but Insertion Sort is generally preferred because it does fewer total operations.
- Large input sizes: Merge Sort, Quick Sort, and Heap Sort are your go-to options since their $O(n \log n)$ average-case complexity scales far better than $O(n^2)$.
- Memory-constrained environments: Bubble Sort, Selection Sort, Insertion Sort, and Heap Sort all use $O(1)$ extra space. Among these, Heap Sort is the strongest performer on large data.
- Stability required: Merge Sort is the standard choice when you need both $O(n \log n)$ performance and stability.

Performance Analysis and Trade-offs

Performance Scenarios in Sorting

Understanding best, average, and worst cases helps you anticipate how an algorithm behaves under different conditions.

- Best case: Bubble Sort and Insertion Sort achieve $O(n)$ on already-sorted input. The other algorithms still run at their usual average-case complexity because their structure doesn't let them "skip" work based on existing order.
- Average case: The $O(n \log n)$ algorithms (Merge Sort, Quick Sort, Heap Sort) clearly separate themselves from the $O(n^2)$ algorithms (Bubble Sort, Selection Sort, Insertion Sort). For 10,000 elements, that's roughly 130,000 operations vs. 100,000,000.
- Worst case: Quick Sort is the one to watch here. With a bad pivot every time, it degrades to $O(n^2)$. This is why real-world implementations use strategies like median-of-three pivot selection or randomized pivots to make the worst case extremely unlikely.

Time vs. Space Trade-offs in Sorting

There's a recurring tension in sorting: faster algorithms tend to use more memory, and memory-efficient algorithms tend to be slower.

- Merge Sort illustrates the "pay space for speed" approach. It guarantees $O(n \log n)$ time but needs $O(n)$ auxiliary space for merging. If you're sorting millions of records, that extra memory allocation can be significant.
- Bubble Sort, Selection Sort, and Insertion Sort sit on the opposite end. They use $O(1)$ space but pay for it with $O(n^2)$ time in the average and worst cases.
- Heap Sort is an interesting middle ground: $O(n \log n)$ time with $O(1)$ space. It avoids the extra memory of Merge Sort and the worst-case blowup of Quick Sort, though its constant factors are higher in practice.
- Hybrid algorithms try to get the best of multiple worlds. A good example is Introsort, used by C++'s `std::sort`. It starts with Quick Sort for its fast average-case performance, monitors recursion depth, and switches to Heap Sort if the depth exceeds $O(\log n)$. This guarantees $O(n \log n)$ worst-case time while keeping space usage low. For small sub-arrays, it often switches to Insertion Sort for its low overhead.

Quick decision guide: Need stability? Merge Sort. Need low memory? Heap Sort. Need fast average-case and don't mind a rare worst case? Quick Sort. Small or nearly sorted data? Insertion Sort. Need all-around guarantees? A hybrid like Introsort.

Unit 14 – Searching Algorithms and their Analysis

14.1 Linear and Binary Search Algorithms

Search Algorithms

Search algorithms let you find a specific element within a data structure. Linear search checks items one by one, while binary search repeatedly halves the search space in a sorted array. These two approaches represent a fundamental trade-off in computer science: generality versus efficiency.

Linear search works on any list, sorted or not, but it can be slow for large datasets. Binary search is restricted to sorted collections, but its performance on large inputs is dramatically better. Knowing when to use each one is a core skill.

Linear Search Implementation and Complexity

Linear search walks through a list or array from start to finish, comparing each element to the target. If it finds a match, it returns that element's index. If it reaches the end without a match, it returns a sentinel value (typically -1).

Here's the step-by-step process:

1. Start at index 0.
2. Compare the current element to the target.
3. If they match, return the current index.
4. If they don't match, move to the next index.
5. If you've checked every element without finding a match, return -1.

This works on arrays, linked lists, or any sequential data structure. No sorting required.

Time complexity:

- Best case: $O(1)$ — the target is the very first element.
- Worst case: $O(n)$ — the target is the last element, or it's not in the list at all.
- Average case: $O(n)$ — on average, you'll check about half the elements, but constant factors drop out in Big-O notation.

Space complexity: $O(1)$ — you only need a few variables (an index counter and the target), regardless of input size.

Binary Search in Sorted Arrays

Binary search takes advantage of sorted order to eliminate half the remaining elements with each comparison. It's far faster than linear search on large sorted collections, but it only works when the data is already sorted.

Here's the step-by-step process:

1. Set two pointers: low = 0 and high = n - 1 (the first and last indices).
2. Calculate the middle index: $\text{mid} = \lfloor (\text{low} + \text{high}) / 2 \rfloor$.
3. Compare the target to the element at mid. - If they're equal, return mid. You're done. - If the target is less than the middle element, set high = mid - 1 (search the lower half). - If the target is greater than the middle element, set low = mid + 1 (search the upper half).
4. Repeat steps 2–3 until the target is found or low > high (meaning the search space is empty and the target isn't present).

Time complexity:

- Best case: $O(1)$ — the target happens to be the middle element on the first check.
- Worst case: $O(\log n)$ — you keep halving until the search space has one element (or is empty). For an array of 1,000,000 elements, that's roughly 20 comparisons.
- Average case: $O(\log n)$ — same order of growth as the worst case.

Space complexity: $O(1)$ for the iterative version. Note that a recursive implementation uses $O(\log n)$ space due to call stack frames.

Linear vs. Binary Search Performance

The core trade-off comes down to preconditions and scaling behavior.

•

Linear search has no preconditions. It works on unsorted data, and it's simple to implement. For small arrays or cases where the target is likely near the beginning, it performs well. But its time complexity grows linearly: double the input size, double the (worst-case) search time.

•

Binary search requires the array to be sorted and to support random access (so linked lists are out). But its time complexity grows logarithmically: double the input size, and you only add one extra comparison.

To put this in concrete terms: searching a sorted array of 1,000,000 elements takes up to 1,000,000 comparisons with linear search, but at most about 20 with binary search.

If your data is unsorted and you only need to search once, linear search is your only practical option. If you're searching the same sorted collection many times, binary search pays off enormously.

Search Algorithms for Diverse Data Types

Linear search works with any data type that supports an equality check ($=$). That includes strings, objects, custom types, or anything where you can define what "equal" means. It also adapts easily to different goals: finding the first occurrence, the last occurrence, or all occurrences of a target.

Binary search requires a total order on the data, meaning every pair of elements can be compared with $<$, $>$, or $=$. This is naturally true for integers and floating-point numbers, and it can be defined for strings (lexicographic order) or custom types with a comparator.

Binary search also supports useful variations on sorted arrays:

- Lower bound search — finds the first position where the target could be inserted while keeping the array sorted (useful for finding the first occurrence among duplicates).

- Upper bound search — finds the position after the last occurrence of the target.

These variations are common in standard libraries (e.g., C++'s `lower_bound` and `upper_bound`, or Python's `bisect` module) and show up frequently in problems that go beyond simple "find this element" tasks.

14.2 Tree and Graph Search Algorithms

Tree and Graph Traversal Algorithms

Tree and graph traversal algorithms give you systematic ways to visit every node in a data structure. DFS and BFS take fundamentally different approaches to this problem, and choosing the right one depends on what you're trying to accomplish.

These traversal techniques show up everywhere in computer science: finding shortest paths, detecting cycles, identifying connected components, and powering more advanced graph algorithms like Dijkstra's and Bellman-Ford.

DFS and BFS for Traversal

Depth-First Search (DFS) explores as far as possible along each branch before backtracking. Think of it like navigating a maze by always taking the next available turn and only retracing your steps when you hit a dead end.

- Implemented with a stack (explicit or via recursion)
- For graphs, you must mark nodes as visited to avoid infinite loops in cyclic structures
- DFS has three traversal orders for binary trees:
 - Pre-order: Visit root → traverse left subtree → traverse right subtree. Useful for copying a tree or evaluating expression trees.
 - In-order: Traverse left subtree → visit root → traverse right subtree. On a BST, this produces elements in sorted order.
 - Post-order: Traverse left subtree → traverse right subtree → visit root. Useful for deleting a tree or evaluating postfix expressions.

Breadth-First Search (BFS) explores level by level, visiting all neighbors of a node before moving deeper.

- Implemented with a queue
- For trees, BFS visits every node at depth d before visiting any node at depth $d + 1$
- For graphs, you again mark visited nodes to prevent revisiting. BFS naturally finds the shortest path (by number of edges) from the starting node, which is why it's used in applications like social network friend suggestions and web crawlers.

Complexity of Search Algorithms

DFS and BFS share the same asymptotic complexity, but their space usage behaves differently in practice.

Time complexity:

- DFS: $O(V + E)$ — each vertex and edge is processed exactly once
- BFS: $O(V + E)$ — same reasoning; every vertex is dequeued once, and every edge is examined once

Space complexity (both $O(V)$ in the worst case, but for different reasons):

- DFS: The stack (or recursion call stack) can grow as deep as the number of vertices. Worst case is a skewed tree or a graph shaped like a linked list, where depth equals V .
- BFS: The queue can hold all vertices at a single level. Worst case is a star graph or the bottom level of a complete binary tree, where up to $O(V)$ nodes sit in the queue at once.

In practice, DFS tends to use less memory on wide, shallow graphs, while BFS tends to use less memory on narrow, deep graphs.

Tree Search Algorithms

Search in Tree Structures

A binary search tree (BST) organizes data so that for every node, all values in the left subtree are smaller and all values in the right subtree are larger. This property makes searching efficient.

Searching a BST:

1. Compare the target value to the current node's value.
2. If the target is smaller, move to the left child.
3. If the target is larger, move to the right child.
4. If the target equals the current node's value, the search succeeds.
5. If you reach a null child, the value isn't in the tree.

Time complexity depends entirely on tree shape:

- Balanced BSTs (AVL trees, Red-Black trees) guarantee $O(\log n)$ for search, insertion, and deletion. They enforce a height constraint — for example, AVL trees require the height difference between left and right subtrees of any node to be at most 1.
- Unbalanced BSTs can degenerate into a linked list (imagine inserting 1, 2, 3, 4, 5 in order). In that case, search becomes $O(n)$.

Applications of Search Algorithms

Shortest Path Problems

Dijkstra's algorithm finds the shortest path from a single source to all other vertices in a weighted graph with non-negative edge weights.

1. Initialize the source distance to 0 and all other distances to infinity.
 2. Add all vertices to a priority queue (min-heap) keyed by tentative distance.
 3. Extract the vertex U with the smallest tentative distance.
 4. For each neighbor V of U , check if the path through U is shorter than V 's current tentative distance. If so, update V 's distance (this is called edge relaxation).
 5. Repeat until the priority queue is empty.
- Time complexity: $O((V + E)\log V)$ with a binary heap, or $O(V^2)$ with a simple array

Bellman-Ford algorithm also solves single-source shortest paths but handles negative edge weights.

1. Initialize the source distance to 0 and all others to infinity.
 2. Relax all edges $V - 1$ times (since the longest shortest path can have at most $V - 1$ edges).
 3. Run one more pass over all edges. If any distance decreases, a negative-weight cycle exists (meaning no valid shortest path).
- Time complexity: $O(VE)$, which is slower than Dijkstra's but necessary when negative weights are present

Connected Components

You can identify all connected components in an undirected graph using either DFS or BFS:

1. Pick any unvisited vertex and run DFS or BFS from it.
 2. Every vertex reached during that traversal belongs to the same connected component.
 3. Repeat from another unvisited vertex until all vertices have been visited.
- Time complexity: $O(V + E)$ for the entire process, since each vertex and edge is visited exactly once across all traversals

14.3 String Searching Algorithms

String Searching Algorithms

String searching algorithms find occurrences of a pattern (a short string) within a larger text. They show up everywhere: text editors doing find-and-replace, search engines scanning documents, bioinformatics tools matching DNA sequences. This section covers three approaches: brute-force, KMP, and Boyer-Moore, each with different tradeoffs in preprocessing cost and search speed.

Brute-force String Searching

The brute-force approach is the most straightforward way to search for a pattern in text. You slide the pattern across the text one position at a time and check for a match at each position.

How it works:

1. Align the pattern with position 0 of the text.
2. Compare characters left to right: `pattern[0]` with `text[0]`, `pattern[1]` with `text[1]`, and so on.
3. If all characters match, you've found the pattern. Return the current position.
4. If a mismatch occurs, shift the pattern one position to the right and start comparing again from `pattern[0]`.
5. Repeat until the pattern is found or you've exhausted all valid positions in the text.

For example, searching for "hello" in "hello world" starts by comparing at index 0. All five characters match, so it returns 0.

Complexity:

- Time: $O(mn)$ worst case, where m is the pattern length and n is the text length. Searching for a pattern of length 5 in text of length 100 could require up to 500 comparisons. The worst case happens

with inputs like pattern "AAAB" in text "AAAAAAAAAB," where almost-complete matches force many comparisons before each mismatch.

- Best case: $O(n)$, when the first character of the pattern mismatches at most positions, so you skip ahead quickly.
- Average case: $O(n)$ for typical text with a reasonably large alphabet (like English), since mismatches tend to happen early.
- Space: $O(1)$, just a few index variables.

The brute-force method works fine for short patterns and small texts, but it wastes effort. After a partial match fails, it "forgets" everything it just learned and starts from scratch. KMP and Boyer-Moore fix this problem in different ways.

KMP Algorithm for Substring Searching

The Knuth-Morris-Pratt (KMP) algorithm avoids redundant comparisons by preprocessing the pattern. When a mismatch occurs partway through a match, KMP uses information about the pattern's internal structure to skip ahead intelligently instead of restarting from the next position.

The failure function (prefix function):

The key idea is the failure function (sometimes called the prefix function or partial match table). For each position i in the pattern, it stores the length of the longest proper prefix of the substring `pattern[0..i]` that is also a suffix of that substring.

For the pattern "ABABC":

Position	Substring	Longest prefix = suffix	Failure value
0	A	(none)	0
1	AB	(none)	0
2	ABA	A	1
3	ABAB	AB	2
4	ABABC	(none)	0

So the failure function is [0, 0, 1, 2, 0].

Algorithm steps:

1. Preprocess: Compute the failure function for the pattern.
2. Search: Compare the pattern against the text left to right, maintaining a pointer i into the pattern.
3. On a match: Advance both the text pointer and i . If i reaches the end of the pattern, you've found a match.
4. On a mismatch: Don't reset i to 0. Instead, set i to the failure function value at position $i - 1$. This jumps i back to the length of the longest prefix that could still be part of a valid match, skipping the characters you already know match.
5. Repeat until the text is exhausted.

Why this helps: Suppose you're matching "ABABC" and you've matched "ABAB" before a mismatch. The failure value at position 3 is 2, meaning the last two characters of what you matched ("AB") are also the first two characters of the pattern. So you can continue comparing from `pattern[2]` instead of starting over.

Complexity:

- Time: $O(m + n)$. Building the failure function takes $O(m)$, and the search takes $O(n)$. Each character in the text is examined at most twice (once on match, once on mismatch fallback), so the total is linear.
- Space: $O(m)$ for the failure function array.

Boyer-Moore Algorithm

The Boyer-Moore algorithm takes a different approach: it compares characters right to left within the pattern and uses two preprocessing heuristics to skip large portions of the text. In practice, it's often the fastest string search algorithm for typical inputs.

Two heuristics:

•

Bad character heuristic: When a mismatch occurs, look at the mismatched character in the text. If that character appears elsewhere in the pattern, shift the pattern so that character aligns with its rightmost occurrence in the pattern. If it doesn't appear in the pattern at all, shift the entire pattern past that position. For example, if you're matching against text character "x" and "x" doesn't appear anywhere in the pattern, you can safely skip ahead by the full pattern length.

•

Good suffix heuristic: When a mismatch occurs after matching some suffix of the pattern, check whether that matched suffix appears elsewhere in the pattern. If it does, shift the pattern to align with that other occurrence. If not, shift to align with the longest prefix of the pattern that matches a suffix of the matched portion.

Algorithm steps:

1. Preprocess the pattern to build the bad character table and the good suffix table.
2. Align the pattern with the beginning of the text.
3. Compare characters from right to left (starting at the last character of the pattern).
4. If all characters match, report the match.
5. On a mismatch, compute the shift from both heuristics and take the maximum of the two shifts.
6. Shift the pattern by that amount and repeat from step 3.

Why right-to-left comparison matters: By starting at the end of the pattern, a mismatch on the very first comparison can let you skip ahead by up to m positions, since you immediately learn something about a character that's far into the text.

Complexity:

- Time: $O(mn)$ worst case (pathological inputs like pattern "AAA" in text "AAAAAA"), but the average case is $O(n/m)$. That sublinear average means it can actually examine fewer characters than exist in the text. Searching for a 5-character pattern in 100 characters of text might need only about 20 comparisons on average.
- Space: $O(m + |\Sigma|)$, where $|\Sigma|$ is the alphabet size. The bad character table needs an entry for each possible character (e.g., 256 for ASCII), and the good suffix table needs $O(m)$ space.

Performance Comparison

Factor	Brute-Force	KMP	Boyer-Moore
Preprocessing	None	$O(m)$	$O(m + \Sigma)$
Worst-case time	$O(mn)$	$O(m + n)$	$O(mn)$
Average-case time	$O(n)$	$O(m + n)$	$O(n/m)$
Space	$O(1)$	$O(m)$	$O(m + \Sigma)$
Comparison direction	Left to right	Left to right	Right to left

When to use each:

- Brute-force is fine for short patterns in short texts (searching for a word in a sentence). The simplicity means less overhead, and for small inputs the $O(mn)$ worst case doesn't matter.
- KMP excels when the pattern contains repeated substrings or characters (like "ABABABABAB"). The failure function captures that repetition and prevents backtracking. It also guarantees $O(m + n)$ regardless of input, making it a safe default for large inputs.
- Boyer-Moore tends to be fastest in practice for searching short patterns in large texts with a reasonably sized alphabet. The sublinear average case means it often skips most of the text entirely. It's the algorithm behind many real-world search tools.

Factors that affect your choice:

- Pattern length vs. text length: Longer patterns give Boyer-Moore more room to skip. Very long patterns with internal repetition favor KMP.
- Alphabet size and character distribution: A larger alphabet (like ASCII) helps Boyer-Moore's bad character heuristic, since mismatched characters are less likely to appear in the pattern. A small alphabet (like DNA's {A, C, G, T}) reduces that advantage.
- Repetition in the pattern: Patterns with lots of repeated prefixes/suffixes are where brute-force performs worst and KMP provides the biggest improvement.

Unit 15 – Algorithm Design: Greedy, D&C, Dynamic Prog.

15.1 Greedy Algorithm Design

Greedy Algorithm Design

Greedy algorithms solve optimization problems by making the best-looking choice at each step, hoping those local decisions add up to a globally optimal solution. They're simple to implement and often efficient, but the tricky part is knowing when they actually work and when they'll lead you astray.

Principles of Greedy Algorithms

Two properties must hold for a greedy approach to produce an optimal solution:

Greedy choice property means that picking the locally optimal choice at each stage can lead to a globally optimal solution. You never need to reconsider a choice once it's made. Dijkstra's algorithm demonstrates this: at each step, it commits to the nearest unvisited vertex and never revisits that decision.

Optimal substructure means the optimal solution to the whole problem contains optimal solutions to its subproblems. For shortest paths, this is intuitive: if the shortest path from A to C goes through B, then the A-to-B portion must itself be the shortest path from A to B.

A greedy algorithm builds its solution incrementally, one choice at a time, and never backtracks. This is both its strength (efficiency) and its weakness (it can fail on problems that lack the two properties above). The Traveling Salesman Problem is a classic example where always visiting the nearest unvisited city often produces a suboptimal tour.

Common problems where greedy does work:

- Minimum Spanning Tree (Prim's, Kruskal's)
- Huffman Coding (optimal prefix-free codes)
- Dijkstra's Algorithm (single-source shortest paths with non-negative weights)

Greedy Techniques for Optimization

Designing a greedy algorithm follows a consistent process:

1. Define the objective function. What are you maximizing or minimizing? (e.g., minimize total edge weight, maximize total profit)
2. Identify the greedy choice. At each step, which locally optimal decision is feasible and moves you toward the objective? In Dijkstra's, that's selecting the unvisited vertex with the smallest tentative distance.
3. Prove the greedy choice is safe. This is the step most students skip, and it's the most important. You need to show that committing to the greedy choice at each step doesn't prevent you from reaching the global optimum. The two standard proof techniques are: - Exchange argument: Assume an optimal solution that differs from the greedy choice, then show you can "swap in" the greedy choice without making the solution worse. - Proof by contradiction: Assume the greedy choice leads to a suboptimal solution and derive a contradiction.

4. Build the algorithm. Repeatedly make the greedy choice and update the problem state until a complete solution is formed (e.g., Prim's algorithm adds the cheapest crossing edge until the spanning tree is complete).
5. Verify optimality. If you can't prove optimality, look for a counterexample. Finding one means greedy won't work for that problem, and you'll need a different strategy (often dynamic programming).

Analysis of Greedy Algorithms

Time Complexity

To analyze a greedy algorithm's running time:

1. Count the number of iterations the main loop performs.
2. Determine the cost of each iteration (selecting the greedy choice, updating state).
3. Combine them using Big-O notation.

For example, Dijkstra's algorithm with a binary heap priority queue: - The main loop runs V times (once per vertex). - Each extraction from the priority queue costs $O(\log V)$. - Each edge relaxation may trigger a decrease-key operation, also $O(\log V)$. - Overall: $O((V + E)\log V)$.

With a Fibonacci heap, the decrease-key drops to amortized $O(1)$, giving $O(V\log V + E)$.

Optimality Analysis

If the greedy approach is correct, prove it using an exchange argument or contradiction (as described above). If it's not correct, a single counterexample is enough to show failure. For the Traveling Salesman Problem, you can construct small graphs where the nearest-neighbor heuristic produces a tour significantly longer than the optimal one.

Implementation of Greedy Algorithms

Huffman Coding

Huffman coding assigns variable-length binary codes to characters so that more frequent characters get shorter codes. This minimizes the total number of bits needed to encode a message.

1. Create a leaf node for each character, weighted by its frequency.
2. Insert all nodes into a min-priority queue.
3. While more than one node remains in the queue: - Extract the two nodes with the lowest frequencies. - Create a new internal node with these two as children, with frequency equal to their sum. - Insert the new node back into the queue.
4. The remaining node is the root of the Huffman tree.
5. Assign codes by traversing the tree: go left = append 0, go right = append 1.

For frequencies {a: 45, b: 13, c: 12, d: 16, e: 9, f: 5}, the resulting codes might be something like a: 0, b: 101, c: 100, d: 111, e: 1101, f: 1100. Characters with higher frequencies (like a) get shorter codes.

The time complexity is $O(n\log n)$ where n is the number of distinct characters, since each insertion and extraction from the priority queue costs $O(\log n)$.

Dijkstra's Shortest Path Algorithm

Dijkstra's finds the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights.

1. Initialize the distance to the source as 0 and all other distances as ∞ . Mark all vertices as unvisited.
2. Set the source as the current vertex.
3. For the current vertex, examine each unvisited neighbor. Calculate the tentative distance through the current vertex. If it's less than the neighbor's recorded distance, update it.
4. After checking all neighbors, mark the current vertex as visited (it won't be reconsidered).
5. Select the unvisited vertex with the smallest tentative distance as the next current vertex.
6. Repeat steps 3-5 until all vertices are visited or the destination is reached.

Why non-negative weights? Dijkstra's greedy choice assumes that once a vertex is visited, its shortest distance is finalized. A negative-weight edge could later provide a shorter path to an already-visited vertex, violating this assumption. For graphs with negative weights, use Bellman-Ford instead.

Example: In a graph where the shortest path from vertex 0 to vertex 4 is $0 \rightarrow 2 \rightarrow 3 \rightarrow 4$ with total distance 9, Dijkstra's discovers this by progressively locking in the shortest distances to vertices 2, then 3, then 4.

15.2 Divide and Conquer Strategies

Divide and Conquer Strategies

Divide and conquer is one of the most fundamental algorithm design paradigms. It works by splitting a problem into smaller subproblems, solving each one recursively, and then combining the results. Understanding this pattern is essential because it underpins many of the most efficient algorithms you'll encounter, from sorting to matrix multiplication to binary search.

The Divide and Conquer Paradigm

Every divide and conquer algorithm follows three steps:

1. Divide the problem into smaller subproblems of the same type.
2. Conquer each subproblem by solving it recursively. When a subproblem is small enough (the base case), solve it directly.
3. Combine the solutions of the subproblems to build the solution to the original problem.

The key distinction here is that the subproblems are independent and non-overlapping. Each piece gets solved on its own, and you don't re-solve the same subproblem twice. (This is what separates divide and conquer from dynamic programming, which we'll compare below.)

Think of binary search as a minimal example: you divide the sorted array in half, determine which half your target lives in (conquer), and there's nothing to combine because you've narrowed down to a single answer. Merge sort is a fuller example where all three steps do real work.

Design of Divide and Conquer Algorithms

Merge Sort

Merge sort is the textbook divide and conquer sorting algorithm:

1. Divide: Split the unsorted list into two halves.
2. Conquer: Recursively sort each half. The base case is a single-element list, which is already sorted.
3. Combine: Merge the two sorted halves into one sorted list by comparing elements from each half and placing them in order.

The merge step does the heavy lifting. You walk through both sorted halves simultaneously, always picking the smaller element, which takes $O(n)$ time for n total elements.

- Time complexity: $O(n \log n)$ in all cases (best, average, and worst).
- Space complexity: $O(n)$ because the merge step needs a temporary array.

That guaranteed $O(n \log n)$ regardless of input is merge sort's biggest advantage.

Quick Sort

Quick sort also uses divide and conquer, but the division strategy is different:

1. Divide: Select a pivot element (could be the first, last, median, or a random element). Partition the array so that all elements less than or equal to the pivot go to the left, and all elements greater go to the right.
2. Conquer: Recursively apply quick sort to the left and right partitions.
3. Combine: Nothing to do. After partitioning and recursion, the array is already sorted in place.

Here the partition step does the heavy lifting, while the combine step is trivial. This is the opposite of merge sort, where dividing is trivial but merging takes work.

- Average-case time complexity: $O(n \log n)$
- Worst-case time complexity: $O(n^2)$, which happens when the pivot consistently lands at an extreme (e.g., the smallest or largest element every time). Already-sorted or reverse-sorted input with a naive pivot choice triggers this.
- Choosing a random pivot or using median-of-three selection makes the worst case extremely unlikely in practice.

Time Complexity and Recurrence Relations

Divide and conquer algorithms naturally produce recurrence relations that describe their running time. The recurrence captures two costs: the time to solve the subproblems and the time to divide/combine.

Merge sort recurrence:

$$T(n) = 2T(n/2) + O(n)$$

- $2T(n/2)$: you make two recursive calls, each on half the input.
- $O(n)$: the merge step scans through all n elements.

Quick sort average-case recurrence:

$$T(n) = 2T(n/2) + O(n)$$

- $2T(n/2)$: on average, the pivot splits the array roughly in half.
- $O(n)$: the partition step examines each element once.

Both recurrences have the same form, and both solve to $O(n \log n)$. You can solve recurrences like these using two main tools:

- The Master Theorem: A formula that directly gives the solution for recurrences of the form $T(n) = aT(n/b) + O(n^d)$. For merge sort, $a = 2$, $b = 2$, $d = 1$. Since $a = b^d$, the Master Theorem gives $O(n \log n)$.
- Recursion tree method: Draw out the recursive calls as a tree. At each level, sum the work done. The total across all levels gives you the overall complexity. For merge sort, each level does $O(n)$ work and there are $\log n$ levels, giving $O(n \log n)$.

Divide and Conquer vs. Other Techniques

Understanding when to use divide and conquer means understanding how it differs from the other two major paradigms:

	Divide and Conquer	Dynamic Programming	Greedy
Subproblem structure	Non-overlapping	Overlapping	No explicit subproblems
Core idea	Split, solve recursively, combine	Solve subproblems once, store results (memoization/tabulation)	Make the locally optimal choice at each step
Optimality	Guarantees correctness for problems it's designed for	Guarantees optimal solution when optimal substructure holds	Does not always guarantee a global optimum
Space tradeoff	Higher space from recursive call stack	Extra space for storing subproblem results	Typically low space
Examples	Merge sort, quick sort, binary search	Fibonacci sequence, longest common subsequence	Dijkstra's algorithm, Huffman coding

The critical difference between divide and conquer and dynamic programming is overlapping subproblems. If you find yourself solving the same subproblem multiple times in a recursive approach, that's a signal to switch to dynamic programming. If the subproblems are independent, divide and conquer is the right fit.

Greedy algorithms are structurally different: they don't recurse into subproblems at all. They just make one choice at a time and move forward. They're faster and simpler when they work, but they only work when a greedy choice property can be proven.

15.3 Dynamic Programming Fundamentals

Dynamic programming solves complex optimization problems by breaking them into smaller subproblems and storing the results so you never solve the same subproblem twice. This technique is essential whenever a brute-force recursive approach would waste time recomputing the same values over and over. Two properties signal that DP applies: overlapping subproblems and optimal substructure.

Introduction to Dynamic Programming

Characteristics of Dynamic Programming

Optimal substructure means the optimal solution to the whole problem can be built from optimal solutions to its subproblems. For example, the shortest path from A to C through B is the shortest path from A to B plus the shortest path from B to C. If a problem doesn't have this property, DP won't work.

Overlapping subproblems means a naive recursive solution would solve the same subproblems many times. The classic example is the Fibonacci sequence: computing $fib(5)$ recursively calls $fib(3)$ twice, $fib(2)$ three times, and so on. Without storing results, the call tree explodes exponentially.

DP addresses this redundancy through two strategies:

- Memoization (top-down): Write the solution recursively, but cache each result in a table (dictionary or array). Before making a recursive call, check the cache first. If the answer is already there, return it immediately.
- Tabulation (bottom-up): Build a table iteratively, starting from the base cases and working up to the final answer. You fill entries in an order that guarantees every subproblem you need has already been computed.

Both approaches yield the same time complexity. Memoization is often easier to write because it mirrors the recursive structure. Tabulation avoids recursion overhead and makes space optimization easier.

Applications of Dynamic Programming

- Optimization problems: resource allocation, scheduling, knapsack variants
- Graph algorithms: shortest paths (Floyd-Warshall, Bellman-Ford), network flow
- Sequence analysis: DNA sequence alignment, longest common subsequence (LCS), edit distance
- Combinatorics: counting paths in a grid, binomial coefficients, Fibonacci numbers

The common thread is that all of these involve overlapping subproblems and optimal substructure.

Applying Dynamic Programming

Solutions for Classic Problems

Knapsack Problem (0/1)

You have n items, each with a weight and value, and a knapsack with capacity W . The goal is to maximize total value without exceeding the capacity.

1. Define the subproblem: $dp[i][w]$ = maximum value achievable using the first i items with remaining capacity w .
2. Write the recurrence: For each item i , you either skip it or take it (if it fits):
 $dp[i][w] = \max(dp[i-1][w], dp[i-1][w - \text{weight}[i]] + \text{value}[i])$ The first term is "skip item i "; the second is "take item i ."
3. Set base cases: $dp[0][w] = 0$ for all w (no items means no value), and $dp[i][0] = 0$ for all i (no capacity means no value).

4. Fill the table row by row from $i = 1$ to n , column by column from $w = 1$ to W . The answer is $dp[n][W]$.

Longest Common Subsequence (LCS)

Given two strings of lengths m and n , find the length of their longest common subsequence.

1. Define the subproblem: $dp[i][j]$ = length of the LCS of the first i characters of $str1$ and the first j characters of $str2$.
2. Write the recurrence: - If $str1[i] == str2[j]$: $dp[i][j] = dp[i-1][j-1] + 1$ - Otherwise: $dp[i][j] = \max(dp[i-1][j], dp[i][j-1])$
3. Set base cases: $dp[i][0] = 0$ and $dp[0][j] = 0$ (an empty string has no common subsequence with anything).
4. Fill the table row by row. The answer is $dp[m][n]$.

Optimization of Dynamic Programming

Memoization (Top-Down)

1. Write the recursive solution as you normally would.
2. Add a lookup table (hash map or array) initialized to a sentinel value (e.g., -1).
3. At the start of each recursive call, check if the result is already in the table. If so, return it.
4. Otherwise, compute the result, store it in the table, then return it.

This turns an exponential recursive solution into a polynomial one without changing the recursive logic.

Tabulation (Bottom-Up)

1. Create a DP table sized to hold all subproblems.
2. Fill in the base cases.
3. Iterate through the table in an order that ensures every subproblem you depend on is already solved.
4. Read the final answer from the appropriate cell.

Space Optimization

You don't always need the full table. If the recurrence for row i only depends on row $i-1$, you can keep just two rows (or even one row filled in reverse). For example, the 0/1 Knapsack recurrence only looks at the previous row, so you can reduce space from $O(nW)$ to $O(W)$. Similarly, LCS can be reduced from $O(mn)$ to $O(\min(m, n))$ by keeping only two rows.

Complexity Analysis in Dynamic Programming

A useful rule of thumb: DP time complexity = (number of subproblems) $\times$ (time per subproblem).

Time Complexity:

Problem	Subproblems	Time per Subproblem	Total
0/1 Knapsack	$n \times W$	$O(1)$	$O(nW)$
LCS	$m \times n$	$O(1)$	$O(mn)$

Problem	Subproblems	Time per Subproblem	Total
Fibonacci (memoized)	n	$O(1)$	$O(n)$

Note that $O(nW)$ for Knapsack is pseudo-polynomial, not truly polynomial, because W is a numeric value, not the size of the input. If W is encoded in binary, the input length is $\log W$, making the runtime exponential in input size. This distinction matters for understanding why Knapsack is NP-hard despite having a DP solution.

Space Complexity:

- Knapsack: $O(nW)$ for the full table, $O(W)$ with space optimization
- LCS: $O(mn)$ for the full table, $O(\min(m, n))$ with space optimization
- Fibonacci: $O(n)$ with memoization, $O(1)$ if you only track the last two values

15.4 Comparative Analysis of Algorithm Design Techniques

Algorithm design techniques give you a toolkit for tackling complex problems efficiently. Greedy, divide-and-conquer, and dynamic programming each solve different kinds of problems well, and choosing the wrong one can mean the difference between an optimal solution and a slow or incorrect one. This section focuses on how to compare these three approaches and pick the right one for a given problem.

Algorithm Design Techniques

Greedy vs divide-and-conquer vs dynamic programming

These three techniques differ in how they break down and solve problems. Understanding each one's core strategy helps you recognize which fits a given situation.

Greedy algorithms make the locally optimal choice at each step, hoping those choices add up to a globally optimal solution. They're often the fastest and simplest to implement, but they don't always produce the best answer. They work correctly only when the problem has the greedy choice property, meaning local optima genuinely lead to a global optimum. Classic examples: Huffman coding, Dijkstra's shortest path, activity selection.

Divide-and-conquer algorithms recursively split a problem into smaller subproblems, solve each one independently, then combine the results. The key word here is independently: the subproblems don't share work or overlap. Examples include merge sort, quicksort, and binary search. This technique is a natural fit when a problem cleanly divides into parts that don't depend on each other.

Dynamic programming (DP) also breaks problems into subproblems, but it handles the case where those subproblems overlap, meaning a naive recursive approach would re-solve the same subproblem many times. DP stores (memoizes or tabulates) the results of subproblems so each is solved only once. It requires two properties:

- Optimal substructure: the optimal solution to the whole problem can be built from optimal solutions to its subproblems.
- Overlapping subproblems: the same subproblems recur during recursion.

Examples: Fibonacci sequence, longest common subsequence, 0/1 knapsack problem.

A useful way to remember the distinction: divide-and-conquer has independent subproblems, while dynamic programming has overlapping subproblems. If subproblems overlap but you use divide-and-conquer anyway, you'll end up doing redundant work.

Selection of algorithm design techniques

Choosing the right technique is a structured process. Here's how to think through it:

1.

Analyze the problem's structural properties. - Does it have optimal substructure? Can you build the overall optimal solution from optimal solutions to smaller pieces? - Does it have overlapping subproblems? Would a recursive solution re-solve the same subproblems repeatedly? - Does it have the greedy choice property? Can you prove that always picking the locally best option leads to the globally best result?

2.

Clarify your requirements. - Time complexity: How fast does the algorithm need to be? Can you afford $O(n^2)$ or do you need $O(n \log n)$? - Space complexity: How much memory is available? Can you afford a 2D table, or do you need $O(1)$ extra space? - Optimality: Do you need the provably best solution, or is a good approximation acceptable?

3.

Match technique to properties. - If the problem has the greedy choice property (and you can prove it), use greedy. It'll be fast and simple. - If the problem splits into independent subproblems that combine cleanly, use divide-and-conquer. - If the problem has both optimal substructure and overlapping subproblems, use dynamic programming.

A common mistake is jumping to DP when greedy would work, or using greedy when the greedy choice property doesn't actually hold. Always verify the structural properties before committing to a technique.

Trade-offs in algorithm design

	Greedy	Divide-and-Conquer	Dynamic Programming
Typical time	$O(n)$ or $O(n \log n)$	$O(n \log n)$ (varies)	Depends on table size, e.g. $O(n^2)$ or $O(nW)$
Typical space	$O(1)$ or $O(n)$	$O(\log n)$ to $O(n)$ (recursion stack)	$O(n)$ to $O(n^2)$ (memoization table)
Implementation	Usually straightforward	Requires recursion; combining step can be tricky	Recurrence relations and table management add complexity
Optimality guarantee	Only with greedy choice property	Yes (for correctly formulated problems)	Yes (when optimal substructure holds)

A few things to note from this table:

- Greedy wins on speed and simplicity, but you sacrifice guaranteed optimality unless the problem specifically supports it.
- Divide-and-conquer's space cost comes mainly from the recursion stack. For something like merge sort, you also need $O(n)$ auxiliary space for merging.
- DP's space cost can be significant. For a 2D table like in the knapsack problem, you're looking at $O(n \cdot W)$ space, where W is the capacity. Sometimes you can optimize this with space-reduction tricks (e.g., keeping only the previous row of the table).

Combining techniques for complex problems

Some real-world problems are too complex for a single technique. In those cases, you can apply different strategies to different parts of the problem.

Example: Traveling Salesman Problem (TSP)

The TSP asks for the shortest route visiting all cities exactly once. A pure brute-force approach is $O(n!)$, which is unusable for large inputs.

- DP (Held-Karp algorithm) solves TSP exactly in $O(n^2 \cdot 2^n)$ time, which is much better than brute force but still exponential. This works well for small to moderate numbers of cities.
- Greedy heuristics (like nearest-neighbor) give fast approximate solutions in $O(n^2)$ time but don't guarantee optimality.
- A combined approach might use DP to solve small subsets of cities optimally, then use a greedy strategy to connect those subsets together.

General process for combining techniques:

1. Decompose the complex problem into subproblems.
2. Identify which technique best fits each subproblem based on its properties.
3. Solve each subproblem with its matched technique.
4. Combine the subproblem solutions, verifying that the combined result is still correct.
5. Analyze the overall time and space complexity of the combined approach. Sometimes the combination introduces overhead that needs to be accounted for.

The goal is to exploit each technique's strengths where they apply, rather than forcing one approach onto the entire problem.