Data Structures (DI03000021)

GTU Diploma Engineering

Table of Contents I

Data Structures (DI03000021)

GTU Diploma Engineering

Understanding Data in Computer Science

A

t its core, data is a representation of raw facts, concepts, or instructions in a formalized manner suitable for communication, interpretation, or processing by humans or automated means. In computing, all data is represented as binary digits (bits), but high-level programming models abstract this into structured formats.

- **Data vs Information:** Data consists of raw, unorganized facts. Information is processed, structured, and presented in a meaningful context.
- **Data Representation:** On physical hardware, data is stored as high/low voltages corresponding to 1s and 0s.
- **Data Types:** Programming languages classify data into types (e.g., Integer, Float, Character, Boolean) which dictate the size in memory and the operations that can be legally performed.

Data Types vs. Data Structures

I

t is crucial to distinguish between a data type (a classification of data) and a data structure (an organization of data). While a data type defines the kind of values a variable can hold, a data structure defines how multiple data elements are organized, stored, and accessed collectively in memory.

- **Data Type:** A set of values and a set of operations allowed on those values (e.g., integer type supports addition, subtraction, division).
- **Data Structure:** A physical or logical organization of data elements designed to facilitate efficient access and modifications.
- **Examples:** An integer is a primitive data type requiring a fixed number of bytes (e.g., 4 bytes in Java). An array is a data structure storing a collection of these integers contiguously

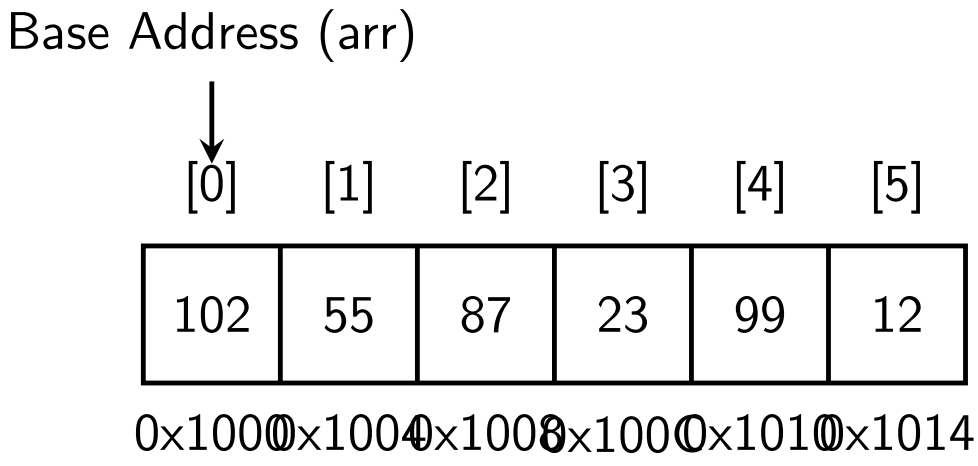
Defining a Data Structure

A

data structure is a specialized format for organizing, processing, retrieving, and storing data. It is characterized by its mathematical logical model, physical storage representation, and the set of algorithms that operate on it. Choosing the right data structure directly impacts algorithm efficiency and time/space complexity.

- **Components of a Data Structure:** The logical model (the abstract mathematical definition) and the physical implementation (how it is laid out in RAM).
- **Operation Set:** Every data structure supports a fundamental set of operations, typically including Traversal, Insertion, Deletion, Searching, Sorting, and Merging.
- **Efficiency:** Different structures are optimized for different tasks. For example, arrays offer $O(1)$ random access, whereas

Memory Layout of Data



- **Contiguous Memory Allocation:** An array stores elements in adjacent memory locations.
- **Address Calculation:** The address of element at index i is: $\text{Address}(\text{arr}[i]) = \text{BaseAddress} + i \times \text{sizeof}(\text{DataType})$.
- **Random Access:** Due to contiguous allocation, accessing any arbitrary element takes constant time, $O(1)$.

Abstract Data Types (ADT)

A

n Abstract Data Type (ADT) is a mathematical model for data types where the data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations.

- **Abstraction:** An ADT defines "WHAT" operations can be performed, completely hiding "HOW" those operations are implemented (information hiding).
- **Separation of Concerns:** The user interacts with the ADT interface. The programmer writes the underlying implementation (using concrete data structures).
- **Examples:** Stack (LIFO: push, pop, peek), Queue (FIFO: enqueue, dequeue), List (get, insert, remove).

Types of Data Structures: Linear vs. Non-Linear

D

ata structures are broadly classified into Linear and Non-Linear based on how their elements are logically organized. Linear data structures arrange elements sequentially, where each element has a single predecessor and successor (except the first and last elements). Non-Linear structures arrange elements hierarchically or in networks.

- **Linear Data Structures:** Elements form a sequence. Examples include Arrays, Linked Lists, Stacks, and Queues. Traversing them is simple and linear.
- **Non-Linear Data Structures:** Elements are connected in a multi-level hierarchy or arbitrary networks. Examples include Trees (parent-child relationship) and Graphs (network of nodes).
- **Complexity:** Linear structures have simpler traversal

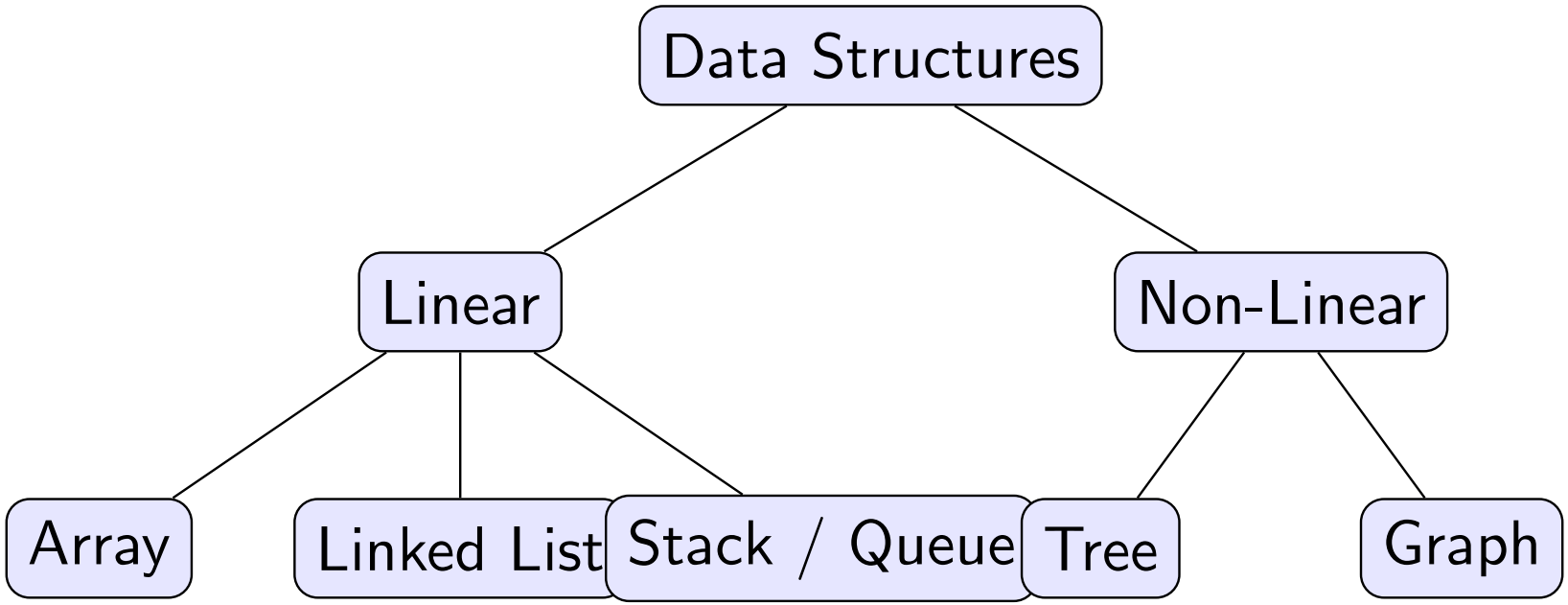
Types of Data Structures: Static vs. Dynamic

A

nother key classification is based on memory allocation behavior: Static vs. Dynamic. Static data structures have a fixed size determined at compile time, whereas dynamic data structures can grow or shrink in size during program execution, optimizing memory utilization.

- **Static Data Structures:** Size is allocated during compilation. Memory cannot be resized or reallocated at runtime. Example: Standard Arrays.
- **Dynamic Data Structures:** Size can be modified dynamically at runtime. Memory is allocated from the heap using pointer manipulation. Example: Linked Lists, Dynamic Arrays (`vector`/`ArrayList`).
- **Trade-offs:** Static structures have zero memory management overhead and faster access, but can lead to memory wastage

Classification of Data Structures



- **Hierarchical Breakdown:** Data structures are divided into Linear (sequential) and Non-Linear (hierarchical/graphical) groups.
- **Linear Sub-types:** Contiguous arrays, pointer-based linked lists, and restricted access structures (Stack – LIFO, Queue – FIFO).
- **Non-Linear Sub-types:** Trees represent hierarchical relationships (e.g., BSTs, Heaps), while Graphs represent network-like, many-to-many relationships.

Lecture Summary & Course Preview

I

n this introductory lecture, we established the fundamental definitions of data, data types, and data structures. We distinguished between ADTs and their concrete implementations, and classified structures into Linear/Non-Linear and Static/Dynamic. These concepts form the bedrock of software engineering and algorithm design.

- **Key Takeaway:** A data structure is not just data storage; it is the union of physical memory layout + operations + algorithms.
- **Goal of the Course:** Learn how to analyze, select, implement, and optimize data structures to solve complex computational problems.
- **Next Lecture Preview:** In Lecture 2, we will dive deep into the Array data structure, looking at multi-dimensional arrays

Data Structures (DI03000021)

GTU Diploma Engineering

Primitive Data Structures

P

rimitive data structures are the most basic data types predefined by the programming language. They serve as the building blocks for more complex structures and map directly to machine-level memory layouts.

- **Predefined Types:** Integer, Float, Character, and Boolean are classic primitive data types.
- **Hardware Mapping:** These types represent values directly in computer memory (e.g., 32-bit two's complement for integers, IEEE 754 standard for floating-point representation).
- **Fixed Size:** They have a fixed size in memory which is architecture-dependent (e.g., 4 bytes for an int, 1 byte for a char).
- **Operations:** Support basic machine operations directly, such as arithmetic (+, -, *, /) and logical comparisons.

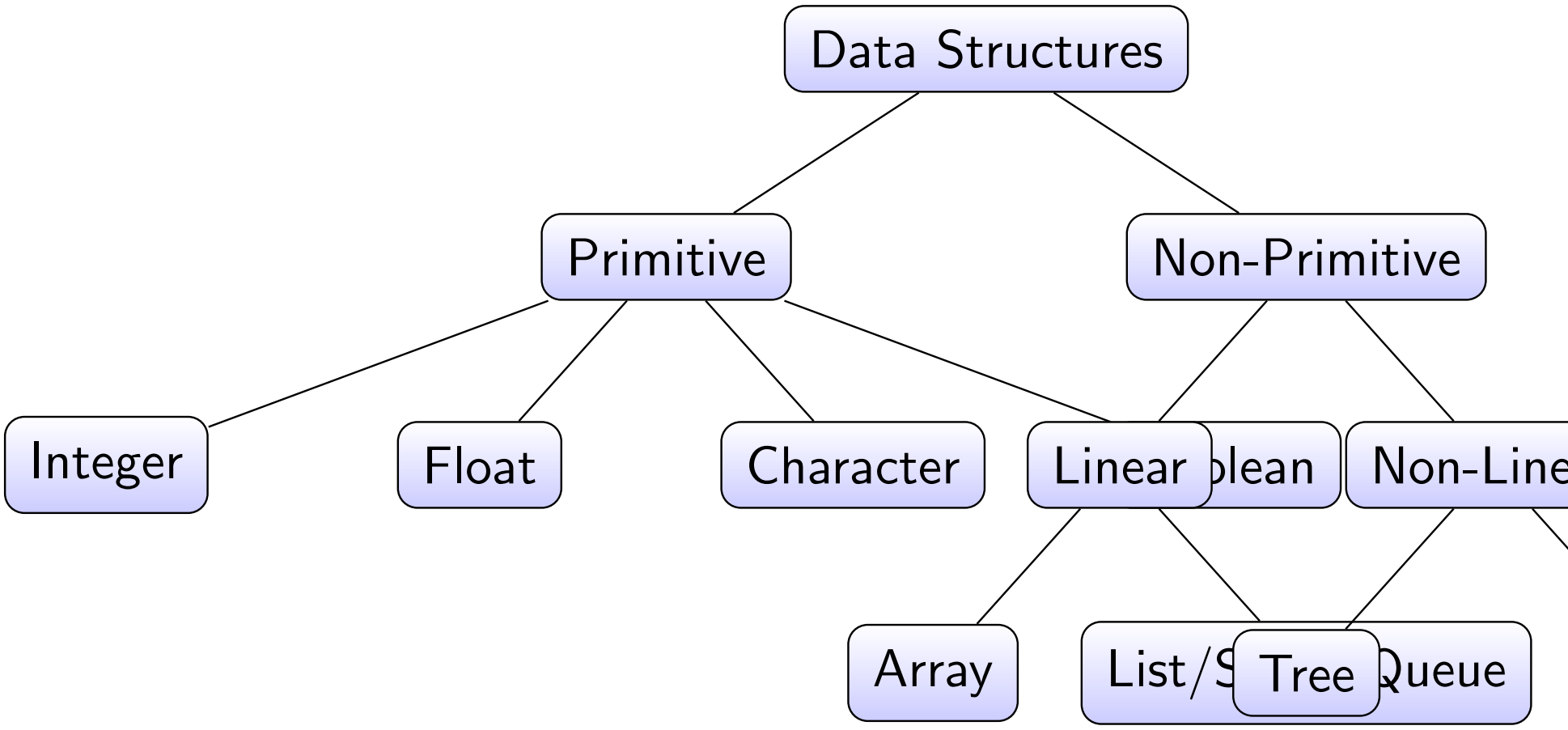
Non-Primitive Data Structures

N

on-primitive (or composite) data structures are user-defined or language-provided structures created by combining primitive types. They are designed to group multiple values and manage complex relationships.

- **Derived Nature:** Created using primitive types and other non-primitive types as building blocks.
- **Categorization:** Divided primarily into Linear (Arrays, Lists, Stacks, Queues) and Non-Linear (Trees, Graphs) categories.
- **Reference/Pointer Based:** Often managed using references or pointers to memory locations rather than storing value copies directly.
- **Dynamic Allocation:** Can often grow or shrink dynamically in memory (e.g., linked list nodes allocated on the heap).

Classification Hierarchy



- **Visual Hierarchy:** The diagram illustrates the branching of Data Structures into Primitive and Non-Primitive.
- **Primitive Branch:** Represents atomic types like Integer, Float, Char, and Bool.

Linear Data Structures

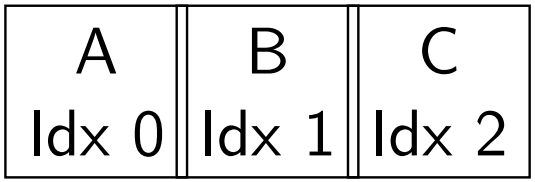
L

inear data structures organize elements in a sequential, 1D order. Each element has a unique predecessor and a unique successor, except for the first and last elements.

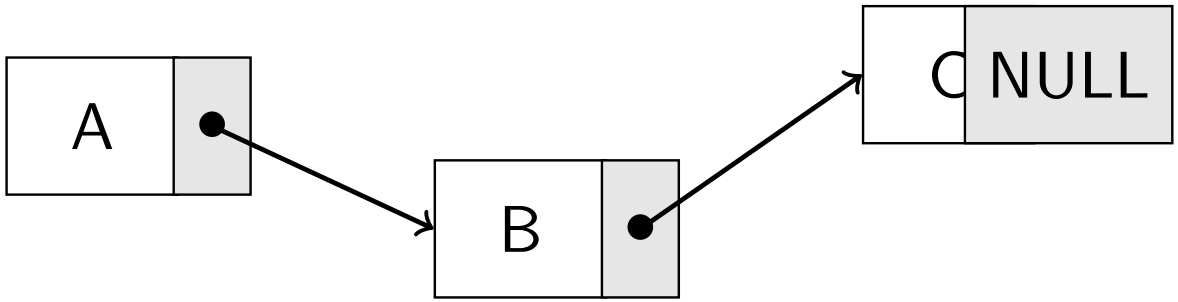
- **Sequential Ordering:** Elements are accessed in a single, predictable linear traversal sequence.
- **Arrays:** Contiguous memory blocks with static size and constant-time $O(1)$ random access using indices.
- **Linked Lists:** Non-contiguous nodes connected via pointers, allowing dynamic resizing and $O(1)$ insertion/deletion if the position is known.
- **Restricted Access:** Stacks follow LIFO (Last-In-First-Out); Queues follow FIFO (First-In-First-Out) policies.

Array vs. Linked List Memory Visual

Array (Contiguous Memory)



Linked List (Pointers)



- **Contiguous Array:** Elements are stored in adjacent memory locations, allowing fast index math:
  Address $=$ Base $+$ Index $\times$ Size.
- **Linked Pointer Links:** Nodes can be scattered anywhere in the heap memory, linked explicitly by next-node pointers.
- **Tradeoffs:** Arrays offer fast read access, whereas Linked Lists

Non-Linear Data Structures

N

on-linear data structures organize elements hierarchically or interconnectedly. An element can be attached to one or more other elements, forming multi-dimensional relationships.

- **No Single Sequence:** Cannot be traversed in a single run; traversals are multi-dimensional (e.g., Breadth-First, Depth-First).
- **Trees:** Hierarchical structure containing a root node and children nodes, preventing cycle formation. Used for representing family trees, folder systems, and search indices.
- **Graphs:** Networks of vertices (nodes) and edges. Edges can be directed/undirected, weighted/unweighted. Used for social networks, web search links, and maps.
- **Complexity:** Algorithms for non-linear structures are often recursive and require tracking visited nodes to avoid infinite

Introduction to Algorithms

A

n algorithm is a step-by-step, finite sequence of well-defined, computer-implementable instructions to solve a specific problem or perform a computation. Data structures store the data; algorithms process it.

- **Finiteness:** The algorithm must terminate after a finite number of steps.
- **Definiteness:** Each step must be precisely defined; actions to be carried out must be rigorously and unambiguously specified.
- **Input and Output:** An algorithm takes zero or more inputs and produces one or more well-defined outputs.
- **Effectiveness:** Operations must be basic enough to be carried out exactly and in a finite length of time.

Algorithm Analysis: Complexity

T

o evaluate the efficiency of an algorithm, we analyze its execution time and memory usage relative to the input size n. This is quantified using Big O notation.

- **Time Complexity:** The rate of growth of the execution time as a function of the input size n, denoted as $O(g(n))$.
- **Space Complexity:** The total memory space required by the algorithm, including auxiliary space and the input data size.
- **Worst-Case Analysis:** Focuses on the maximum steps an algorithm could take, guaranteeing a performance upper bound.
- **Common Complexities:** $O(1)$ constant, $O(\log n)$ logarithmic, $O(n)$ linear, $O(n \log n)$ linearithmic, and $O(n^2)$ quadratic.

Summary and Key Takeaways

D

ata structures and algorithms form the core foundations of computer science. Selecting the correct data structure dictates the efficiency of the algorithm that processes it.

- **Primitive vs. Non-Primitive:** Primitive structures represent hardware-level types; Non-primitive structures combine them to model real-world concepts.
- **Linear vs. Non-Linear:** Linear structures (Arrays, Lists) organize elements sequentially; Non-linear structures (Trees, Graphs) organize them hierarchically or dynamically.
- **Algorithms & Data:** Algorithms are the logic that manipulates data structures. Their efficiency is measured via Time and Space complexity analysis.
- **Up Next:** Detailed exploration of Arrays, dynamic array scaling, and basic searching algorithms in Lecture 3.

Data Structures (DI03000021)

GTU Diploma Engineering

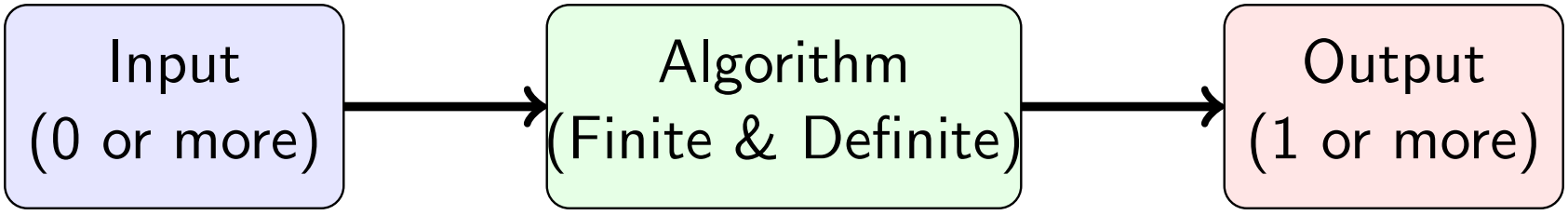
Defining an Algorithm

A

n algorithm is a step-by-step procedure or set of rules to solve a computational problem in a finite amount of time and space.

- **Finiteness**: The algorithm must terminate after a finite number of steps.
- **Definiteness**: Each step must be precisely defined and unambiguous; the actions to be carried out must be rigorously specified.
- **Input**: Zero or more quantities are externally supplied as inputs.
- **Output**: One or more quantities are produced as outputs, having a specific relation to the inputs.
- **Effectiveness**: Every instruction must be sufficiently basic that it can in principle be carried out by a person using only paper and pencil.

Key Characteristics: Solvability and Determinism



- **Input and Output constraints**: An algorithm maps a well-defined input domain to a well-defined output domain.
- **Feasibility**: The operations must be feasible on the target computational architecture.
- **Language Independence**: The logic must be expressible in pseudocode, mathematical notation, or any programming language.

Algorithmic Complexity & Resource Estimation

W

hy do we analyze algorithms? To predict resource consumption (CPU time and memory) without depending on specific hardware configurations.

- **Theoretical Analysis**: Focuses on how execution time or memory scale as input size n grows to infinity.
- **Empirical Analysis**: Measures actual execution time on concrete inputs, which varies based on processor speed, compiler optimizations, and system load.
- Complexity analysis isolates the mathematical properties of the algorithm from engineering artifacts.

Time Complexity: Counting Operations

T

ime Complexity represents the running time of an algorithm as a function of the input size $T(n)$.

- We identify "basic operations"—those that contribute the most to the total running time (e.g., key comparisons, arithmetic ops).
- Let $C(n)$ be the count of basic operations. The running time $T(n)$ is approximately $c \cdot C(n)$, where c is the execution time of a single basic operation.
- **Worst-case Complexity**: Maximum steps taken by the algorithm on any input of size n.
- **Best-case Complexity**: Minimum steps taken on any input of size n.
- **Average-case Complexity**: Expected steps taken over a probability distribution of all possible inputs of size n.

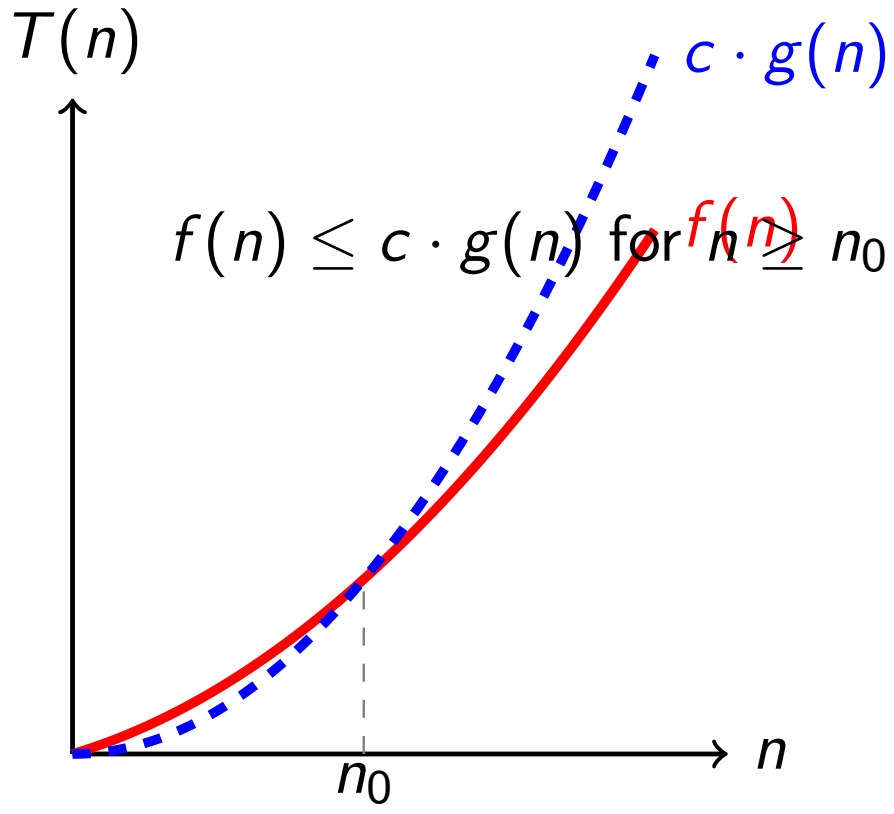
Space Complexity: Memory Footprint

S

pace Complexity is the amount of memory an algorithm needs to run to completion as a function of the input size $S(n)$.

- **Fixed Part**: Space required by instructions, simple variables, constants, and fixed-size structures (independent of input size).
- **Variable Part**: Space required by dynamic structures, recursive call stack frames, and variables whose size depends on input size n.
- **Auxiliary Space**: The temporary or extra space used by the algorithm, excluding the input size itself.
- For example, Merge Sort has an auxiliary space complexity of $O(n)$ due to temporary arrays, whereas Quick Sort uses $O(\log n)$ stack space.

Understanding Asymptotic Behavior



- Asymptotic analysis evaluates performance as the input size n approaches infinity.
- It filters out constants and low-order terms that become insignificant for very large datasets.
- Standard notations: Big-Oh (O) for upper bound, Omega (Ω) for lower bound, and Theta (Θ) for tight bound.

Big-Oh (O) Notation: Upper Bound

f(n) = O(g(n)) *if there exist positive constants c and* n_0 such that $0 \leq f(n) \leq c \cdot g(n)$ for all $n \geq n_0$.

- Provides an asymptotic upper bound for the worst-case behavior of an algorithm.
- Example: If $f(n) = 3n^2 + 5n + 8$, then $f(n) = O(n^2)$ using $c = 16$ and $n_0 = 1$.
- It guarantees that the growth rate of the algorithm's running time does not exceed the growth rate of $g(n)$.

Hierarchy of Growth Rates

Ordered from fastest to slowest growth (most efficient to least efficient):

- **Constant**: $O(1)$ — Array element access by index.
- **Logarithmic**: $O(\log n)$ — Binary search.
- **Linear**: $O(n)$ — Linear search, finding max value in unsorted array.
- **Linearithmic**: $O(n \log n)$ — Merge Sort, Heap Sort.
- **Quadratic**: $O(n^2)$ — Bubble Sort, Insertion Sort.
- **Cubic**: $O(n^3)$ — Matrix multiplication (naive).
- **Exponential**: $O(2^n)$ — Solving Tower of Hanoi.
- **Factorial**: $O(n!)$ — Naive Traveling Salesperson Problem.

Analyzing a Simple Code Snippet

Let us analyze the time complexity of a nested loop:

```
for (int i = 0; i < n; i++) {
  for (int j = 0; j < n; j++) {
    // Basic operation: O(1)
  }
}
```

- The outer loop runs n times.
- For each iteration of the outer loop, the inner loop runs n times.
- The basic operation is executed exactly $n \times n = n^2$ times.
- Thus, the time complexity of this code fragment is $O(n^2)$ (Quadratic time complexity).
- **Resource trade-offs**: We often trade space for time (e.g., using a Hash Map to reduce search time from $O(n)$ to $O(1)$ at the cost of $O(n)$ space).

Data Structures (DI03000021)

GTU Diploma Engineering

Understanding Big Omega Notation

Big Omega Notation

Big Omega (Ω) notation provides an asymptotic lower bound on the running time of an algorithm, representing the best-case scenario or a minimum rate of growth.

- We write $T(n) = \Omega(g(n))$ if the growth rate of $T(n)$ is greater than or equal to the growth rate of $g(n)$.
- It is used to describe the best-case execution time of an algorithm or to establish lower bounds on complexity classes of problems.
- While Big O provides a pessimistic (upper bound) view, Big Omega provides an optimistic (lower bound) view of algorithm performance.
- For example, the best-case time complexity of Bubble Sort is $\Omega(n)$ when the array is already sorted.

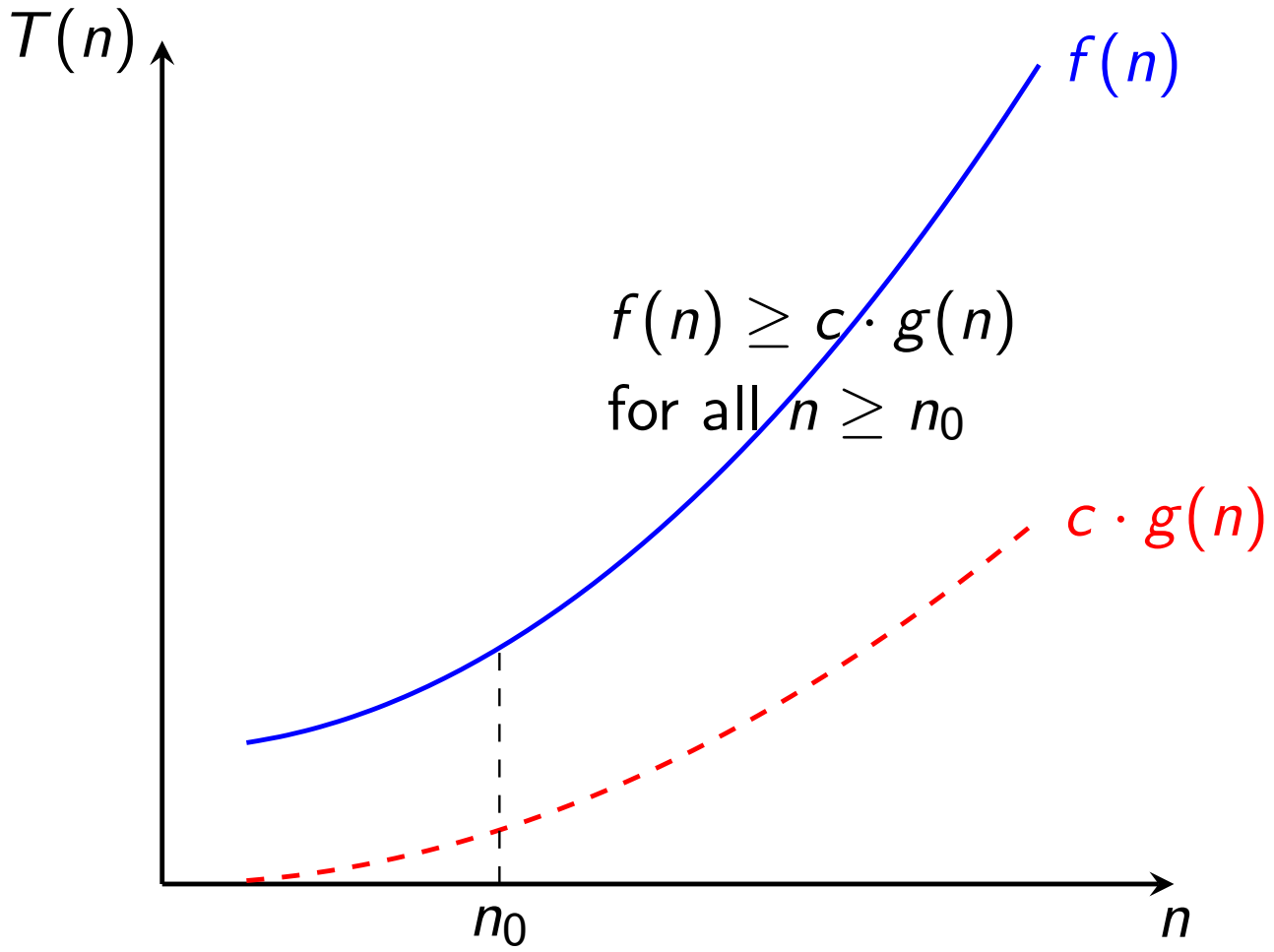
Mathematical Definition of Big Omega

Formal Definition of Big Omega

Formal definition: $f(n) = \Omega(g(n))$ if and only if there exist positive constants c and n_0 such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.

- The constant c must be strictly positive ($c > 0$) and is used to scale $g(n)$.
- The threshold value n_0 ($n_0 \geq 1$) defines the point at which the lower bound becomes permanent.
- The definition implies that for all sufficiently large inputs, the function $f(n)$ is at least a constant multiple of $g(n)$.
- To establish a Big Omega relation, one must identify a specific pair of constants (c, n_0) that satisfies this inequality.

Visualizing Big Omega Notation



- The graphical representation illustrates that $f(n)$ lies above the scaled function $c \cdot g(n)$ for all values of n greater than or equal to n_0.
- Before the threshold n_0, the relative order of the functions

Proving Big Omega: Step-by-Step Example

Problem Statement

Problem: Prove that $f(n) = 3n^2 + 5n + 2$ is $\Omega(n^2)$.

- We must find positive constants c and n_0 such that $3n^2 + 5n + 2 \geq c \cdot n^2$ for all $n \geq n_0$.
- Observe that for all $n \geq 1$, we have $3n^2 + 5n + 2 \geq 3n^2$ because the terms $5n$ and 2 are positive.
- Thus, we can choose $c = 3$ and $n_0 = 1$. The inequality $3n^2 + 5n + 2 \geq 3n^2$ holds true for all $n \geq 1$.
- Alternative proof: We could also choose $c = 1$ and $n_0 = 1$, since $3n^2 + 5n + 2 \geq n^2$ holds for all $n \geq 1$. Any valid constants suffice.

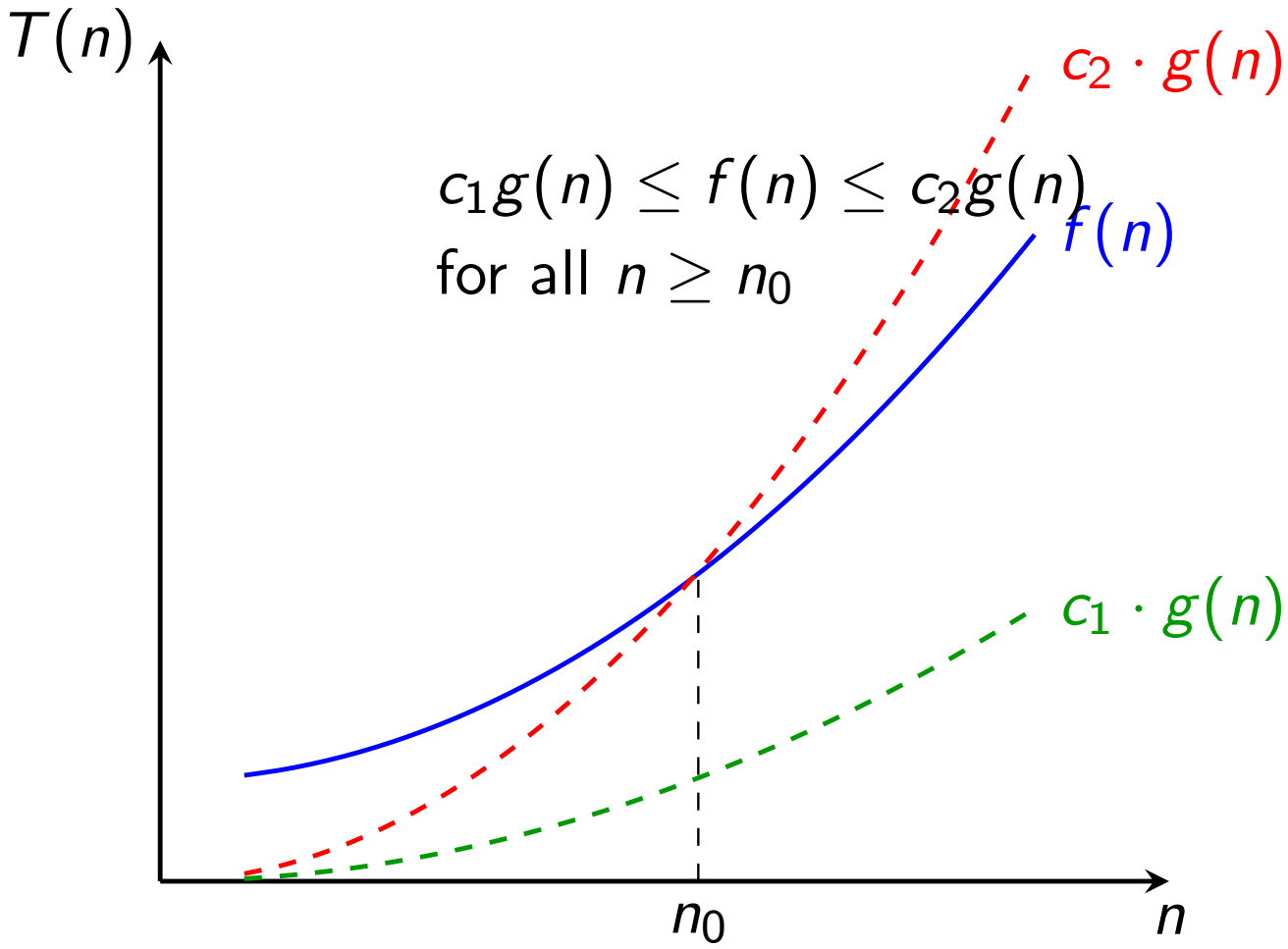
Introduction to Big Theta Notation

Big Theta Notation

Big Theta (Θ) notation defines an asymptotically tight bound, meaning it bounds a function from both above and below within constant factors.

- We write $f(n) = \Theta(g(n))$ if and only if $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ simultaneously.
- It provides the exact asymptotic rate of growth of a function, leaving no gap between the upper and lower bounds.
- For example, finding the maximum element in an unsorted array of size n always takes $\Theta(n)$ time because it requires checking all elements.
- In contrast, searching for an element in an unsorted array takes $O(n)$ in the worst case and $\Omega(1)$ in the best case, so it is not $\Theta(n)$.

Visualizing Big Theta Notation



- The graph shows the function $f(n)$ sandwiched between the lower bound $c_1 \cdot g(n)$ and the upper bound $c_2 \cdot g(n)$ for all $n \geq n_0$.
- Beyond the threshold n_0, $f(n)$ is restricted to a corridor

Array Representation: Memory Layout

Array Structure

An array is a linear data structure containing elements of the same data type stored in contiguous memory locations, enabling direct access.

- Contiguous allocation: Elements are positioned adjacent to one another in physical memory without any intervening gaps.
- Base Address (B): The starting memory address of the very first element (index 0) of the array.
- Element Size (W): The number of bytes required to store a single element (e.g., 4 bytes for an integer, 8 bytes for a double).
- Direct calculation of address allows constant time $O(1)$ access to any index, bypassing sequential traversal.

Addressing Formulas: 1D and 2D Arrays

Indexing Logic

Compilers translate array indices into memory addresses using mathematical mapping formulas based on dimensions and layout orders.

- 1D Array Address Formula: $\text{Address}(A[i]) = B + i \times W$, where B is the base address and W is the size of each element.
- 2D Array Row-Major Order (RMO): Elements are stored row-by-row in memory. Formula for $A[i][j]$ in an $M \times N$ array: $\text{Address}(A[i][j]) = B + (i \times N + j) \times W$.
- 2D Array Column-Major Order (CMO): Elements are stored column-by-column in memory. Formula for $A[i][j]$ in an $M \times N$ array: $\text{Address}(A[i][j]) = B + (j \times M + i) \times W$.
- These indexing operations require only simple arithmetic multiplication and addition, ensuring fast $O(1)$ execution.

Summary of Big Omega, Big Theta, and Arrays

Summary

Recap of foundational tools for evaluating algorithm efficiency and basic data structures.

- Big Omega (Ω) establishes a lower limit on growth, ensuring the algorithm will not run faster than this bound in the best case.
- Big Theta (Θ) establishes a tight, precise bound, indicating that the upper and lower limits are asymptotically identical.
- Arrays represent the simplest contiguous data structure, offering $O(1)$ random access but requiring $O(n)$ time for insertion and deletion.
- Mastering asymptotic bounds and physical memory representations forms the basis for studying advanced data structures and algorithm analysis.

Data Structures (DI03000021)

GTU Diploma Engineering

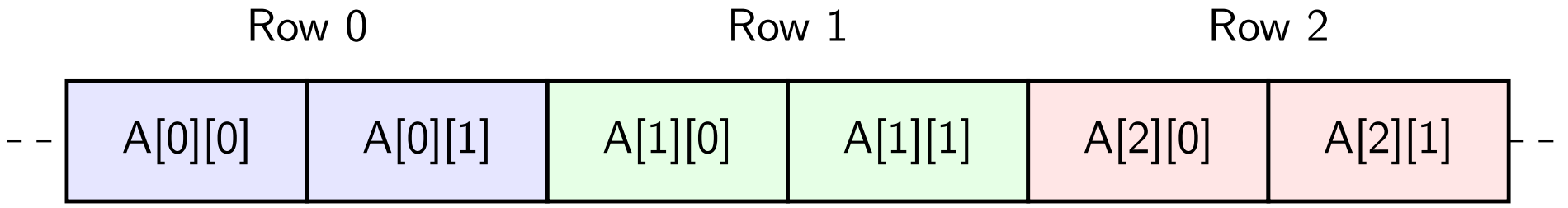
Multi-Dimensional Arrays Representation

Multi-Dimensional Array Representation

A multi-dimensional array is a logical abstraction representing data in grid formats. However, physical memory (RAM) is a linear, 1D array of bytes.

- To store a 2D array of size $M \times N$ in memory, it must be flattened (linearized) into a 1D sequence.
- The mapping from 2D coordinates (row, col) to 1D index is determined by the array's storage order.
- **Row-Major Order (RMO)** stores consecutive elements of a row contiguously.
- **Column-Major Order (CMO)** stores consecutive elements of a column contiguously.

Row-Major Order Layout



Row-Major Order: Contiguous Rows in 1D Memory

- Each row's elements are stored back-to-back in memory.
- Used by C, C++, Java, C#, and Python (NumPy default).
- Accessing elements along rows takes advantage of processor cache line prefetching.

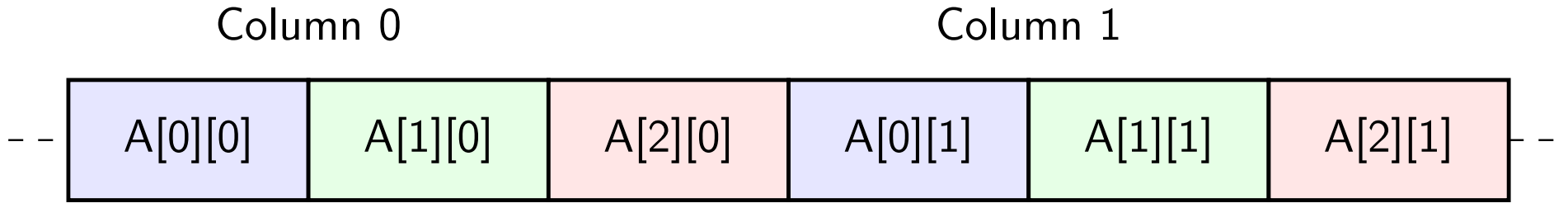
Row-Major Addressing Formula

Row-Major Addressing Formula

For a 2D array $A[M][N]$ with base address B and element size W bytes, the memory address of element $A[i][j]$ is computed mathematically.

- **Under 0-based indexing:**
  $\text{Address}(A[i][j]) = B + (i \times N + j) \times W$
- **Under 1-based indexing:**
  $\text{Address}(A[i][j]) = B + ((i-1) \times N + (j-1)) \times W$
- In a 3D array $A[D_1][D_2][D_3]$, the address of $A[i][j][k]$ is:
  $B + (i \times D_2 \times D_3 + j \times D_3 + k) \times W$
- The time complexity to compute the physical address of any element is $O(1)$.

Column-Major Order Layout



Column-Major Order: Contiguous Columns in 1D Memory

- Each column's elements are stored back-to-back in memory.
- Used by Fortran, MATLAB, R, and Julia.
- In this layout, $A[i][j]$ is physically adjacent to $A[i+1][j]$ rather than $A[i][j+1]$.

Column-Major Addressing Formula

Column-Major Addressing Formula

For a 2D array $A[M][N]$ with base address B and element size W bytes, the memory address of element $A[i][j]$ in Column-Major layout shifts columns first.

- **Under 0-based indexing:**
  $\text{Address}(A[i][j]) = B + (j \times M + i) \times W$
- **Under 1-based indexing:**
  $\text{Address}(A[i][j]) = B + ((j-1) \times M + (i-1)) \times W$
- In a 3D array $A[D_1][D_2][D_3]$, the address of $A[i][j][k]$ is:
  $B + (k \times D_2 \times D_1 + j \times D_1 + i) \times W$
- Incorrect assumptions about storage layout can lead to inefficient linear scans.

Cache Locality and Memory Strides

Cache Locality

Modern CPU caches load memory in blocks (cache lines, usually 64 bytes). Spatial locality determines cache performance.

- Accessing contiguous memory locations results in a "cache hit", whereas non-contiguous access causes a "cache miss".
- In C/C++ (Row-Major), nested loops should iterate Row-first: `for i { for j { ... A[i][j] ... } }`
- If loops are nested Column-first: `for j { for i { ... A[i][j] ... } }`, memory access stride is N elements, causing high cache misses.
- Aligning matrix traversal with the underlying layout can optimize runtime by several factors.

Array Operations: Insertion Mechanics

Insertion Mechanics

Inserting an element at an arbitrary index in a contiguous 1D array requires shifting elements to maintain sequence integrity.

- **Check for overflow:** ensure the current size is strictly less than the allocated physical capacity.
- **Shifting right:** elements from the end down to the target insertion index must be moved forward by one index.
- **Write element:** place the new data value at the target index which is now safely vacated.
- **Size increment:** update the active count of array elements.

Algorithm: Array Insertion in C++

```
bool insertAt(int arr[], int &size, int capacity,
    if (size >= capacity || index < 0 || index >
        return false; // Bounds/Overflow check
    }
    for (int i = size - 1; i >= index; i--) {
        arr[i + 1] = arr[i]; // Shift right
    }
    arr[index] = value; // Write new value
    size++; // Update size
    return true;
}
```

- The shift loop runs in reverse to prevent overwriting existing elements.
- Worst-case time complexity: $O(N)$ when inserting at index 0 (all N elements shift).
- Best-case time complexity: $O(1)$ when inserting at index `size`

Summary of Array Representations & Operations

Summary

Understanding physical memory layout and array overheads guides structural design choices in applications.

- Multi-dimensional arrays must linearize: row-major for C/C++ family, column-major for Fortran/scientific environments.
- Cache-friendly code matches iteration orders with the memory layout to optimize hardware utilization.
- Static arrays offer $O(1)$ random access but require $O(N)$ insertion/deletion operations due to data shifting.
- Dynamic sizing can be achieved by reallocation, but insertion remains asymptotically bounded by $O(N)$.

Data Structures (DI03000021)

GTU Diploma Engineering

Search Operation in Linear Data Structures

L

ocating the position of a target element K within a collection of elements.

- Linear Search: Scan sequentially from start to end. Best case: $O(1)$, Worst case: $O(n)$, Average case: $O(n)$. Used for unsorted arrays or linked lists.
- Binary Search: Divide-and-conquer on sorted collections. Compares target with middle element, reducing search space by half. Time complexity: $O(\log n)$.
- Search efficiency depends highly on the underlying physical storage representation (contiguous memory vs node-based pointers).

Binary Search Algorithm Implementation

```
int binarySearch(int arr[], int low, int high, in
  while (low <= high) {
    int mid = low + (high - low) / 2;
    if (arr[mid] == x) return mid;
    if (arr[mid] < x) low = mid + 1;
    else high = mid - 1;
  }
  return -1;
}
```

- Avoid integer overflow in midpoint calculation: use `low + (high - low) / 2` instead of `(low + high) / 2`.
- Iterative approach requires $O(1)$ auxiliary space, whereas recursive implementation consumes $O(\log n)$ stack frames.
- Requires array elements to be strictly sorted, which introduces an $O(n \log n)$ sorting pre-processing overhead.

Traversing Data Structures

S

ystematically visiting each node/element in a data structure exactly once to process or display data.

- Linear traversal: Iterating through arrays or linked lists from head to tail. Time complexity is strictly linear, $O(n)$.
- Non-linear traversal: Tree and graph traversals (Pre-order, In-order, Post-order, Breadth-First, Depth-First).
- Applications: Printing elements, computing aggregate metrics (sum, average, max), serialization, and garbage collection sweeping.

Updating Elements: Mutating Values

M

odifying the data payload of an existing element within a structure without changing its physical position.

- Index-based Update (Arrays): Direct memory addressing using base address + index * element size. Completes in $O(1)$ time.
- Value-based Update (Linked Lists / Unsorted Arrays): Requires a search operation first to locate the target node, then writes the new value.
- Time complexity of value-based update: $O(n)$ for linear search, $O(\log n)$ for binary search on sorted arrays, or $O(1)$ if pointer to node is already given.
- In sorted arrays, updating an element's value may require repositioning (shifting) to preserve sorting order, escalating time complexity to $O(n)$.

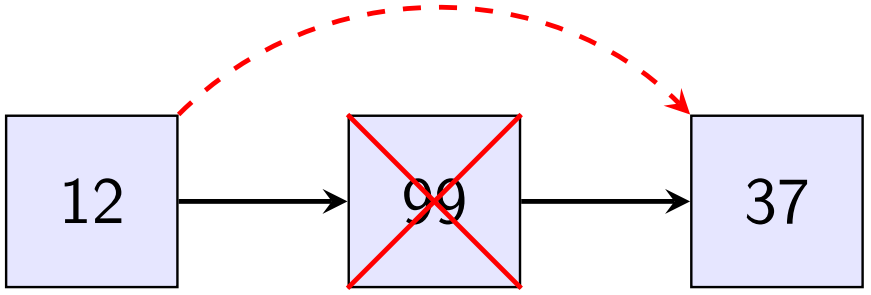
The Delete Operation

R

emoving an existing element from a data structure and restoring structural integrity.

- Contiguous Memory (Array): Deleting an element requires shifting subsequent elements left to close the gap. Time complexity: $O(n)$.
- Node-Based (Linked List): Deleting a node involves updating pointers of adjacent nodes to bypass the target node. Space is deallocated.
- Linked list deletion: $O(1)$ pointer redirection, but finding the predecessor node requires $O(n)$ search unless a doubly-linked list is used.
- Lazy Deletion: Mark elements as deleted without immediately freeing space or shifting. Amortizes cleanup overhead to a later phase.

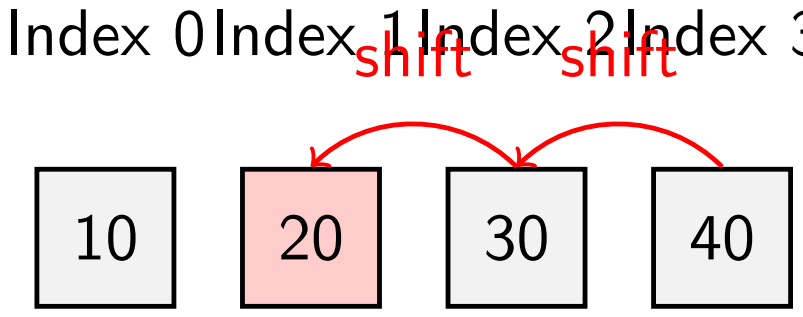
Visualizing Linked List Deletion



To Be Deleted

- Initial state: Node A points to Node B, and Node B points to Node C.
- Deletion action: Redirection of Node A's next pointer to bypass Node B and point directly to Node C.
- Post-action: Node B is isolated from the list and can be safely deallocated in memory using `free()` or garbage collection.

Visualizing Array Deletion and Element Shifting



- Target deletion at Index 1 (value 20).
- Elements at indices greater than the target index (Index 2 and 3) must be shifted one position to the left.
- Worst-case scenario occurs when deleting the first element (index 0), requiring $n - 1$ shifts, making array deletion $O(n)$.

Operation Complexity Comparison

S

ummary of Time Complexities for Core Operations across Sequential (Array) and Linked (List) structures.

- Search: Array (Unsorted: $O(n)$, Sorted: $O(\log n)$), Singly Linked List ($O(n)$).
- Traversal: Array ($O(n)$), Singly Linked List ($O(n)$). Visited elements are linear in size.
- Update: Array (Index-based: $O(1)$, Search-based: $O(n)$), Linked List (Search-based: $O(n)$).
- Delete: Array ($O(n)$ due to shifting), Linked List ($O(1)$ given pointer to node predecessor, otherwise $O(n)$ search time).

Key Takeaways & Summary

C

ore operations form the building blocks of all complex, abstract data structures.

- Choice of data structure dictates the time and space efficiency of search, update, traverse, and delete operations.
- Contiguous layouts support fast random access but suffer from slow insertion and deletion due to memory shifts.
- Linked layouts allow fast modification if pointers are known, but lose direct random indexing and binary search capabilities.
- In the next lecture, we will look at how these four primitives are implemented in dynamic binary search trees.

Data Structures (DI03000021)

GTU Diploma Engineering

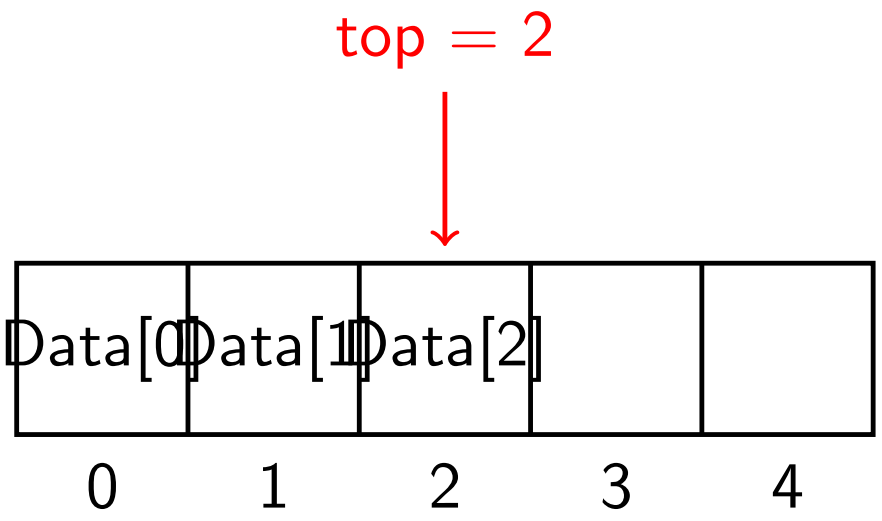
The Stack Abstract Data Type (ADT)

F

ormal mathematical definition of Stack ADT operations and behaviors.

- Defined as an ordered list where insertions and deletions occur at a single end called the 'top'.
- Core operations: `push(element)` $\rightarrow$ `void`, `pop()` $\rightarrow$ `element`, `peek()` / `top()` $\rightarrow$ `element`, `isEmpty()` $\rightarrow$ `boolean`, `isFull()` $\rightarrow$ `boolean`.
- LIFO constraint: For any elements A and B inserted in that order, B must be removed before A can be removed.

Array-Based Stack Layout



- We allocate a static 1D array of fixed maximum size (MAX) to hold the stack elements.
- An integer tracking variable 'top' stores the index of the topmost element in the array.
- Initialization state: 'top' is set to -1, indicating the stack is empty.

Initialization and Boundary Conditions

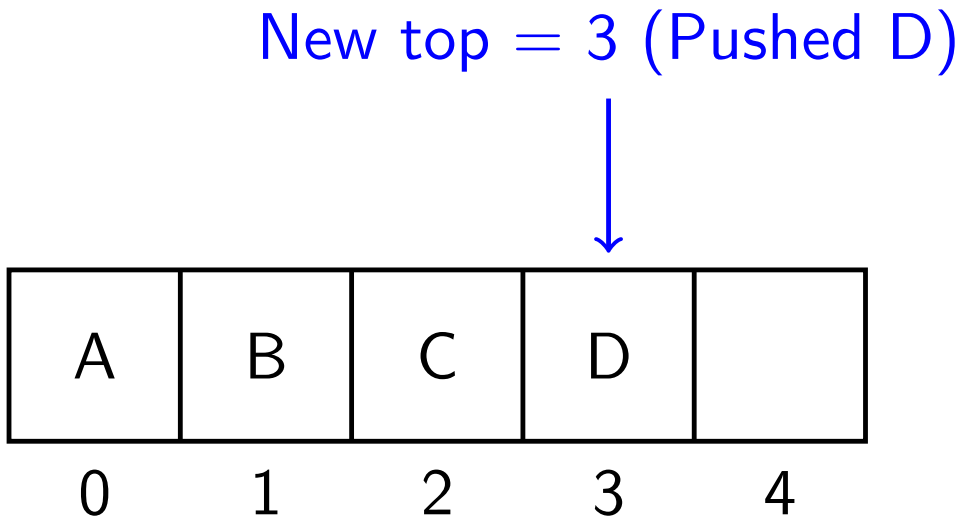
```
#define MAX 1000

struct Stack {
    int top;
    int arr[MAX];
};

void init(Stack* s) {
    s->top = -1;
}

bool isEmpty(Stack* s) {
    return s->top == -1;
}

bool isFull(Stack* s) {
    return s->top == MAX - 1;
```

The Push Operation: Implementation & Safety

```
bool push(Stack* s, int x) {
    if (isFull(s)) {
        printf("Stack-Overflow!-Cannot-push-%d.\n
        return false;
    }
    s->arr[++(s->top)] = x;
    return true;
}
```

- Overflow Condition: Attempting to insert an element when top equals MAX - 1 leads to a Stack Overflow.
- Pre-increment mechanism: Increment 'top' first, then write the element to arr[top] (order of operations is critical).
- Time Complexity: $O(1)$ constant time, as it requires only index calculation and direct memory writes.

Push Operation Visualized



- Step 1: The validity check verifies that the array has remaining capacity at index top + 1.
- Step 2: The value 'top' is incremented from 2 to 3.
- Step 3: Value 'D' is successfully stored at array index 3, updating the logical boundary of the stack.

The Pop Operation: Implementation & Underflow

```
int pop(Stack* s) {
    if (isEmpty(s)) {
        printf("Stack Underflow! Cannot pop.\n");
        return -1;
    }
    return s->arr[(s->top)--];
}
```

- Underflow Condition: Attempting to pop from an empty stack (top == -1) triggers a Stack Underflow error.
- Post-decrement mechanism: Retrieve the element at arr[top] first, then decrement 'top'.
- Logical vs. Physical deletion: The popped element remains in physical memory inside the array, but is logically removed since top is decremented.

The Peek / Top Operation

```
int peek(Stack* s) {
    if (isEmpty(s)) {
        printf("Stack is empty! Cannot peek.\n");
        return -1;
    }
    return s->arr[s->top];
}
```

- Unlike `pop()`, `peek()` returns the element at the top of the stack without modifying the stack's state.
- Used for inspect-and-decide algorithms, such as parsing operators in infix-to-postfix conversions.
- Maintains $O(1)$ time complexity and $O(1)$ space complexity.

Evaluation of Array-Based Stacks

C

ritical trade-offs between static arrays and linked representations for stack implementation.

- Advantage - Speed: Extremely fast operations since arrays provide contiguous physical memory access.
- Advantage - Space: Zero memory overhead for pointers (unlike linked lists which require next/prev pointers).
- Disadvantage - Fixed Size: Maximum capacity must be defined a priori. If underestimated, it causes Overflow; if overestimated, it wastes memory.

Dynamic Array Stack (Amortized Analysis)

S

caling capacity dynamically to handle unknown bounds while keeping high performance.

- When capacity is reached, allocate a new array of double size ($2 \times \text{MAX}$), copy elements, and free the old array.
- Push operation worst-case time complexity is $O(N)$ when resizing occurs.
- Amortized time complexity of Push remains $O(1)$ because copying is spread across many simple insertion steps.
- Space efficiency is dynamic, but temporary memory spike occurs during reallocation.

Data Structures (DI03000021)

GTU Diploma Engineering

Introduction to Stack (LIFO)

Stack (LIFO)

A stack is a linear data structure that follows the Last-In, First-Out (LIFO) principle. This means the last element added to the stack will be the first one to be removed.

- Linear collection of elements with a restricted access point called the 'top'.
- Elements are inserted and removed only from the top.
- Fundamental operations: push (insert), pop (delete), peek/top (look at top element without removing), isEmpty, isFull.
- Common applications include function call management (recursion), syntax parsing, undo/redo mechanisms, and backtracking algorithms.

Stack Abstract Data Type (ADT)

Stack ADT

An Abstract Data Type defines the logical properties of a data structure independently of its implementation. The Stack ADT specifies operations and their signatures.

- `push(element)`: Adds 'element' to the top of the stack. Returns nothing or success status.
- `pop()`: Removes and returns the top element. Triggers underflow error if the stack is empty.
- `peek()` or `top()`: Returns the element at the top without removing it. Triggers error if empty.
- `isEmpty()`: Returns true if the stack contains no elements, false otherwise.
- `isFull()`: Returns true if the stack is at its maximum capacity (relevant for fixed-size arrays).

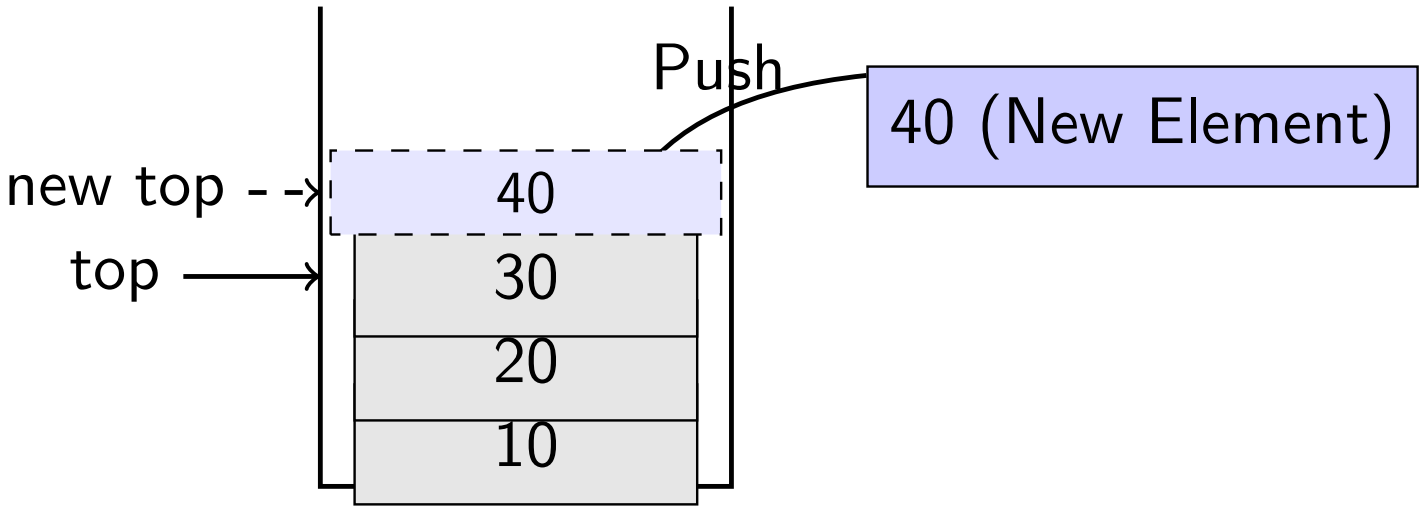
PUSH Operation & Algorithm

PUSH Operation

The PUSH operation inserts a new element onto the top of the stack. Before insertion, we must check if there is remaining capacity to prevent stack overflow.

- Algorithm for Array-based Stack:
  1. If top $\geq$ MAX_SIZE $-$ 1, raise Overflow error.
  2. Increment top: top $=$ top $+$ 1.
  3. Insert element: stack[top] $=$ element.
- Time Complexity of Push is $O(1)$ as it requires constant time to increment the index and write to memory.
- Stack Overflow occurs when an insertion is attempted on a stack that has reached its maximum memory allocation.
- Space Complexity of a single Push operation is $O(1)$ auxiliary space.

Visualizing PUSH Operation



- Initially, 'top' points to index 2 (value 30).
- The new element (40) is pushed onto the stack.
- The 'top' pointer/index increments to 3, and the value 40 is placed at stack[3].
- The stack grows upwards toward higher memory addresses.

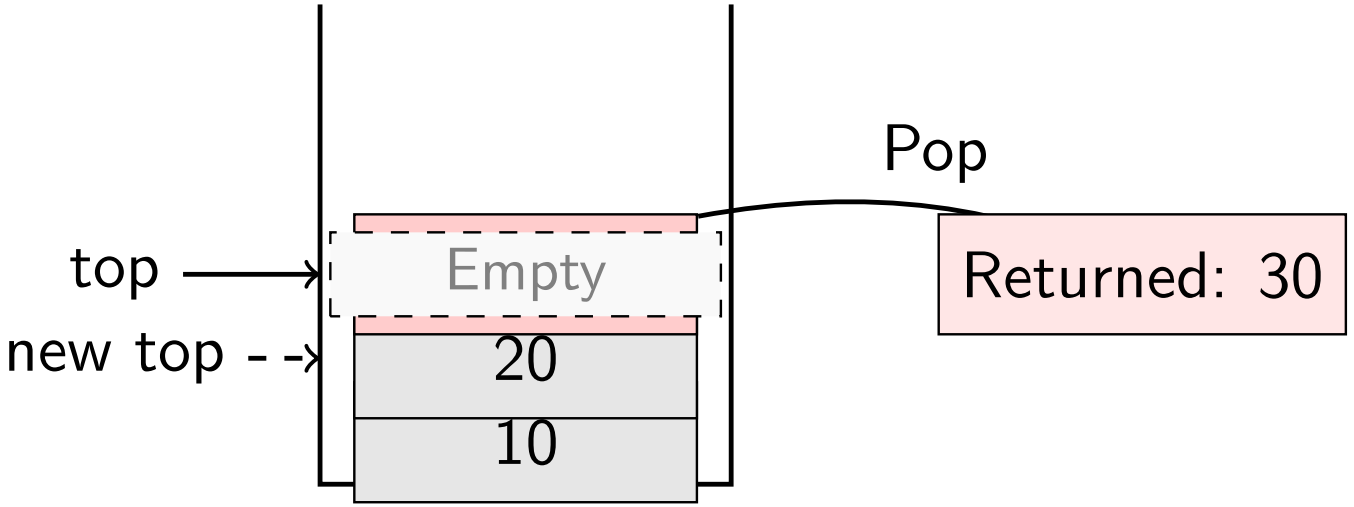
POP Operation & Algorithm

POP Operation

The POP operation removes and returns the top element of the stack. Before removing, we must verify that the stack is not empty to avoid stack underflow.

- Algorithm for Array-based Stack:
  1. If top < 0, raise Underflow error.
  2. Fetch value: element $=$ stack[top].
  3. Decrement top: top $=$ top $- 1$.
  4. Return element.
- Time Complexity of Pop is $O(1)$ as it involves a direct indexing retrieval and pointer decrement.
- Stack Underflow occurs when a pop (or peek) is called on an empty stack (top $==-1$ or NULL).
- In many languages, popped memory slots are cleared (garbage collected or set to null/0) to prevent memory leaks.

Visualizing POP Operation



- Initially, 'top' points to the topmost valid element (30) at index 2.
- The pop operation extracts this value (30) and returns it to the caller.
- The 'top' pointer is decremented to point to index 1 (value 20).
- The logical boundary of the stack shrinks down, ignoring the dereferenced element.

Array Implementation of Stack

Array Implementation

Stacks are commonly implemented using contiguous arrays. This approach offers speed and simplicity but suffers from a fixed capacity limit.

- A fixed-size array `stack[MAX_SIZE]` holds the elements, and an integer variable `top` tracks the current index.
- Initialization: Set top $= -1$, indicating the stack is empty.
- Pros: Direct $O(1)$ memory access, zero pointer overhead, excellent cache locality.
- Cons: Static capacity limits flexibility, resizing requires $O(n)$ array allocation and copy operations.

Linked List Implementation of Stack

Linked List Implementation

A dynamic stack uses a singly linked list where elements (nodes) are allocated dynamically in heap memory. The 'top' pointer corresponds to the head node of the list.

- Each node contains two fields: `data` and a pointer/reference `next` pointing to the node below it.
- Push Algorithm: 1. Allocate new node. 2. Set node.next = top. 3. Update top = node.
- Pop Algorithm: 1. If top is NULL, Underflow. 2. temp = top. 3. top = top.next. 4. Free temp, return data.
- Pros: Dynamic resizing, grows and shrinks according to usage, no pre-allocated overflow limit (except heap limit).
- Cons: Extra memory overhead for 'next' pointers (4 or 8 bytes per node), dynamic allocation overhead, potential cache misses.

Comparison & Lecture Summary

Summary

Stacks are crucial for managing LIFO workflows. Choosing between array and linked list depends on the predictability of the stack size and performance constraints.

- Time Complexity: Both Array and Linked List implementations achieve $O(1)$ for push, pop, and peek.
- Space Complexity: Array allocates fixed $O(\text{MAX\_SIZE})$ up front. Linked list uses $O(n)$ dynamically but has pointer overhead.
- Underflow: Occurs on pop when top $== -1$ (array) or top == NULL (linked list).
- Overflow: Occurs on push when top $==$ MAX_SIZE $- 1$ (array); impossible in linked list unless physical memory is exhausted.

Data Structures (DI03000021)

GTU Diploma Engineering

Stack Abstract Data Type & Memory Model

A

Stack is a LIFO (Last-In-First-Out) linear data structure. All insertion (push) and deletion (pop) operations occur at the top of the stack.

- Fundamental stack operations: push(x), pop(), peek()/top(), and isEmpty() — all executing in $O(1)$ time complexity.
- Memory Call Stack: Manages function call execution, local variables, parameter passing, and return addresses via Stack Frames.
- Recursion mechanism: System implicitly pushes activation records onto the call stack for each recursive call, popping them upon return.

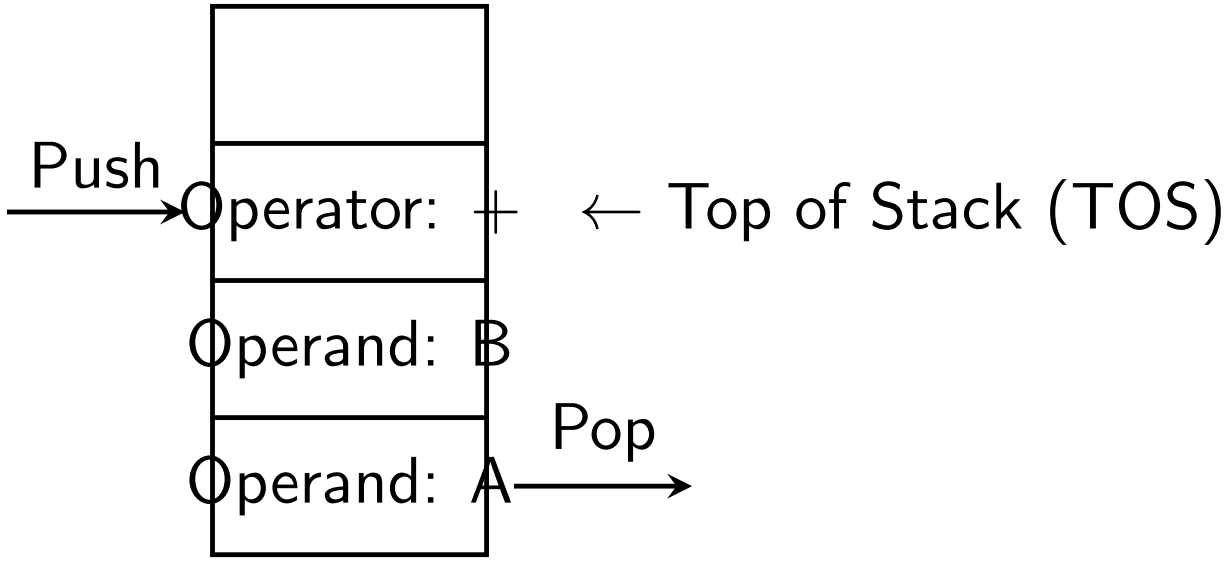
Expression Representations: Infix, Postfix, Prefix

E

xpressions are written in three notations depending on operator placement: Infix (operand-operator-operand), Postfix (operand-operand-operator), and Prefix (operator-operand-operand).

- Infix Notation (e.g., A + B): Standard representation for humans, but requires operator precedence, associativity rules, and parentheses to resolve ambiguity.
- Postfix Notation (e.g., A B +): Also known as Reverse Polish Notation (RPN). It is parenthesis-free and parsed linearly from left to right by computers.
- Prefix Notation (e.g., + A B): Also known as Polish Notation. Parenthesis-free, parsed from right to left.

Visualizing Stack Operations



- The diagram shows the LIFO nature of a stack holding intermediate tokens during expression parsing.
- New items are pushed onto the Top of Stack (TOS), incrementing the stack pointer.
- Popping retrieves the most recently added item, decrementing the stack pointer.

Algorithm: Infix to Postfix Conversion

A

lgorithm to convert Infix to Postfix using an operator stack to preserve precedence rules: Scan the infix expression from left to right, processing each token.

- Rule 1: If the scanned token is an operand, output it immediately.
- Rule 2: If the token is '(', push it onto the operator stack.
- Rule 3: If the token is ')', pop and output from the stack until '(' is encountered. Pop '(' from the stack but do not output it.
- Rule 4: If an operator is scanned, pop operators of greater or equal precedence from the stack and output them, then push the scanned operator.
- Rule 5: After reaching the end of the expression, pop and output all remaining operators from the stack.

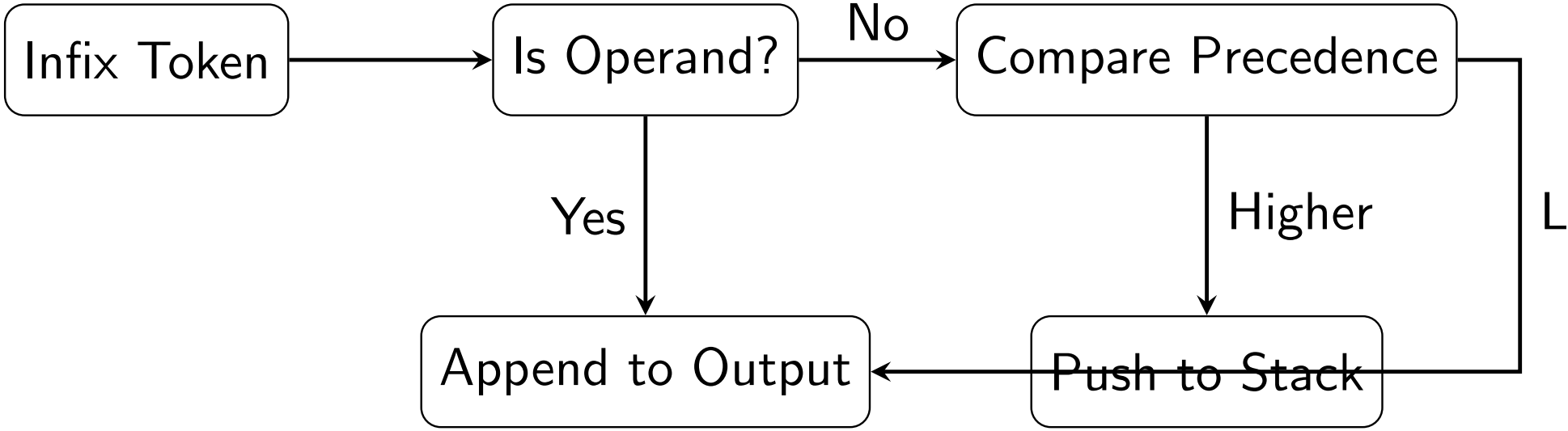
Tracing Infix to Postfix: A + B * C - D

E

xecution trace of the conversion algorithm on the expression 'A + B * C - D' to demonstrate how precedence rules are enforced.

- Tokens 'A', 'B', 'C', 'D' are operands, so they are directly appended to the output buffer.
- Operator '+' is pushed to stack. Operator '*' has higher precedence than '+', so it is pushed on top of '+': Stack is now [+, *].
- Operator '-' has equal precedence to '+' and lower than '*'. The '*' and '+' are popped and output. '-' is then pushed: Stack is [-].
- Final output sequence: 'A B C * + D -'. Time complexity of conversion is $O(N)$ where N is expression length.

Decision Flow for Infix to Postfix



- Flowchart illustrates the branching logic executed for each token in the input string.
- Operands bypass the stack and go directly to output to maintain operand order.
- Stack comparisons determine if operators are deferred (pushed) or evaluated (popped).

Evaluating Postfix Expressions

A

lgorithm to evaluate postfix expressions using an operand stack. Scanning is performed linearly from left to right.

- Rule 1: If the scanned token is an operand, push its value onto the operand stack.
- Rule 2: If the token is an operator, pop the top two operands from the stack.
- Order of operands: First popped is the right operand (op2), second popped is the left operand (op1).
- Rule 3: Evaluate (op1 operator op2) and push the resulting value back onto the stack.
- At the end of evaluation, the stack contains a single element representing the final expression result. Time complexity: $O(N)$.

Syntax Parsing: Parentheses Matching

P

arentheses matching is a fundamental task in compiler design to ensure nested groupings are structurally balanced.

- Algorithm scans expression: Open brackets '(', '[', '{' are pushed onto a character stack.
- When a closing bracket ')', ']', '}' is encountered, the stack's top element is popped and verified to match.
- Error cases: Empty stack when closing bracket is scanned (extra close), or unmatched bracket pair (e.g. '(' with ']').
- If stack is non-empty after scanning the entire string, it indicates unclosed opening brackets.
- This algorithm runs in $O(N)$ time and requires $O(N)$ auxiliary space.

Summary of Stack Applications

S

tacks serve as the primary engine for parsing, processing hierarchical structures, and tracking execution states.

- Algorithm optimization: Expression conversion and evaluation run in linear $O(N)$ time, optimal for interpreter pipelines.
- Call Stack abstraction: Hardware and OS levels use stacks for control flow redirection, recursion, and interrupt handling.
- Design trade-offs: Linked list stacks provide dynamic sizing (preventing stack overflow) but incur pointer overhead; array-based stacks are cache-friendly but have fixed size limit.

Data Structures (DI03000021)

GTU Diploma Engineering

Introduction to Expression Notations

A

rithmetic expressions are written in three notations: Infix, Prefix (Polish), and Postfix (Reverse Polish). The placement of operator relative to operands defines the notation.

- **Infix Notation:** Operators reside between operands (e.g., $A + B$). Requires precedence rules (PEMDAS) and brackets for disambiguation.
- **Prefix Notation:** Operators precede operands (e.g., $+AB$). Allows expressions to be parsed unambiguously without parenthesis.
- **Postfix Notation:** Operators follow operands (e.g., $AB+$). Ideal for stack-based execution because operators appear in order of evaluation.

Algorithm for Postfix Evaluation using Stack

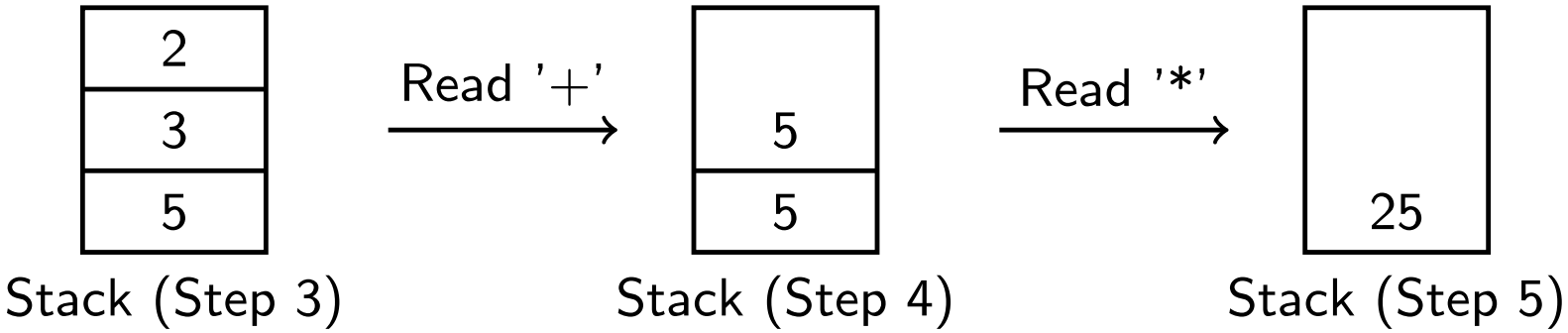
A

stack is used to evaluate postfix expressions in a single pass. The algorithm runs in $O(N)$ time and requires $O(M)$ memory where M is stack height.

- Scan the postfix expression from left to right token by token.
- If the token is an operand, push it onto the stack.
- If the token is an operator, pop the top two operands from the stack.
- Evaluate: apply the operator to the popped operands. Order is critical: first pop is right operand, second pop is left operand.
- Push the evaluation result back onto the stack and repeat.

Postfix Evaluation: Step-by-Step Stack Trace

Evaluating Postfix: 5 3 2 + *



- Input postfix expression: 5 3 2 + *
- Operands 5, 3, and 2 are pushed sequentially (Stack Step 3).
- Operator '+' pops 2 (right operand) and 3 (left operand). Evaluates $3 + 2 = 5$ and pushes it (Stack Step 4).
- Operator '*' pops 5 (right operand) and 5 (left operand). Evaluates $5 \times 5 = 25$ and pushes it (Stack Step 5).

Infix to Postfix Conversion: Shunting-Yard Algorithm

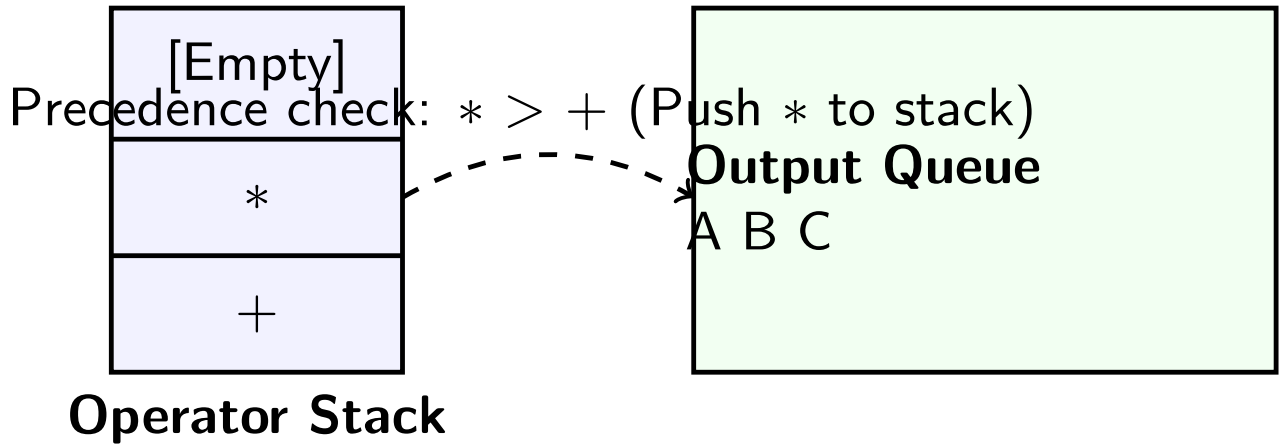
D

ijkstra's Shunting-Yard algorithm is an operator-precedence parser that converts an infix expression to a postfix expression using an operator stack.

- Scan the infix expression from left to right.
- If a token is an operand, output it directly.
- If an operator is encountered, pop from stack to output while stack operator has higher/equal precedence. Then push new operator.
- Left parenthesis '(' is pushed to stack. Right parenthesis ')' pops to output until '(' is popped and discarded.

Shunting-Yard Precedence Evaluation

Infix Input: $A + B * C$



- Infix input 'A + B * C' is parsed.
- Operands A, B, and C are passed directly to the output queue.
- Operator '+' is pushed first. When '*' is read, it is compared with '+' on stack. Since precedence('*') > precedence('+'), it is pushed without popping.
- At the end of string, stack contents are popped sequentially to output queue: A B C * +

Recursive Functions and the System Call Stack

R

ecursion is a programming technique where a function solves a problem by calling a smaller instance of itself. The call stack tracks active execution frames.

- **Stack Frame (Activation Record):** Created on the call stack for each active call. Contains parameters, local variables, and return address.
- **LIFO Execution:** The most recently invoked recursive call is evaluated first and popped before returning to the caller function.
- **Base Case:** Terminating condition that prevents infinite recursion. Must be reachable to avoid System Stack Overflow.
- **Recursive Step:** Reduces problem size and guarantees progression towards the base case.

Recursive Case Study: Factorial Function

F

actorial of n ($n!$) is mathematically defined as: $n! = n \times (n-1)!$ for $n > 0$, with base case $0! = 1$.

- **C++ Code:** `int factorial(int n) { if (n <= 1) return 1; return n * factorial(n-1); }`
- **Recursive Call Trace:** `factorial(3)` spawns `factorial(2)`, which spawns `factorial(1)`.
- **Time Complexity:** $O(N)$ as there are exactly N recursive function invocations.
- **Space Complexity:** $O(N)$ auxiliary space on the call stack to store the N active stack frames.

Stack Frame Lifecycles during Factorial(3)

A

detailed lookup of stack frames as factorial(3) is executed by the CPU.

- **1. factorial(3) called:** Stack Frame 1 allocated. Needs value of `factorial(2)` to compute $3 \times$ factorial(2).
- **2. factorial(2) called:** Stack Frame 2 allocated. Needs value of `factorial(1)` to compute $2 \times$ factorial(1).
- **3. factorial(1) called:** Stack Frame 3 allocated. Hits base case ($n \leq 1$), returns 1, Frame 3 popped.
- **4. factorial(2) resumes:** receives 1, evaluates $2 \times 1 = 2$, returns 2, Frame 2 popped.
- **5. factorial(3) resumes:** receives 2, evaluates $3 \times 2 = 6$, returns 6, Frame 1 popped. Final result is 6.

Recursion vs Iteration: Computational Trade-offs

C

omparison of Recursive and Iterative approaches for solving repetitive problems.

- **Space Overhead:** Iteration uses $O(1)$ space (keeps modifying single state variables). Recursion requires $O(N)$ space due to call stack frames.
- **Execution Speed:** Iteration is typically faster as it avoids function call overhead (saving registers, jumping, initializing stack frames).
- **Readability:** Recursion is highly expressive and matches mathematical definitions, making complex algorithms (e.g. Quicksort, tree traversals) cleaner.
- **Tail-Call Optimization (TCO):** Modern compilers can optimize recursive calls in tail-position to reuse the current stack frame, reducing space to $O(1)$.

Data Structures (DI03000021)

GTU Diploma Engineering

Mathematical Definition of GCD

T

he Greatest Common Divisor (GCD) of two non-zero integers a and b is the largest positive integer d such that d divides both a and b. Mathematically, it is denoted as $\text{GCD}(a, b)$.

- Formal Definition:
  $\text{GCD}(a, b) = \max\{d \in \mathbb{Z}^+ : d \mid a \text{ and } d \mid b\}$
- GCD is essential in modern cryptography (e.g., RSA key generation relies on coprime numbers where GCD is 1).
- Important Property: $\text{GCD}(a, b) = \text{GCD}(b, a \bmod b)$ for any non-zero integers.
- Base Case: $\text{GCD}(a, 0) = |a|$, since every integer divides 0.

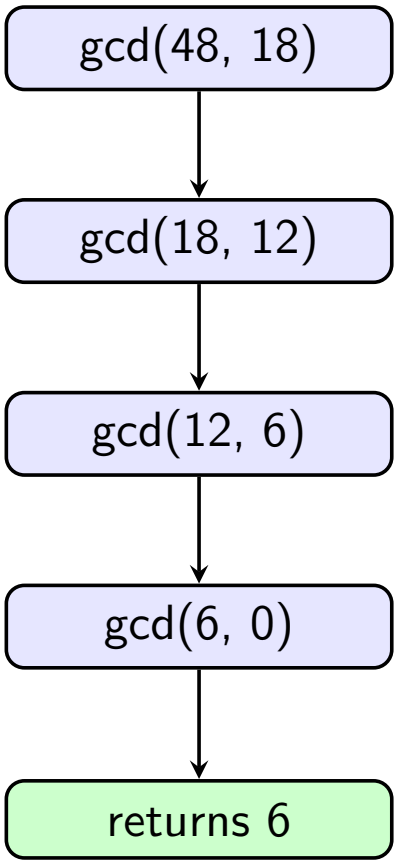
The Euclidean Algorithm: Iterative & Recursive

T

he Euclidean Algorithm is an extremely efficient method for computing the GCD. Instead of finding prime factors, it repeatedly applies the division algorithm.

- Recursive Approach: `gcd(a, b) = if b == 0 then a else gcd(b, a % b)`
- Iterative Approach: `while (b != 0) { int t = b; b = a % b; a = t; } return a;`
- Time Complexity: $\mathcal{O}(\log(\min(a, b)))$ because in each step, the modulo operation reduces the larger number by at least half.
- Space Complexity: $\mathcal{O}(\log(\min(a, b)))$ for recursive recursion frames due to call stack, or $\mathcal{O}(1)$ auxiliary space for iterative version.

Euclidean Algorithm Recursion Tree for GCD(48, 18)



- Step 1: `gcd(48, 18)` calls `gcd(18, 48 % 18)` = `gcd(18, 12)`.
- Step 2: `gcd(18, 12)` calls `gcd(12, 18 % 12)` = `gcd(12, 6)`.
- Step 3: `gcd(12, 6)` calls `gcd(6, 12 % 6)` = `gcd(6, 0)`.
- Step 4: `gcd(6, 0)` reaches the base case since $b = 0$, returning $a = 6$.

Fibonacci Series and Lame's Theorem

T

he Fibonacci sequence $F(n)$ is defined as $F(n) = F(n-1) + F(n-2)$ with $F(0) = 0$ and $F(1) = 1$. Consecutive Fibonacci numbers represent the worst-case inputs for the Euclidean algorithm.

- Worst-case Input: Computing GCD$(F(n), F(n-1))$ requires exactly $n-2$ recursive calls.
- Lame's Theorem: If the Euclidean algorithm takes k steps, then the smaller input b satisfies $b \geq F(k+2)$.
- Number of steps is bounded by 5 times the number of decimal digits of the smaller number: $k \leq 5 \times d$.
- This proves the logarithmic time complexity of the Euclidean algorithm: $\mathcal{O}(\log b)$.

Generating Fibonacci Series using a Queue

A

Queue (First-In-First-Out structure) can generate the Fibonacci series iteratively using constant auxiliary space and $\mathcal{O}(n)$ time complexity.

- Initialization: Enqueue 0, then enqueue 1 to the Queue.
- Step 1: Dequeue the front element (a). This is the current Fibonacci number.
- Step 2: Peek at the next front element (b) without removing it.
- Step 3: Enqueue the sum ($a + b$) at the rear of the Queue.
- Step 4: Output the value 'a', and repeat the process to generate the next numbers.

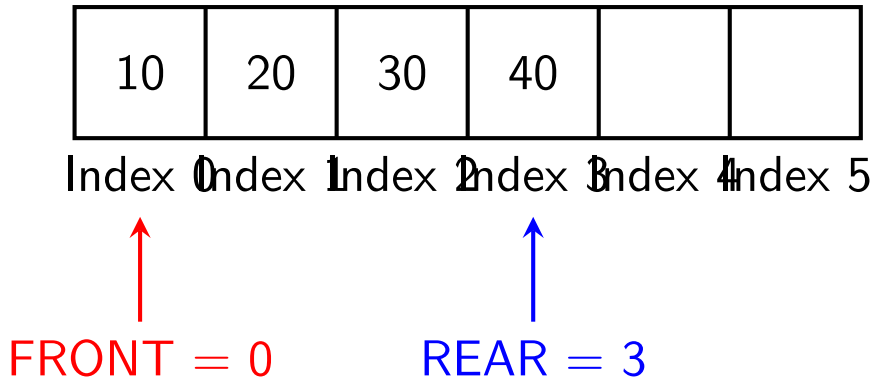
Introduction to Queue Data Structure

A

Queue is a linear data structure that operates under the First-In-First-Out (FIFO) paradigm. Elements are inserted at the back and removed from the front.

- Key operations: Enqueue (adds an item to the rear) and Dequeue (removes an item from the front).
- Auxiliary operations: Front/Peek (returns front element), IsEmpty, IsFull.
- Pointers: `'FRONT'` tracks the deletion index; `'REAR'` tracks the insertion index.
- Applications: Job scheduling in operating systems, printer queues, handling asynchronous data transfers.

Array Representation of a Linear Queue



- In a linear queue of size `MAX`, `FRONT` and `REAR` are initialized to -1.
- Enqueue: If queue is not full, increment `REAR` and store the element at `Array[REAR]`.
- Dequeue: If queue is not empty, retrieve `Array[FRONT]` and increment `FRONT`.
- Condition: Queue is Empty when `FRONT` $==-1$ or `FRONT` $>$ `REAR`. Queue is Full when `REAR` $==$ `MAX` -1.

Queue Boundary Operations & Code Logic

B

elow is the structural algorithm and time complexities for Enqueue and Dequeue operations on a linear queue.

- Enqueue Implementation: `void enqueue(int val) { if(rear == MAX-1) error('Overflow'); if(front == -1) front = 0; rear++; arr[rear] = val; }`
- Dequeue Implementation: `int dequeue() { if(front == -1 || front > rear) error('Underflow'); int val = arr[front]; front++; return val; }`
- Time Complexity: Enqueue $\mathcal{O}(1)$, Dequeue $\mathcal{O}(1)$, Peek $\mathcal{O}(1)$.
- Space Complexity: $\mathcal{O}(\texttt{MAX})$ array storage, where MAX is static pre-allocated memory size.

Limitations of Linear Queues & Solutions

L

inear queues implemented using static arrays suffer from a critical issue: the waste of memory spaces due to pointer progression.

- The Memory Wastage Problem: Once `REAR` reaches $\texttt{MAX} - 1$, no more items can be enqueued, even if elements were dequeued and the front indices are free.
- Inefficient Solution: Shift all elements to the left on every dequeue operation. This makes Dequeue $\mathcal{O}(N)$ instead of $\mathcal{O}(1)$.
- Efficient Solution: Circular Queue. The pointers wrap around using modulo arithmetic: $\texttt{front} = (\texttt{front} + 1) \bmod \texttt{MAX}$, $\texttt{rear} = (\texttt{rear} + 1) \bmod \texttt{MAX}$.
- Circular Queue Full Condition: $(\texttt{rear} + 1) \bmod \texttt{MAX} == \texttt{front}$

Data Structures (DI03000021)

GTU Diploma Engineering

Introduction to Queue ADT

A

Queue is a First-In, First-Out (FIFO) linear data structure where elements are inserted at the back (rear) and removed from the front.

- FIFO Principle: The element that enters first is processed and removed first.
- Key Pointers: 'front' tracks the index of the next deletion; 'rear' tracks the index of the latest insertion.
- Primary Operations: Enqueue (Insert element at rear) and Dequeue (Remove element from front).
- Auxiliary Operations: `isEmpty()` to check for underflow, `isFull()` to check for overflow, and `peek()` to view front without removal.

Implementing Queues: Array vs. Linked List

Q

ueues can be implemented using contiguous arrays (fixed size) or dynamically allocated linked lists (variable size).

- Array Representation: Fast access time but has static capacity limits. Requires sliding elements or circular index arithmetic to manage space.
- Linked List Representation: Dynamic size adjustment, no overflow risk (unless heap memory is exhausted), but incurs pointer overhead.
- Time Complexity: Both operations run in $O(1)$ time complexity in optimized implementations.
- False Overflow: Linear array implementations suffer from unusable space at the front as the queue slides forward.

Array Implementation of Enqueue

T

he enqueue operation inserts an element at the rear of the queue after validating that the structure has space available.

- Overflow Condition: In a standard linear array queue of capacity `MAX`, overflow occurs if `rear == MAX - 1`.
- Initialization: Empty queue state is represented by `front = -1` and `rear = -1`.
- Logic: If `front` is -1, set both `front` and `rear` to 0. Otherwise, increment `rear` by 1 and store the value at `queue[rear]`.
- Complexity: $O(1)$ runtime because it involves basic index lookup and pointer manipulation.

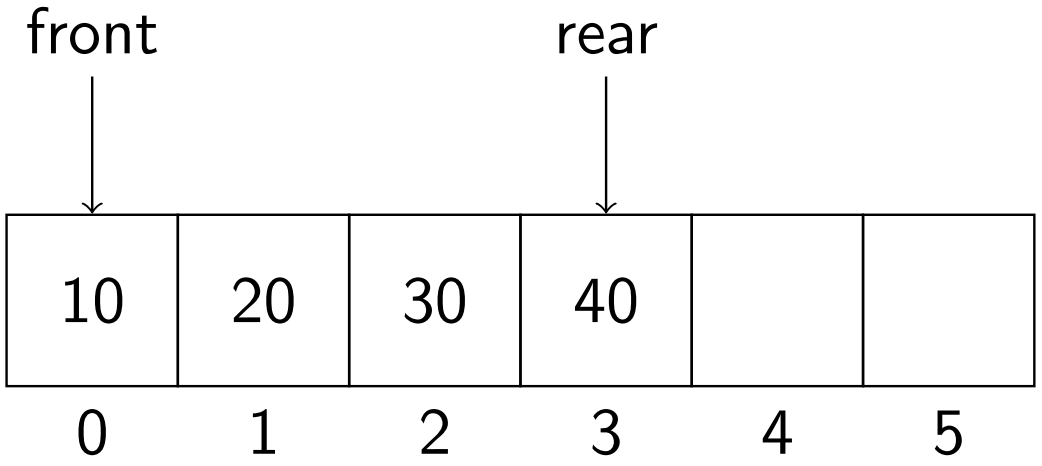
Array Implementation of Dequeue

T

he dequeue operation removes and returns the element at the front of the queue, updating pointer offsets appropriately.

- Underflow Condition: Check if `front == -1` or `front > rear`. If true, operation fails with an underflow error.
- Logic: Retrieve the element at `queue[front]`. If `front` equals `rear` (queue becomes empty), reset `front` and `rear` to -1. Otherwise, increment `front` by 1.
- Memory Waste: Once `front` increments, the slot it leaves behind becomes unreachable, decreasing usable queue capacity.
- Complexity: $O(1)$ runtime as no element shifts are performed.

Diagram of a Linear Queue (Array-based)



- The diagram represents a linear array-based queue of capacity 6.
- Values 10, 20, 30, and 40 occupy indices 0 through 3 respectively.
- The 'front' pointer is positioned at index 0, indicating the next element to be dequeued.
- The 'rear' pointer points to index 3, which is the location of the most recently inserted item.

Solving False Overflow with Circular Queues

A

circular queue avoids the false overflow problem by conceptually wrapping the tail of the array back to the head.

- Modulo Arithmetic: The next index of any pointer `i` is computed using circular increments: `i = (i + 1) % MAX`.
- Circular Enqueue: Allowed if the slot next to rear is not front: `(rear + 1) % MAX != front`.
- Circular Dequeue: Increments front via: `front = (front + 1) % MAX`.
- Space Utilization: Allows empty spaces created by dequeues at the beginning of the array to be reused dynamically.

Circular Queue Operations

A

lgorithmic logic changes in circular queues to manage indexing wraps securely without data overwrites.

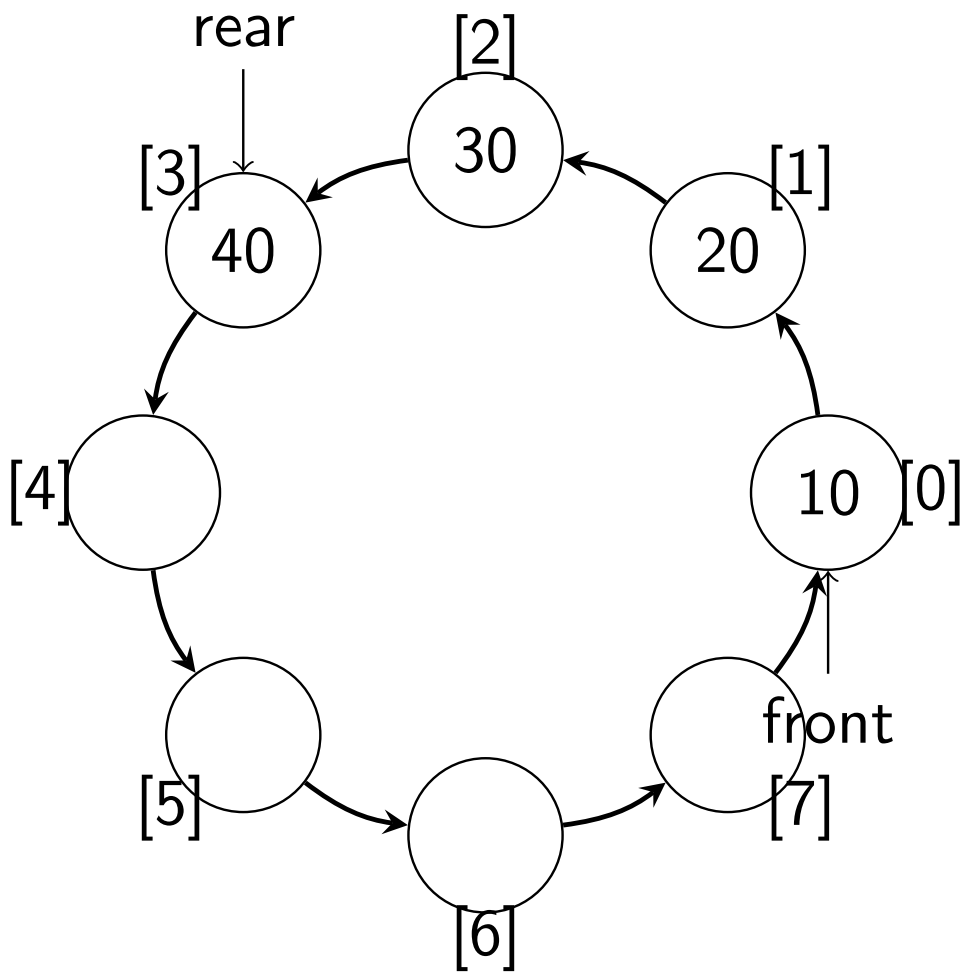
- Circular Overflow: Checked via expression: `((rear + 1) % MAX) == front`.
- Circular Enqueue Code:

```
if ((rear + 1) % MAX == front) return Overflow;
if (front == -1) front = rear = 0;
else rear = (rear + 1) % MAX;
queue[rear] = value;
```

- Circular Dequeue Code:

```
if (front == -1) return Underflow;
value = queue[front];
if (front == rear) front = rear = -1;
else front = (front + 1) % MAX;
```

Diagram of a Circular Queue



- Circular structure with indices 0 through 7. Arrows indicate the direction of logical flow.
- Elements 10, 20, 30, and 40 are enqueued sequentially.
- `front` is located at index 0 and `rear` points to index 3.
- The next enqueue operation will place an element at index 4

Linked List Implementation of Queue

A

linked queue dynamically allocates nodes, eliminating hard capacity boundaries and avoiding circular mapping complexities.

- Structure: Contains a 'front' pointer to the head node (deletion end) and a 'rear' pointer to the tail node (insertion end).
- Enqueue Logic: Create node. Set `node->next = NULL`. If `rear` is `NULL`, set `front = rear = node`. Else, `rear->next = node; rear = node`.
- Dequeue Logic: If `front` is `NULL`, return Underflow. `temp = front; front = front->next; if (front == NULL) rear = NULL; free(temp);`.
- Advantage: Dynamic capacity means no Overflow check is required, other than verifying heap allocation success.

Data Structures (DI03000021)

GTU Diploma Engineering

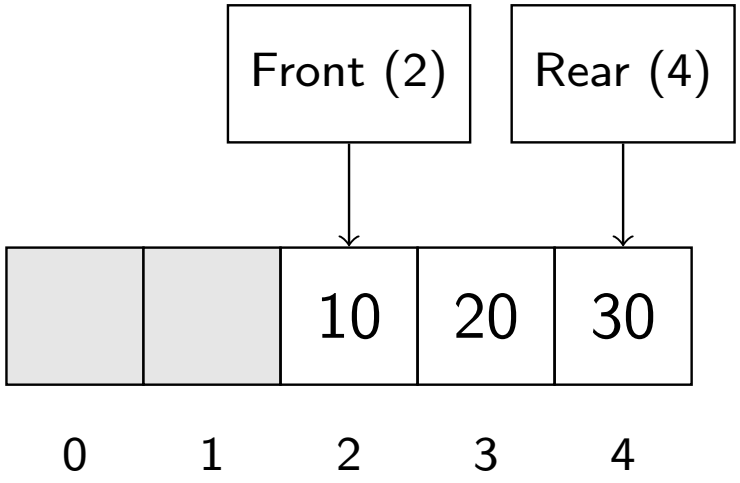
Introduction to Linear Queue

Linear Queue

A Linear Queue is a First-In-First-Out (FIFO) data structure where insertion happens at the Rear and deletion happens at the Front. Memory allocation is typically sequential (array-based).

- Defined by two pointers: Front (index of the first element) and Rear (index of the last element).
- Enqueue operation: Increments Rear and inserts element at queue[Rear]. Time complexity is $O(1)$.
- Dequeue operation: Increments Front and retrieves element from queue[Front]. Time complexity is $O(1)$.
- Queue Empty condition: Front == -1 (or Front ¿ Rear, depending on initialization).
- Queue Full condition: Rear == MAX - 1 (for 0-indexed arrays of size MAX).

Visualizing the Limitation of Linear Queue



- Elements at index 0 and 1 have been dequeued, leaving these memory slots empty.
- Rear has reached index 4 (MAX-1 where MAX = 5), signaling that the queue is full.
- Despite having 2 free slots at the front, any new enqueue attempt will result in a Queue Overflow error.
- This illustrates the memory wastage problem inherent in a simple linear queue implementation.

The False Overflow Problem

False Overflow Condition

The “False Overflow” or “False Full” condition occurs when a linear queue is blocked from accepting new elements even though free slots exist in the array.

- Caused by the unidirectional movement of Front and Rear pointers (always incrementing during operations).
- Once Rear reaches the last index (MAX - 1), it cannot go back to 0, even if Front ¿ 0.
- Attempting to insert a new element fails, triggering a false “Queue Overflow” exception.
- This results in poor memory utilization, as vacated space at the front of the queue cannot be reused.

Evaluating Inefficient Workarounds

Naive Solutions

To resolve memory wastage in linear queues, two naive solutions are often proposed, both introducing significant performance overheads.

- Workaround 1: Shifting elements. Shift all remaining elements to the left by one position upon every Dequeue. Time complexity of Dequeue degrades from $O(1)$ to $O(N)$.
- Workaround 2: Resetting pointers. Shift elements only when Rear == MAX - 1 and Front ¿ 0. This amortizes the cost but still causes periodic $O(N)$ pauses.
- Both methods violate the core FIFO design requirement: $O(1)$ insertion and deletion.
- A more elegant, mathematically sound solution is required to keep both operations at $O(1)$ without shifting.

Concept of Circular Queue

Circular Queue

A Circular Queue is a linear data structure in which the operations are performed based on FIFO principle and the last position is connected back to the first position to make a circle.

- Also known as a 'Ring Buffer'.
- Solves the False Overflow problem by allowing the Rear pointer to wrap around to index 0 when it reaches MAX - 1, provided space is available.
- Maintains constant $O(1)$ time complexity for both Enqueue and Dequeue operations without shifting elements.
- Pointers wrap around using modulo arithmetic: $\text{index} = (\text{index} + 1)\%\text{MAX}$.

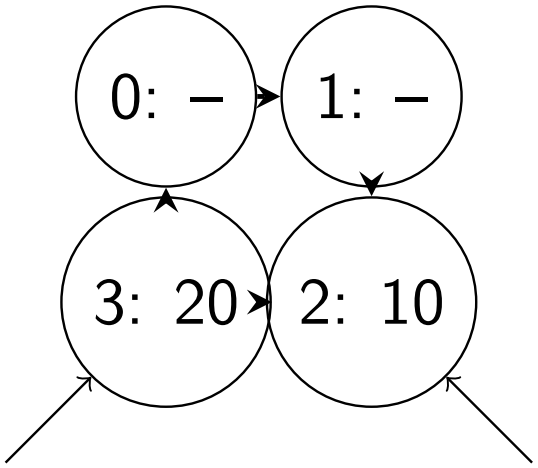
Circular Queue Arithmetic & Conditions

Modulo Arithmetic Operations

Circular Queue operations rely on modulo arithmetic to wrap pointers around the boundaries of the underlying array representation.

- Enqueue Logic: `Rear = (Rear + 1) % MAX; Queue[Rear] = Element;`
- Dequeue Logic: `Element = Queue[Front]; Front = (Front + 1) % MAX;`
- Queue Empty Condition: Front == -1 and Rear == -1 (or Front == Rear in some implementations).
- Queue Full Condition: `(Rear + 1) % MAX == Front`. This means the next position of Rear is Front.
- Number of elements in the queue: `(Rear - Front + MAX) % MAX + 1` (when not empty).

Visualizing the Circular Queue Structure



- The array is logically arranged in a circular configuration with size MAX = 4.
- Indices 0 and 1 are empty, while indices 2 and 3 contain the elements 10 and 20 respectively.
- Front pointer is at index 2 (referencing the oldest element, 10) and Rear is at index 3 (referencing the newest element, 20).
- The next enqueue operation will place an element at index 0

Circular Queue Implementation in C++

Implementation of basic operations using array-based circular queue representation.

- Enqueue Implementation:

```
void enqueue(int val) {
    if ((rear + 1) % MAX == front) return;
    if (front == -1) front = 0;
    rear = (rear + 1) % MAX;
    arr[rear] = val;
}
```

- Dequeue Implementation:

```
int dequeue() {
    if (front == -1) return -1;
    int val = arr[front];
    if (front == rear) front = rear = -1;
    else front = (front + 1) % MAX;
    return val;
```

Summary: Linear vs. Circular Queues

Summary Overview

A comparative summary highlighting the design trade-offs and operational efficiency between Linear and Circular Queues.

- **Memory Utilization:** Linear Queue is inefficient (leads to false overflow); Circular Queue is highly efficient (reuses empty slots).
- **Pointer Wrapping:** Linear Queue does not wrap pointers (increment only); Circular Queue wraps pointers using modulo arithmetic.
- **Complexity:** Both achieve $O(1)$ time complexity for Enqueue and Dequeue, but Circular Queue maintains this without needing $O(N)$ array shifts.
- **Implementation Overhead:** Circular Queue requires slightly more complex index calculations but significantly saves memory.

Data Structures (DI03000021)

GTU Diploma Engineering

The Queue ADT and FIFO Paradigm

A

Queue is a linear data structure that follows the First-In, First-Out (FIFO) principle, where elements are inserted at the Rear (Enqueue) and removed from the Front (Dequeue).

- Enqueue operation: Inserts an element at the rear/tail of the queue. Time complexity: O(1).
- Dequeue operation: Removes an element from the front/head of the queue. Time complexity: O(1).
- Peek/Front operation: Inspects the front element without removal. Time complexity: O(1).
- Underflow occurs when attempting to dequeue from an empty queue; Overflow occurs when enqueuing to a full queue (in static array implementations).

CPU Scheduling: Round Robin & Multi-level Queues

I

n Operating Systems, queues manage the execution of processes sharing a single CPU resource.

- Round Robin (RR) scheduling uses a FIFO queue to hold ready processes. The CPU scheduler enqueues new processes, dequeues the head process, runs it for a time quantum, and if incomplete, re-enqueues it.
- Ensures fairness: No process is starved of CPU time, achieving time-sharing.
- Multilevel Queue Scheduling separates processes into distinct queues based on priority (e.g., system processes, interactive processes, background batch processes).
- Thread scheduling in runtime environments (like JVM or Go scheduler) uses run queues to distribute workloads among threads

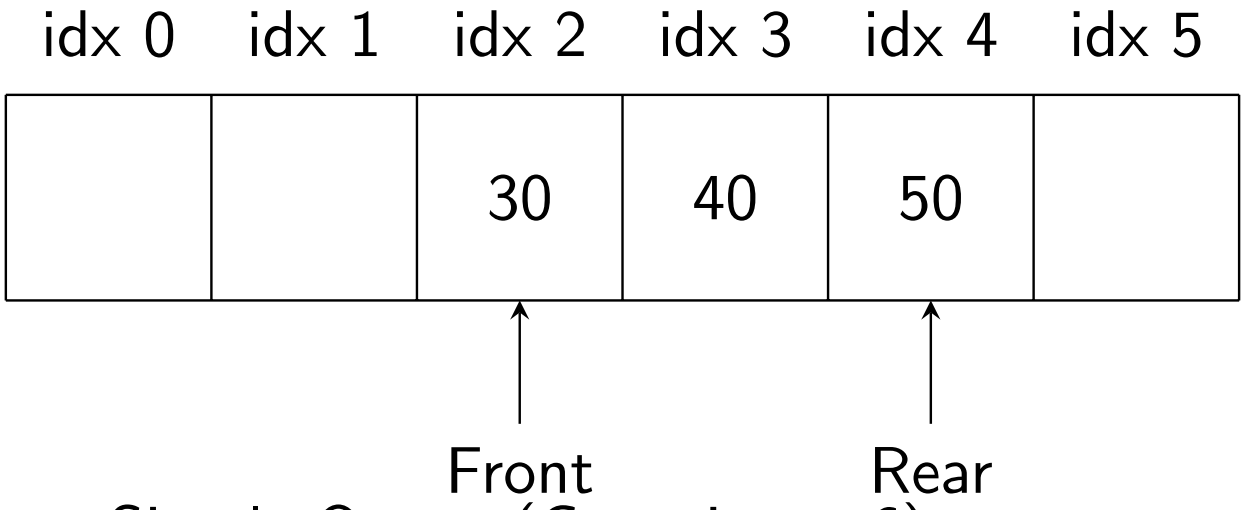
I/O Buffering and Networking

Q

ueues resolve speed mismatches between data producers (fast CPUs) and consumers (slow I/O devices or networks).

- I/O Buffering: Disk writes and print jobs are enqueued to a spooler (e.g., print spooler) so the CPU doesn't block waiting for slow electromechanical or peripheral devices.
- Network Router Buffering: Packets arriving at network interfaces are stored in queues (FIFO packet buffers) when the transmission rate is lower than the arrival rate, preventing immediate packet loss.
- Congestion control algorithms (e.g., RED - Random Early Detection) monitor router queue sizes to manage packet drop behavior.
- Keyboard Buffer: Keypress interrupts write character codes into a FIFO queue, which the operating system polls and

Memory Inefficiency in Simple Array Queues



Simple Queue (Capacity = 6)
Indices 0 and 1 are empty but unusable because Rear = 4.

- Limitation of Simple Array Queue: After multiple enqueue and dequeue operations, the Front pointer moves forward, leaving empty slots at the beginning of the array.
- When Rear reaches the last index (size - 1), the queue is declared 'Full' (Overflow), even if indices 0 to Front-1 are vacant.
- Time Complexity of shifting elements left to reclaim space on

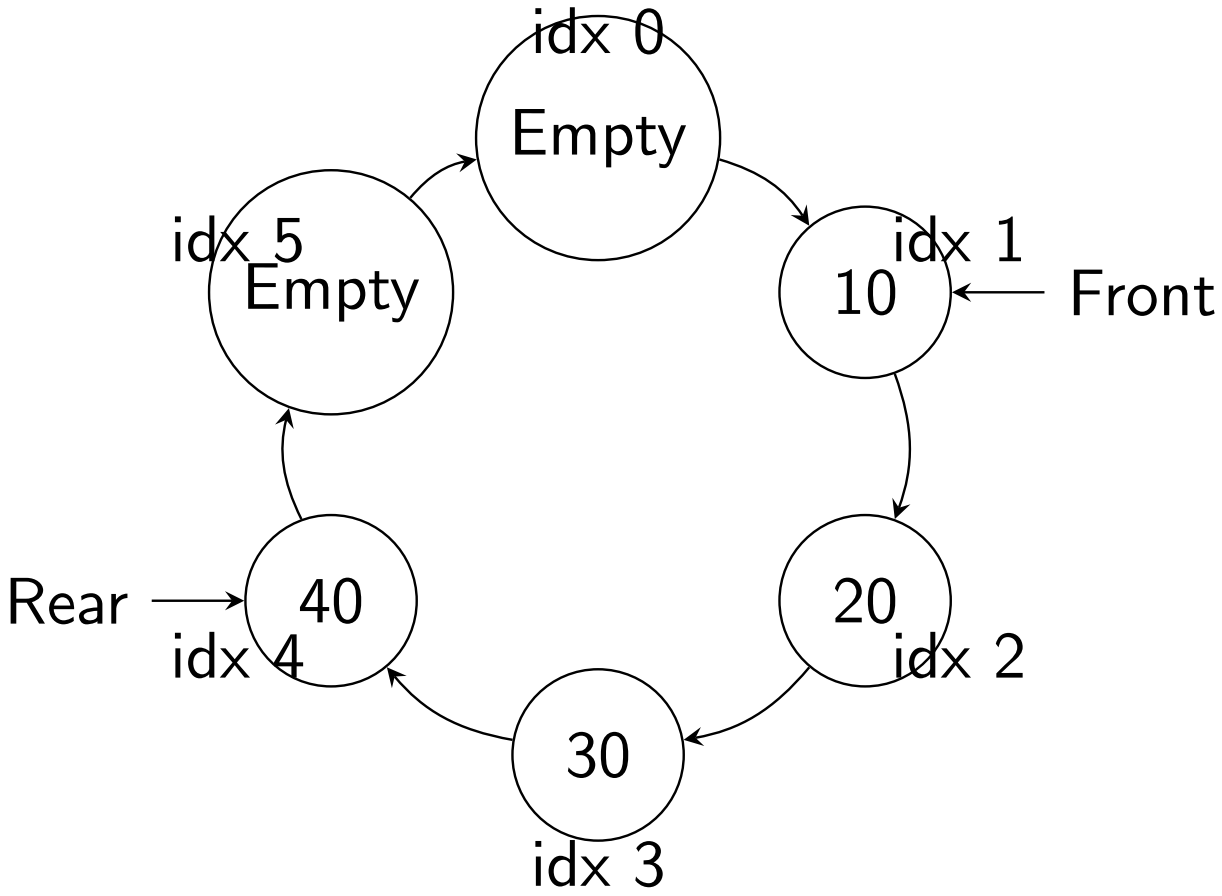
Circular Queue & Modulo Arithmetic

A

Circular Queue is a linear data structure in which operations are performed based on a FIFO principle and the last position is connected back to the first position to make a circle.

- Index Wrapping: Pointers increment using modulo size: `index = (index + 1) % capacity`.
- Enqueue Position: `rear = (rear + 1) % capacity`, then `array[rear] = element`.
- Dequeue Position: `front = (front + 1) % capacity`, after retrieving `array[front]`.
- Avoids shifting elements, maintaining strict O(1) time complexity for both enqueue and dequeue operations.
- Implementation uses a fixed-size array but behaves logically as a ring buffer.

Visualizing the Circular Queue (Ring Buffer)



- The circular configuration allows the rear pointer to wrap around to index 0 after reaching the last index (5 in this diagram).
- Empty spaces (represented by 'Empty' at index 0 and 5) can be reused without shifting remaining elements.

Comparison: Simple Queue vs. Circular Queue

K

ey design, structural, and performance differences between simple array queues and circular ring buffers.

- Memory Utilization: Simple queues suffer from memory leakage/wastage when elements are dequeued. Circular queues utilize all slots dynamically.
- Time Complexity of Dequeue: Simple queue requires either shifting elements (O(N)) or wasting space. Circular queue achieves O(1) dequeue and O(1) space reuse.
- Pointer Traversal: Simple queue pointers increment linearly (`rear++`). Circular queue pointers wrap around using modulo division (`(rear + 1) % size`).
- Full Condition: Simple queue: `rear == size - 1`. Circular queue: `(rear + 1) % size == front`.
- Empty Condition: Simple queue: `front == -1` or `front >`

Algorithmic Applications: Breadth-First Search (BFS)

Q

ueues are the fundamental data structure supporting Breadth-First Search (BFS) in trees and graphs, and in level-order traversal.

- BFS exploration: Explores all neighbors at the current depth before moving to nodes at the next depth level.
- Algorithm: Enqueue root, then enter a loop: Dequeue vertex `u`, process `u`, and enqueue all unvisited neighbors of `u`.
- FIFO property ensures that nodes closer to the starting vertex are processed before nodes that are further away.
- Used in finding the shortest path in unweighted graphs, network broadcasting, and web crawling algorithms.

Summary and Key Takeaways

Q

ueues serve as vital infrastructure across algorithms, OS kernel design, and networking.

- Queue variants: Deque (Double-ended queue) allows insertions/deletions at both ends; Priority Queue removes elements based on priority rather than arrival order.
- Simple queues implemented via arrays are inefficient due to unusable space at the front.
- Circular queues (ring buffers) are highly efficient for streaming data (audio/video buffers) and resource pooling.
- Always analyze the bounds: static arrays have fixed capacity; linked-list queues avoid overflow but introduce pointer overhead.

Data Structures (DI03000021)

GTU Diploma Engineering

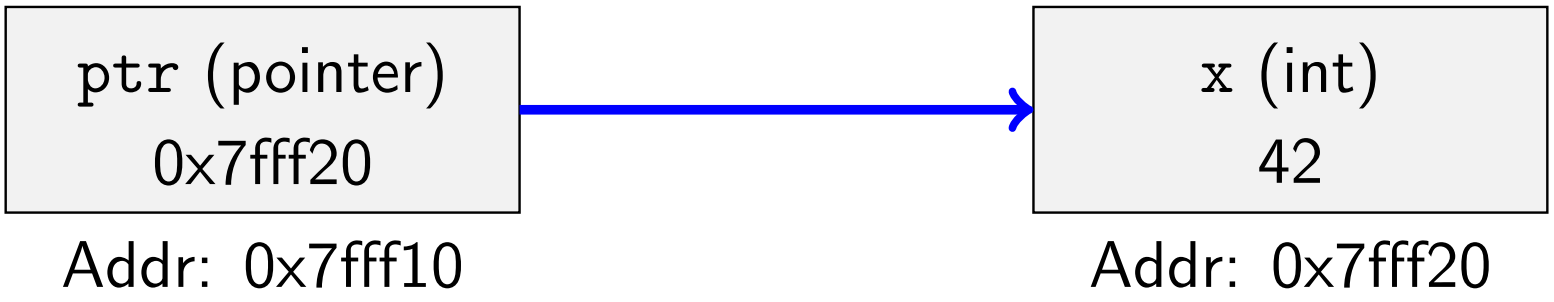
What is a Pointer?

A

pointer is a variable whose value is the address of another variable, i.e., direct address of the memory location.

- Pointers store hexadecimal memory addresses (e.g., `0x7ffd5ebde4ac`) rather than direct data values.
- Declared using the asterisk operator: `data_type *pointer_name;`
- The address-of operator `&` retrieves the memory location of a variable.
- The dereference operator `*` accesses the value stored at the address pointed to by the pointer.

Memory Representation of Pointers (Diagram)



- A pointer variable (`ptr`) occupies its own memory space (e.g., 8 bytes on 64-bit systems) at address `0x7fff10`.
- The value stored inside `ptr` is the memory address of the target variable `x` (`0x7fff20`).
- Dereferencing `ptr` (`*ptr`) instructs the CPU to look at address `0x7fff20` and retrieve or modify its content.
- Pointers must be initialized to a valid address or `NULL`/`nullptr` to prevent segmentation faults.

Pointer Arithmetic and Array Relationship

P

ointer arithmetic allows traversing memory addresses sequentially. An array name acts as a constant pointer to its first element.

- Incrementing a pointer (`ptr++`) increases its address value by `sizeof(type)` bytes, not 1.
- For a 4-byte integer array, `ptr + 1` references an address offset by 4 bytes.
- Subscript notation `arr[i]` is internally evaluated by the compiler as `*(arr + i)`.
- Subtracting two pointers of the same type yields the number of elements between them.

Dynamic Memory Allocation in C/C++

S

tatic allocation occurs at compile-time (stack), while dynamic allocation occurs at runtime (heap) using specific system calls.

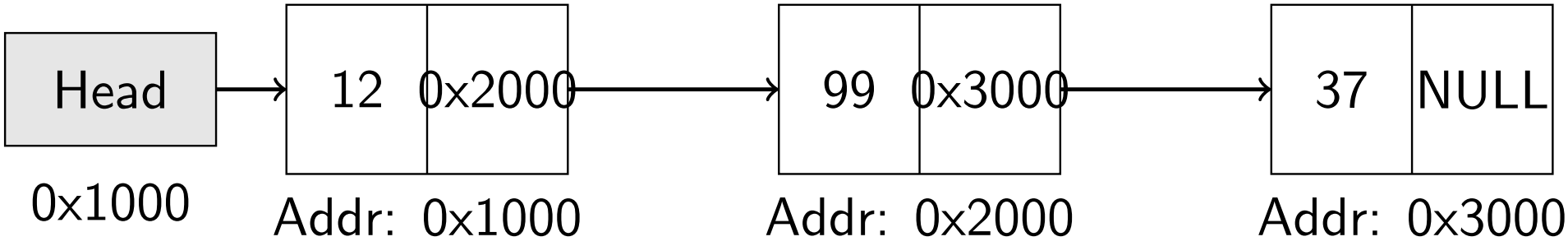
- `malloc(size_t size)` allocates a block of raw memory on the Heap and returns a `void*`.
- In C++, `new` allocates memory and automatically calls the object's constructor.
- Memory allocated dynamically must be manually deallocated using `free()` in C or `delete`/`delete[]` in C++ to prevent memory leaks.
- Dereferencing a pointer after its memory has been freed leads to Undefined Behavior (Dangling Pointer).

Pointer-based Node Structure

```
struct Node {
    int data;
    Node* next;
};
```

- A Node is the fundamental building block of a Linked List.
- It contains at least one data member to store the payload.
- It contains a self-referential pointer (`Node* next`) that points to the next node in the sequence.
- The last node's `next` pointer is set to `nullptr` (or `NULL`) to mark the end of the list.

Linked List Structure in Memory (Diagram)



- The Head pointer stores the address of the first node (`0x1000`).
- Unlike sequential arrays, nodes in a linked list can be scattered non-contiguously in memory.
- Each node's next field contains the absolute heap address of the succeeding node.
- Traversal is performed by updating a cursor pointer: `temp = temp->next;` until `nullptr` is reached.

Double Pointers and Modifying Head

A

double pointer (`Type**`) is a pointer that stores the memory address of another pointer variable.

- Used when a function needs to modify a pointer passed to it (e.g., updating the Head of a Linked List).
- Passing a pointer by value to a function only modifies the local copy, not the caller's pointer.
- In C, updating the list head requires passing `&head` (a double pointer `Node**`) to functions like `insertAtBeginning`.
- In C++, this can also be achieved using references to pointers: `void insert(Node*& head)`.

Common Pointer Bugs and Memory Safety

P

ointer misuse can cause severe system errors like segmentation faults, memory corruption, and security vulnerabilities.

- Memory Leak: Failing to free dynamically allocated memory before losing all pointer references to it.
- Dangling Pointer: A pointer pointing to a memory location that has been deallocated (use after free).
- Wild Pointer: An uninitialized pointer pointing to an arbitrary memory address.
- Null Pointer Dereferencing: Attempting to access members or data through a `nullptr` pointer.

Arrays vs. Linked Lists Pointer Overhead

T

rade-offs between array-based structures and pointer-based linked list structures.

- Arrays have $O(1)$ random access but fixed size and contiguous allocation requirements.
- Linked Lists offer $O(1)$ insertions/deletions at known positions and dynamic size at runtime.
- Linked Lists have an extra storage cost: each node requires pointer overhead (e.g., 8 additional bytes on 64-bit systems).
- Pointers degrade cache locality due to non-contiguous allocation, leading to more cache misses than arrays during traversal.

Data Structures (DI03000021)

GTU Diploma Engineering

Anatomy of a Node

Node Definition

A node is the fundamental structure of a linked list. It is split into data storage and pointers.

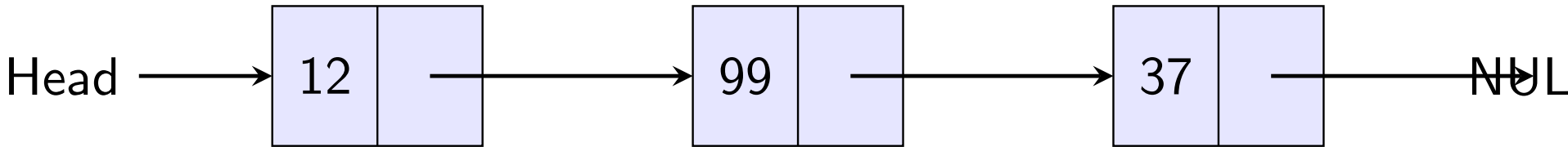
- **Data Field:** Holds the user payload (e.g., integers, objects, structures).
- **Link/Pointer Field:** Stores the memory address of the next node in sequence.
- **Self-Referential Nature:** Nodes contain pointer references pointing to their own type definition.
- **NULL Pointer:** Indicates that the current node has no successor, denoting the end of the list.

Node Representation in C/C++

```
struct Node {
    int data;
    struct Node* next;
};
```

- The declaration `struct Node* next;` is self-referential as it references the same structure type.
- **Size in Memory:** Consists of an integer (4 bytes) and a pointer (8 bytes on 64-bit systems).
- **Structure Padding:** The compiler adds 4 bytes of padding to align the pointer to an 8-byte boundary, totaling 16 bytes.
- **Heap Allocation:** Nodes are allocated dynamically using `malloc(sizeof(struct Node))` or `new Node()`.

Memory Layout of a Singly Linked List



- Head pointer holds the address of the first node (address of node with data value 12).
- **Non-contiguous addresses:** Nodes can be scattered across different regions of heap memory.
- Logical sequential access is maintained strictly via pointer links.
- Last node (data value 37) points to NULL, signifying list termination.

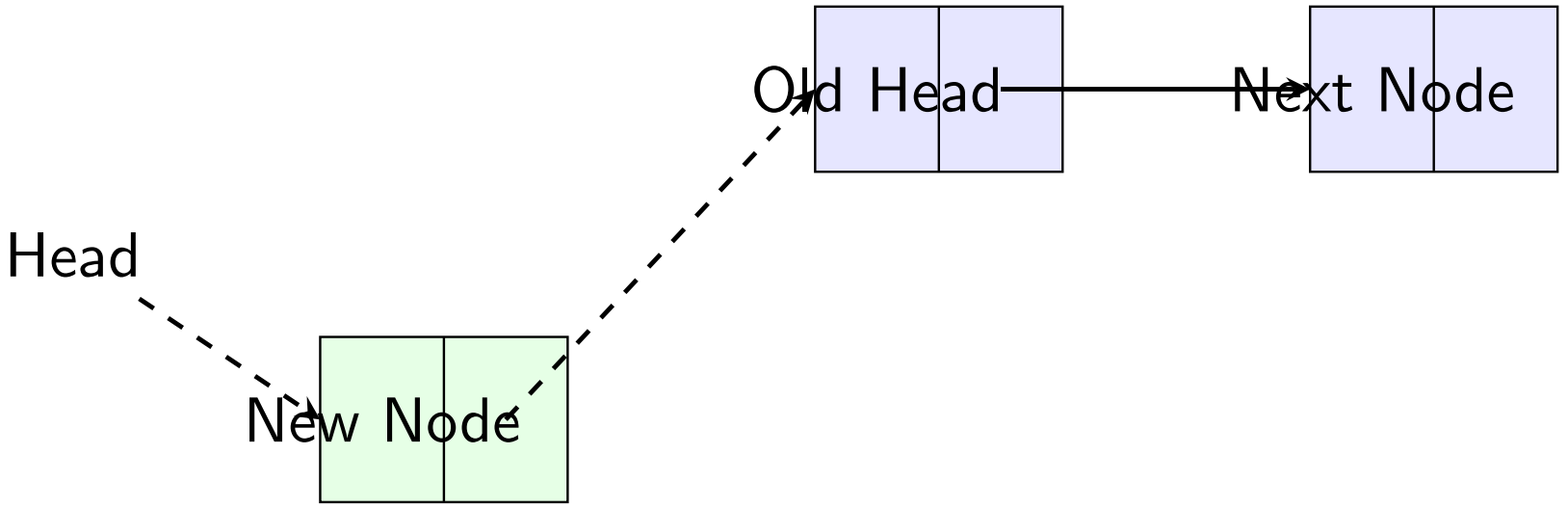
Dynamic Memory and Spatial Locality

Memory Utilization

Analyzing how linked lists utilize memory compared to contiguous arrays.

- **Static vs Dynamic:** Arrays allocate a fixed contiguous block, while linked lists allocate nodes on-demand.
- **Poor Spatial Locality:** Cache lines are populated with adjacent memory. Since list nodes are non-contiguous, cache misses are highly frequent.
- **Allocation Overhead:** Each node incurs allocation metadata overhead from the operating system's heap manager.
- **Pointer Overhead:** Additional memory cost of storing 8-byte pointer addresses on 64-bit platforms.

Inserting a Node at the Head



- **Step 1:** Allocate space for New Node and populate its data field.
- **Step 2:** Assign New Node's next pointer to point to the current Head (Old Head).
- **Step 3:** Update the Head pointer to refer to New Node.
- **Efficiency:** Requires $O(1)$ operations, as no elements need to be shifted in memory.

Doubly Linked List Structure

Doubly Linked List

A bidirectional structure allowing movement in both directions.

- **Two Pointers:** Each node stores reference addresses to both its successor (`next`) and predecessor (`prev`).
- **Structural Layout:** `struct Node { int data; Node* next; Node* prev; };`
- **Advantages:** Easier to delete a node when only given a reference to it; allows reverse traversal.
- **Disadvantages:** Requires 8 additional bytes of pointer memory per node, plus extra pointer maintenance overhead.

Circular Structures and Sentinels

Circular Lists & Sentinels

Circular referencing and dummy nodes to simplify boundary check logic.

- **Circular Lists:** The tail node's next pointer connects back to the head node instead of referencing NULL.
- **Sentinel Node:** A dummy node inserted at the head or tail that stores no user data.
- **Simplifying Code:** Sentinels guarantee that insertion and deletion operations never deal with a NULL head.
- **Performance:** Reduces condition branches (if-statements) inside insertion and deletion loops.

Object-Oriented Node Structure (Java)

```
public class LinkedList {
    private class Node {
        int data;
        Node next;
        Node(int d) {
            this.data = d;
            this.next = null;
        }
    }
    private Node head;
}
```

- **Inner Classes:** The Node class is encapsulated inside the LinkedList class to hide implementation.
- **Object References:** Java doesn't support pointers directly; it uses object reference variables which are safe and managed.
- **Automatic Reclamation:** Dereferenced nodes are

Structural Performance Analysis

Algorithmic Properties

Algorithmic properties of Linked List operations based on structural design.

- **Indexing / Search:** $O(N)$ access complexity because we must traverse nodes sequentially.
- **Head Insert / Delete:** $O(1)$ time complexity as it only requires pointer manipulation.
- **Tail Insert (with tail pointer):** $O(1)$ time complexity by updating the tail reference.
- **Space Complexity:** $O(N)$ linear space, but requires $O(N)$ auxiliary overhead for pointers.

Data Structures (DI03000021)

GTU Diploma Engineering

Motivation for Structure Pointers

Efficiency of Structure Pointers

Passing a structure pointer passes only a memory address (4 or 8 bytes), achieving $O(1)$ overhead compared to $O(N)$ full structure copying.

- Passing structures by value copies the entire structure onto the stack, resulting in $O(N)$ copy overhead.
- Passing a structure pointer passes only a memory address (4 or 8 bytes), achieving $O(1)$ time and space overhead.
- Allows functions to modify the original structure directly without returning a new copy.
- Essential for implementing recursive and dynamic data structures like Linked Lists.

Declaring a Structure Pointer

Structure Pointer Declaration

Syntax for declaring and allocating structure pointers in C.

- Define structure: `struct Node { int data; struct Node* next; };`
- Declaration: `struct Node *ptr;` creates a pointer variable capable of holding a `struct Node` address.
- Initialization: `ptr = NULL;` prevents pointers from pointing to random memory locations (wild pointers).
- Allocation: `ptr = (struct Node*)malloc(sizeof(struct Node));` allocates memory dynamically from the heap.

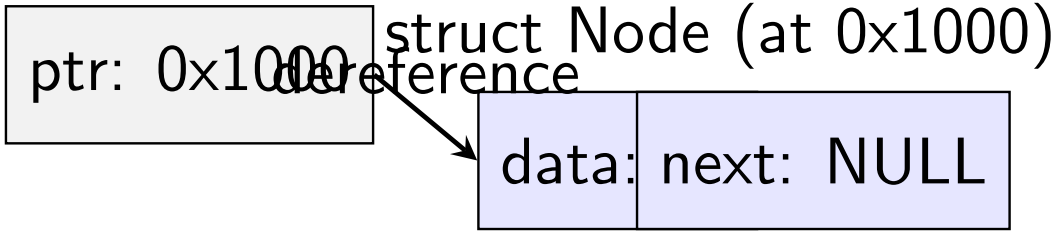
Accessing Structure Members via Pointer

Arrow Operator (->)

The arrow operator `->` is syntactic sugar for dereferencing a pointer and accessing a member: `ptr->data` is equivalent to `(*ptr).data`.

- Dot operator `.` has higher precedence than dereference operator `*`. Thus `*ptr.data` is evaluated as `*(ptr.data)`.
- To access via pointer using dot operator, parentheses are required: `(*ptr).data = 10;`
- The arrow operator `->` is syntactic sugar for dereferencing: `ptr->data = 10;` is equivalent to `(*ptr).data = 10;`
- Using `ptr->next` accesses the pointer member within the structure pointed to by `ptr`.

Memory Layout of a Structure Pointer



- Pointer variable `ptr` resides on the stack and holds the address (e.g., 0x1000) of the heap-allocated node.
- The heap block at 0x1000 contains sequential fields `data` and `next`.
- Using `ptr->data` dereferences `ptr` and offsets to the `data` field.
- The `next` field is initialized to `NULL`, denoting the end of the node or no linked successor.

Dynamic Memory Management

Heap Allocation Lifecycle

Heap memory allocation persists until explicitly deallocated using `free()` in C or `delete` in C++.

- Memory allocated with `malloc` on the heap persists until explicitly freed, unlike stack memory.
- Deallocation is performed using `free(ptr);` in C or `delete ptr;` in C++.
- Memory leaks occur when the pointer variable is reassigned or goes out of scope without freeing the allocated heap memory.
- After freeing, `ptr` becomes a dangling pointer; it should be set to `NULL` to avoid illegal memory access.

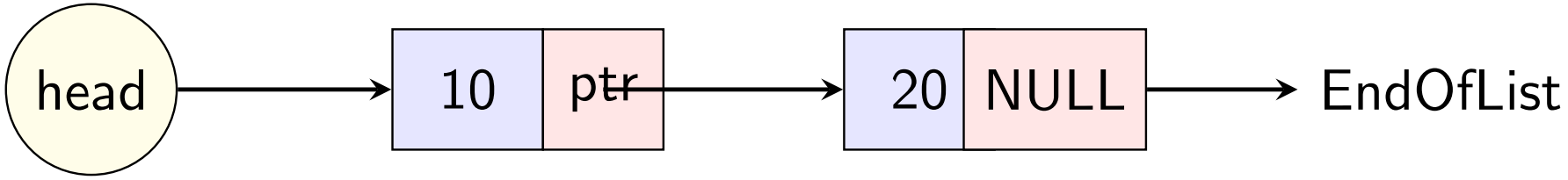
Self-Referential Structures

Self-Referential Structure

A structure that contains a pointer to an instance of the same structure type, forming the foundation of Linked Lists.

- A self-referential structure contains a pointer to an instance of the same structure type.
- Example: `struct Node { int value; struct Node* next; };` is self-referential.
- An instance cannot contain itself directly (`struct Node next;` is compilation error due to recursive sizing).
- Using a pointer (`struct Node* next;`) is valid because pointers have a fixed size regardless of the struct they point to.

Visualizing Node Linkage



- The `head` pointer holds the address of the first node (containing value 10).
- First node's `ptr` contains the memory address of the second node (containing value 20).
- Second node's pointer is `NULL`, indicating the end of the list.
- Linking pointers allows a chain of structures to be dynamically created and traversed sequentially.

Traversal Using Structure Pointers

Sequential Traversal

Traversing a linked list requires initializing a pointer to head and updating it iteratively: `current = current->next;`.

- Create a temporary traversal pointer: `struct Node* current = head;`
- Loop condition: `while (current != NULL)` to process each node in sequence.
- Update pointer in loop body: `current = current->next;` moves the pointer to the next structure address.
- Time complexity of traversal is $O(N)$ where N is the number of nodes, and space complexity is $O(1)$ auxiliary pointer usage.

Common Pitfalls & Best Practices

Best Practices

Always check for `NULL` pointers before dereferencing, initialize pointers properly, and free allocated memory to prevent Segmentation Faults and Memory Leaks.

- Dereferencing a `NULL` pointer: Attempting to access `ptr->data` when `ptr` is `NULL` causes a Segmentation Fault.
- Uninitialized pointers: Ensure all pointers are initialized to `NULL` or a valid memory address before dereference.
- Freeing memory multiple times (double free) leads to heap corruption and security vulnerabilities.
- Check allocation status: Always verify if `malloc` returned `NULL` before using the allocated structure pointer.

Data Structures (DI03000021)

GTU Diploma Engineering

Limitations of Arrays & The Need for Linked Lists

S

equential structures like arrays require contiguous memory allocations which pose serious drawbacks for dynamic datasets.

- Size must be declared at compile time (static array) or at run time with a fixed size (dynamic array), leading to memory under-utilization or capacity limits.
- Insertions and deletions at arbitrary positions are expensive, requiring element shifting with $O(N)$ time complexity.
- Array resizing is costly, requiring allocation of a new, larger memory block, copying all existing elements, and freeing the old block.
- A linked list overcomes these limitations by using non-contiguous memory allocations called nodes, linked via pointers.

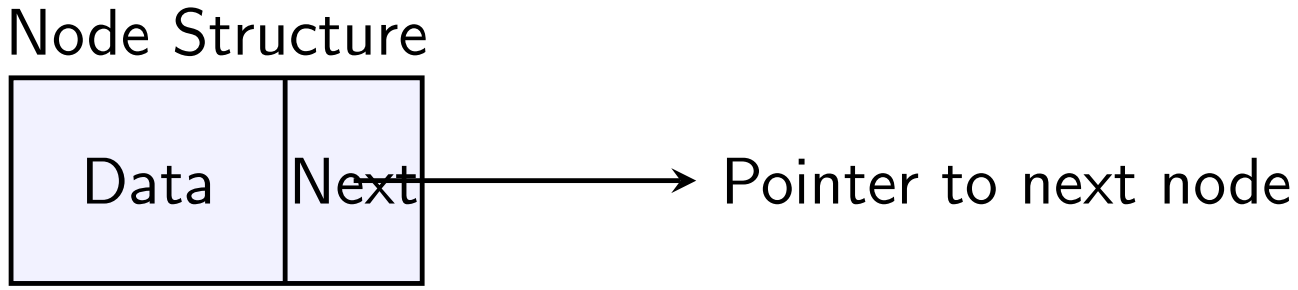
The Self-Referential Structure: Node Representation

- A node is a structure with at least two fields: a data field and a pointer field.
- The data field holds the payload (e.g., an integer value).
- The pointer field 'next' holds the memory address of another object of the same structure type.
- This is a self-referential structure, meaning it contains a pointer to an instance of itself, which enables dynamic growth.

Dynamic Memory Allocation: malloc and free in C

- The 'malloc()' function allocates the requested size of memory on the heap and returns a void pointer to the first byte.
- We use 'sizeof(struct Node)' to compute the exact number of bytes needed dynamically based on the system's architecture.
- Checking if 'malloc' returns 'NULL' is critical to avoid system crashes from dereferencing a null pointer.
- 'free()' releases the allocated memory back to the operating system; setting the pointer to NULL prevents 'dangling pointer' issues.

Visualizing a Linked List Node in Memory

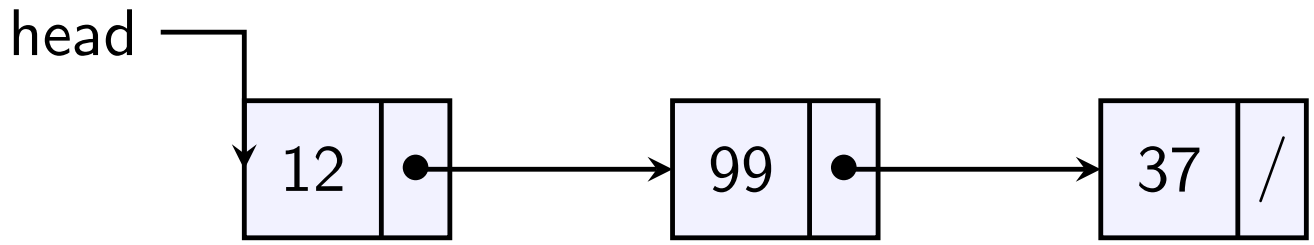


- In memory, the fields of a node layout are stored contiguous within that specific node.
- The first part stores the variable's value (e.g., 'data' of type int, 4 bytes).
- The second part stores the address of the next node (e.g., 'next' pointer, 8 bytes on 64-bit systems).
- The pointer stores a memory address (e.g. 0x7ffd58) that points to the start of the next node's memory boundary.

C++ Dynamic Allocation: new and delete Operators

- C++ provides type-safe memory allocation using the 'new' operator, which automatically computes size and calls constructors.
- 'nullptr' is introduced in C++11 as a type-safe null pointer constant, replacing the macro NULL.
- The 'delete' operator calls the destructor of the class/struct and deallocates memory from the heap.
- Constructor initialization list ': data(val), next(nullptr)' ensures fields are initialized immediately upon node creation.

Visualizing a Linked List: Chain of Nodes



- A Singly Linked List is represented as a chain of nodes where each node points to the next.
- The 'head' pointer stores the memory address of the first node; if the list is empty, head points to NULL.
- The last node points to 'NULL' (indicated by '/' or diagonal cross), signifying the end of the list.
- Nodes are not required to be contiguous in memory; they are connected dynamically via pointers.

Memory Layout: Stack vs. Heap

A

program's memory is divided into Stack and Heap. Let's compare their roles in linked lists.

- Stack: Fast allocation/deallocation, managed automatically by the compiler. Local pointer variables (like 'head', 'temp') are stored here.
- Heap: Large pool of memory, managed manually by the programmer using malloc/new/free/delete. Linked list nodes reside here.
- Stack variables disappear when their scope ends, but heap-allocated nodes persist until explicitly deallocated or the program terminates.
- Memory Leak: Happens when a heap node's address is lost (no stack pointer refers to it) before it is deallocated.

Linked List Traversal Algorithm

- We initialize a helper pointer 'current' to point to the head node, preserving the head pointer itself.
- The loop condition 'current != nullptr' guarantees that every node is visited, stopping after the last node is processed.
- The statement 'current = current-¿next' moves the cursor forward by updating the pointer variable to store the address of the next node.
- Time Complexity: $O(N)$ as we visit N nodes sequentially. Space Complexity: $O(1)$ auxiliary space.

Comparison: Arrays vs. Linked Lists

S

ummary of critical trade-offs between static contiguous arrays and dynamic non-contiguous linked lists.

- Access Time: Arrays allow $O(1)$ random access by index. Linked Lists require $O(N)$ sequential search from the head.
- Insertion/Deletion: Arrays take $O(N)$ time due to element shifting. Linked Lists support $O(1)$ insert/delete at a known pointer location.
- Memory Efficiency: Arrays have zero pointer overhead but can waste memory. Linked Lists have pointer overhead (8 bytes per node) but grow dynamically.
- Cache Friendliness: Arrays exhibit excellent spatial locality (stored sequentially). Linked Lists have poor locality, potentially causing more CPU cache misses.

Data Structures (DI03000021)

GTU Diploma Engineering

What is a Singly Linked List?

A

singly linked list is a linear data structure where elements are stored in nodes, and each node points to the next node in the sequence.

- Unlike arrays, elements are not stored in contiguous memory locations; instead, they are linked using pointers.
- Each node consists of two parts: a data field to store the actual value, and a next pointer to store the address of the subsequent node.
- The list has a 'head' pointer referencing the first node. If the list is empty, 'head' is NULL.
- The last node's 'next' pointer points to NULL, indicating the end of the list.

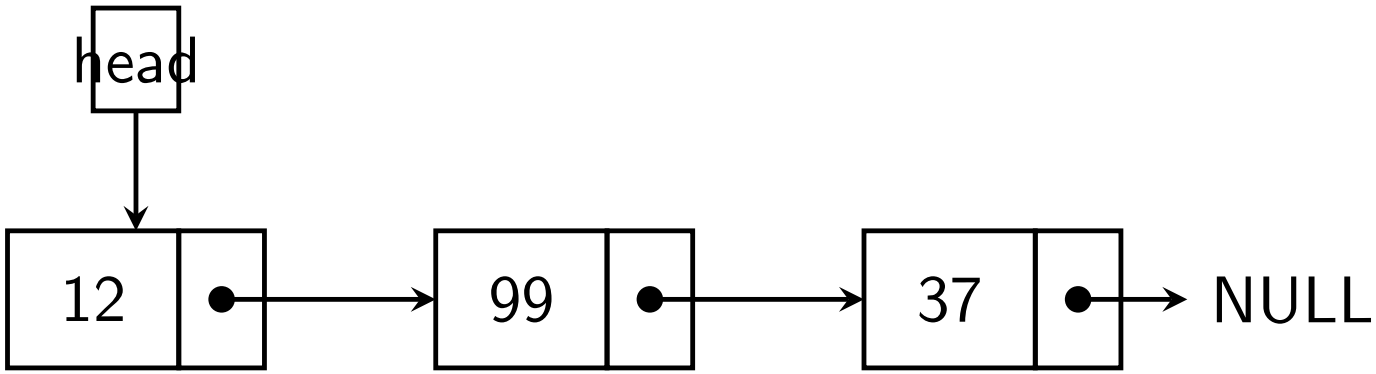
Anatomy of a Node

A node is the fundamental building block of a linked list. It is represented in C++ as:

```
struct Node {
    int data;
    Node* next;
};
```

- The data member can store any data type (e.g., int, float, custom objects).
- The next pointer holds the memory address of the next Node object in the list.
- When a node is created, its next pointer is typically initialized to `nullptr`.
- Nodes are dynamically allocated on the heap using the `new` keyword in C++.

Singly Linked List Diagram



- This diagram shows a linked list with three nodes containing values 12, 99, and 37.
- The head pointer points to the first node (containing 12).
- The next pointer of the last node (containing 37) is NULL.
- Arrows represent the direction of pointers, which are one-way (singly linked).

Basic Operations Overview

S

ingly linked lists support standard operations including insertion, deletion, search, and traversal.

- Traversal: Iterating through the nodes starting from the head until the next pointer is NULL.
- Insertion: Adding a node. Can be at the beginning, at the end, or at a specific position.
- Deletion: Removing a node. Can be from the beginning, from the end, or from a specific position.
- Search: Looking for a node containing a specific data value by traversing the list linearly.
- Time Complexity: Searching and arbitrary insertions/deletions take O(N) time in the worst case.

Traversal Algorithm & Code

```
void traverse(Node* head) {
    Node* current = head;
    while (current != nullptr) {
        std::cout << current->data << " -> ";
        current = current->next;
    }
    std::cout << "NULL" << std::endl;
}
```

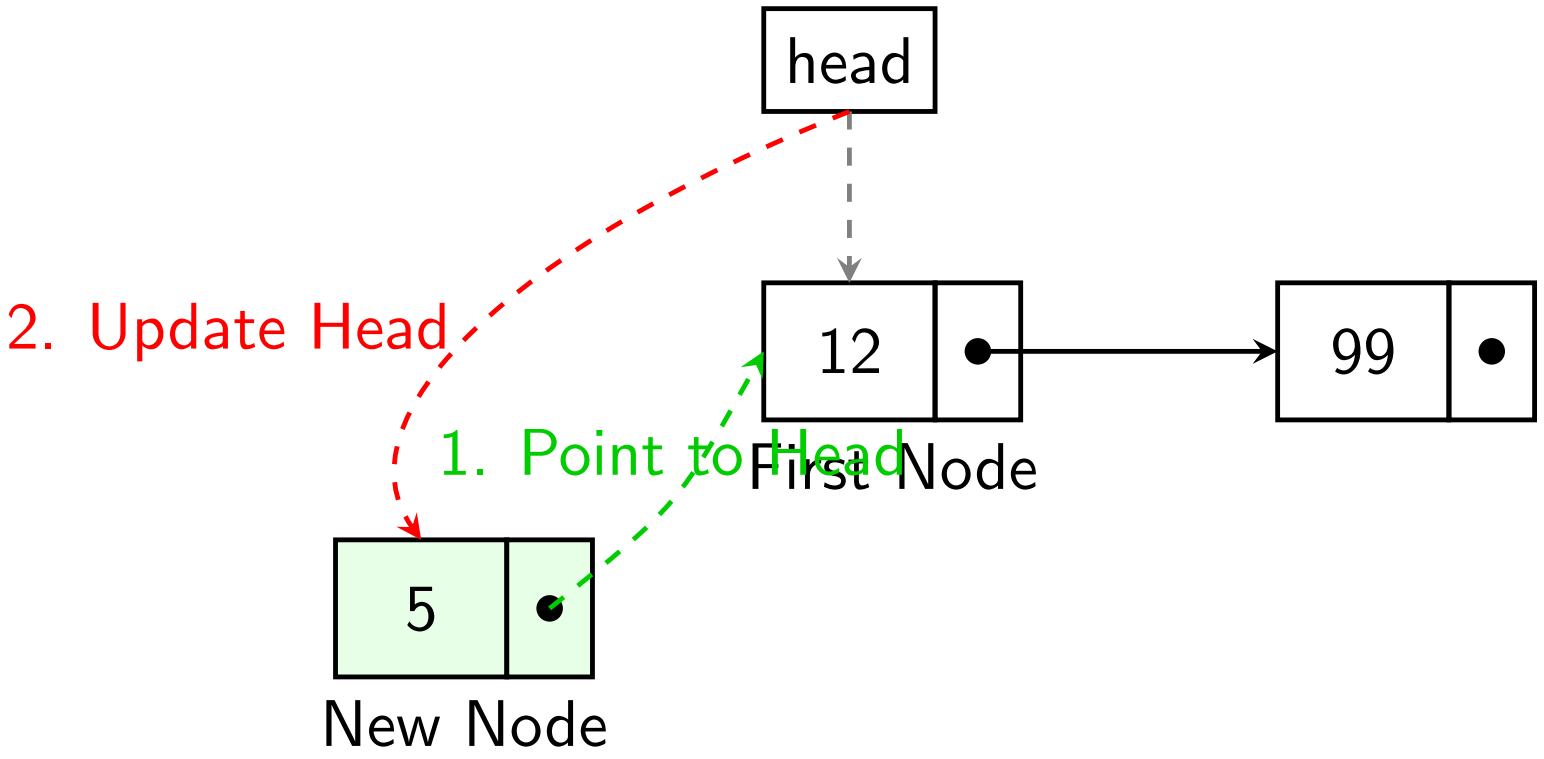
- To traverse, we initialize a temporary pointer 'current' to the 'head' node.
- We use a loop that continues as long as 'current' is not `nullptr`.
- Inside the loop, we access the data and advance 'current' to `current->next`.
- Time Complexity: O(N) because we visit every node exactly once.

Insertion at the Beginning (Prepend)

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

- Step 1: Allocate memory for the new node (`newNode`) and assign the value to its data field.
- Step 2: Set the next pointer of `newNode` to point to the current head node.
- Step 3: Update head to point to `newNode` so it becomes the first node.
- Time Complexity: O(1) since we only update pointers regardless of the list size.
- Crucial: head must be passed by reference (`Node*&`) or we must return the new head.

Visualizing Prepend (Insertion at Head)



- This diagram shows how prepend updates connections without traversing the list.
- First, the new node's next pointer is set to point to the current first node (value 12).
- Second, the head pointer is updated to point to the new node (value 5).
- This operation runs in O(1) constant time, which is much

Deletion of a Node (First Node)

```
void deleteFirstNode(Node*& head) {
    if (head == nullptr) return;
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

- Step 1: Check if the list is empty. If it is, there is nothing to delete.
- Step 2: Store the current head node in a temporary pointer `temp` to keep track of it for memory deallocation.
- Step 3: Advance the head pointer to `head->next` (the second node).
- Step 4: Free/delete the memory occupied by `temp` to prevent memory leaks.
- Time Complexity: O(1) as it only involves changing pointers and releasing one node's memory.

Summary, Advantages & Disadvantages

S

ingly Linked Lists provide dynamic memory management but suffer from sequential access limitations.

- Advantages: Dynamic size (grows/shrinks on demand), efficient insertion/deletion at head (O(1)), no memory wastage for pre-allocation.
- Disadvantages: Sequential access (O(N) search and access), extra memory overhead for storing the pointer, lack of backward traversal.
- Alternative variants: Doubly Linked Lists (two-way navigation), Circular Linked Lists (circular traversal).
- Common Use Cases: Implementation of Stacks, Queues, Adjacency Lists for Graphs, and Hash Chaining.

Data Structures (DI03000021)

GTU Diploma Engineering

Limitations of Singly Linked Lists

Singly Linked List (SLL) Constraints

A standard Singly Linked List (SLL) stores nodes with a single forward pointer (`next`). While memory-efficient, SLL has structural constraints that limit traversal flexibility and algorithm performance in specific contexts.

- **Unidirectional traversal:** Traversals are strictly head-to-tail; reversing direction requires rebuilding stack contexts or auxiliary storage.
- **Predecessor access penalty:** Finding the node prior to a target node requires a full $O(N)$ search from the head of the list.
- **Boundary transitions:** The tail node's `next` field is null, requiring reset logic to re-traverse the list.
- **Practical limitations:** Hard to implement applications

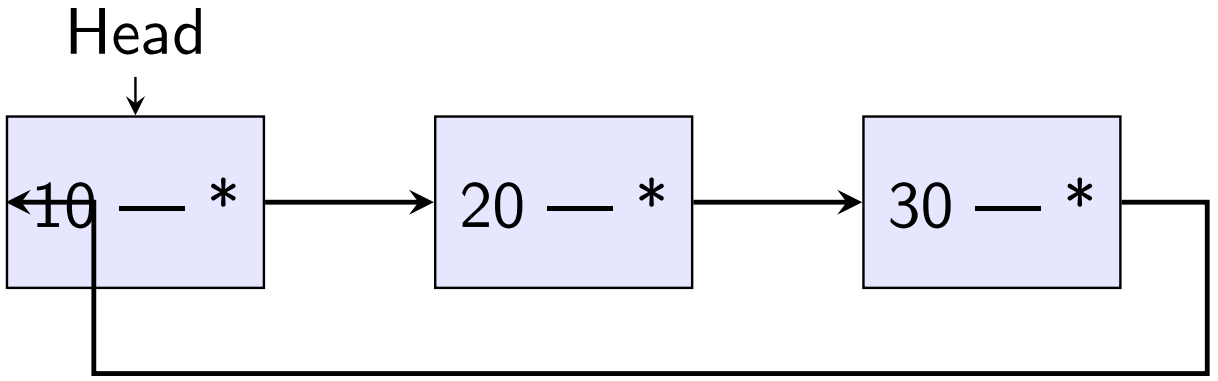
Concept of Circular Singly Linked Lists

Circular Singly Linked List (CSLL)

A Circular Singly Linked List (CSLL) removes the concept of a "NULL tail". The final node's `next` pointer points directly back to the head node, forming a closed loop structure.

- **Continuous loop:** You can start at any arbitrary node and traverse the entire list without encountering a terminal null pointer.
- **Tail pointer optimization:** Storing a pointer to the Tail node instead of the Head node gives $O(1)$ access to both the tail (`tail`) and the head (`tail->next`).
- **Use cases:** Highly suited for circular queues, round-robin process scheduling in operating systems, and media player playlist loops.
- **Traversal completion:** Terminating traversals requires

CSLL Structural Diagram



- Visual layout showing nodes 10, 20, and 30.
- Notice how the `next` field of node 30 wraps back to the input of node 10.
- There are no NULL pointers, representing a complete circular loop.

CSLL Operations and Algorithms

CSLL Operations Overview

Due to the lack of NULL boundaries, boundary insertions and deletions in a CSLL require precise updates to avoid breaking the circular link chain.

- **Insert at Head:** Create node, set `node->next = tail->next`, `tail->next = node`. This takes $O(1)$ time using a tail pointer reference.
- **Insert at Tail:** Create node, set `node->next = tail->next`, `tail->next = node`, `tail = node`. This is also $O(1)$ time.
- **Delete at Head:** If single node, set `tail = NULL`. Otherwise, `tail->next = tail->next->next`. Free original head node ($O(1)$ time).
- **Delete at Tail:** Traverse to find node predecessor (X) where `X->next == tail`. Set `X->next = tail->next`, `tail = X`, free old tail. Takes $O(N)$ time.

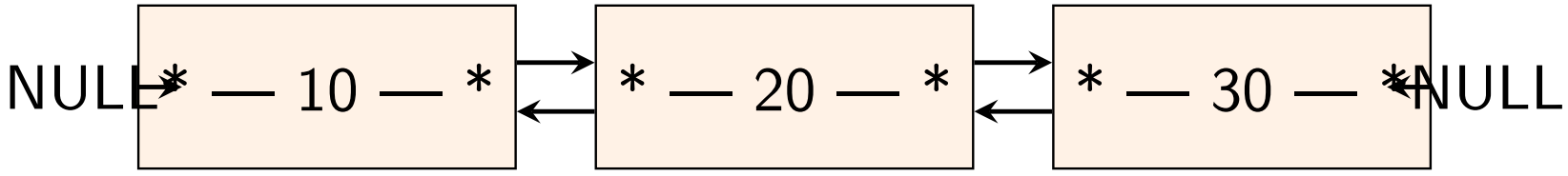
Introduction to Doubly Linked Lists

Doubly Linked List (DLL)

A Doubly Linked List (DLL) addresses SLL limitations by adding a second pointer field (`prev`) to each node, pointing to the preceding node in the sequence.

- **Node structure:** Contains a `prev` pointer, `data` field, and `next` pointer, doubling pointer storage requirements.
- **Bidirectional traversal:** Can traverse from head to tail, or tail to head, allowing efficient bidirectional algorithms.
- $O(1)$ **node deletion:** A node can delete itself in $O(1)$ time when provided a direct pointer to it, as its predecessor is known.
- **Memory/Code complexity tradeoff:** Simplifies algorithms but increases memory per node and requires maintenance of double pointers during updates.

DLL Structural Diagram



- Visualizes the bidirectional link between nodes 10, 20, and 30.
- Note the use of two opposing arrows between adjacent nodes representing `prev` and `next` pointers.
- The outermost pointers end in NULL, defining list boundary termination.

DLL Insertion and Deletion Algorithms

Pointer Maintenance in DLL

DLL updates require modifying twice as many pointers as SLL. Both forward (`next`) and backward (`prev`) links must be carefully updated in sequence.

- **Insert at Head:** Create node, set `node->next = head`, `node->prev = NULL`. If `head != NULL`, `head->prev = node`. Set `head = node` ($O(1)$ complexity).
- **Insert after Node X:** Create node, set `node->next = X->next`, `node->prev = X`. If `X->next != NULL`, set `X->next->prev = node`. Set `X->next = node` ($O(1)$ complexity).
- **Delete Node X:** If X is head, update `head = X->next`. If `X->next != NULL`, `X->next->prev = X->prev`. If `X->prev != NULL`, `X->prev->next = X->next`. Free X ($O(1)$ complexity).

Circular Doubly Linked List (CDLL)

Circular Doubly Linked List (CDLL)

A Circular Doubly Linked List (CDLL) merges circularity with bidirectionality. The head node's `prev` pointer points to the tail node, and the tail node's `next` pointer points to the head node.

- **Double circularity:** No boundary NULL pointers exist. The structure is fully circular in both directions.
- **Constant-time tail operations:** If you have a head reference, the tail is directly accessible via `head->prev`. Tail operations take $O(1)$ time without an explicit tail variable.
- **Use cases:** Advanced buffer layouts, browser tabs navigation (`Ctrl+Tab` / `Ctrl+Shift+Tab`), and continuous undo-redo circular history.
- **Complexity:** Most complex linked structure to maintain; a single insertion or deletion requires updating exactly 4 pointers.

Linked List Structures Comparison

Overview of Structure Tradeoffs

A side-by-side complexity and memory overhead comparison for Singly Linked List (SLL), Circular Singly Linked List (CSLL), Doubly Linked List (DLL), and Circular Doubly Linked List (CDLL).

- **Search Complexity:** $O(N)$ across all structures as nodes are not indexed.
- **Delete Tail:** $O(N)$ in SLL and CSLL (must find predecessor); $O(1)$ in DLL (with tail pointer) and CDLL (via `head->prev`).
- **Insert Head/Tail:** $O(1)$ for CSLL and CDLL using optimal tail or head references.
- **Memory Overhead:** SLL and CSLL require N pointers. DLL and CDLL require $2N$ pointers, plus additional code size for complex pointer maintenance.

Data Structures (DI03000021)

GTU Diploma Engineering

Applications of Linked Lists: Systems & Networking

Overview

Linked lists are the foundation of many system-level utilities where dynamic size, frequent insertion, and deletion are primary requirements.

- **Dynamic Memory Allocation:** Heap managers track free memory blocks of varying sizes using lists (e.g., first-fit or best-fit allocation algorithms).
- **Operating System Scheduling:** Process schedulers maintain queues (like ready queues or blocked queues) as linked lists to easily insert and swap processes.
- **Network Packet Buffering:** Network routers and switches buffer incoming packets in queue structures implemented using linked lists due to unpredictable traffic sizes.
- **File System Allocation:** File systems (such as FAT) use chained block allocation where each cluster points to the next

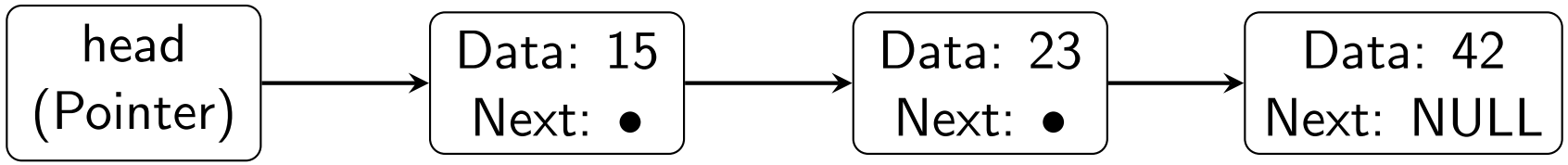
Applications of Linked Lists: Software & ADT Implementations

Overview

At the application layer, linked lists serve as structural backbones for various user-facing software behaviors and fundamental Abstract Data Types (ADTs).

- **Undo/Redo Functionality:** Text editors and graphic tools use doubly linked lists to traverse backward and forward through state histories.
- **Web Browser Navigation:** Forward and backward page history navigation maps directly to a doubly linked list structure.
- **Abstract Data Types (ADTs):** Stacks, Queues, and Deques are frequently implemented using linked lists to guarantee $O(1)$ operations without resizing overhead.

Anatomy of a Singly Linked List Node



- A Singly Linked List (SLL) node contains two components: a data field (payload) and a pointer `next` pointing to the subsequent node.
- The `head` pointer stores the memory address of the first node of the list, acting as the single entry point.
- The final node's `next` field points to NULL, indicating the logical termination of the sequence.
- Unlike arrays, nodes are not contiguous in memory; elements are reached via pointer traversal.

Insertion at the Beginning (Prepend Operation)

Prepend Operation

Inserting a new element at the beginning of a singly linked list is an $O(1)$ time complexity operation because it requires no traversal.

- **Step 1:** Allocate memory dynamically for the new node (e.g., using `malloc()` in C or `new` in C++).
- **Step 2:** Assign the data payload value to the new node's data field.
- **Step 3:** Point the new node's `next` pointer to the current first node of the list (the current address stored in `head`).
- **Step 4:** Update the `head` pointer to store the memory address of the newly allocated node.

C Implementation: Insertion at the Beginning

```c
struct Node {
    int data;
    struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, in
    // 1. Allocate memory for node on the heap
    struct Node* new_node = (struct Node*)malloc(
    if (new_node == NULL) return; // Handle alloc

    // 2. Put in the data
    new_node->data = new_data;

    // 3. Make next of new node point to the curr
    new_node->next = (*head_ref);

    // 4. Move the head pointer to point to the n
```

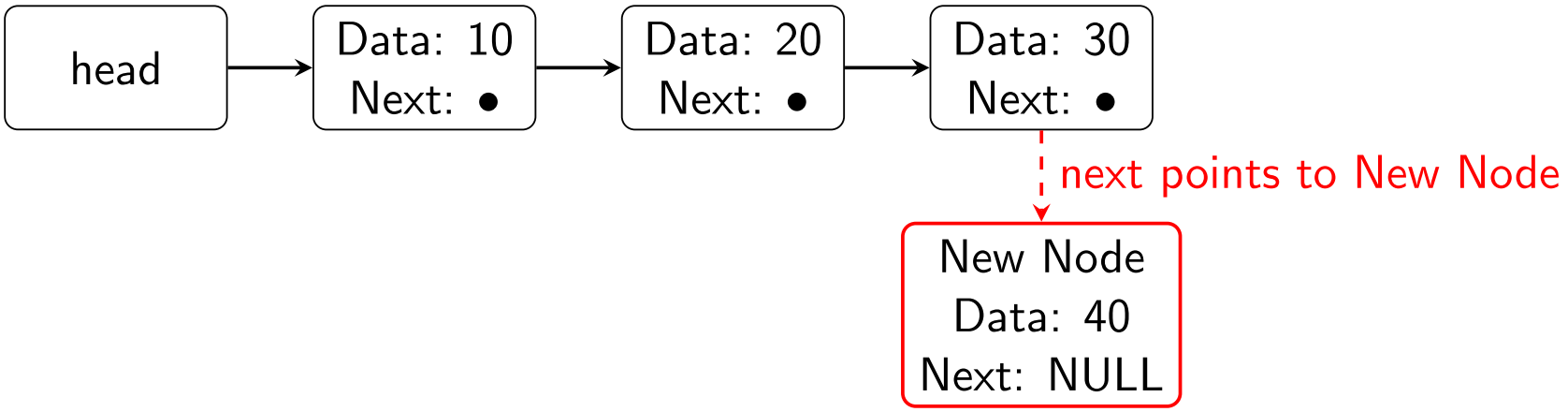
Insertion at the End (Append Operation)

Append Operation

Inserting at the end of a singly linked list requires locating the current terminal node, which involves a linear traversal of the list.

- **Step 1:** Allocate memory for the new node and assign its data. Since it will become the last node, set its `next` pointer to NULL.
- **Step 2:** Check if the list is empty (`head` is NULL). If so, update `head` to point directly to the new node and return.
- **Step 3:** If not empty, initialize a temporary pointer at `head` and loop until the node's `next` field is NULL.
- **Step 4:** Update the `next` pointer of the final node to store the address of the new node.
- **Time Complexity:** $O(N)$ where N is the number of nodes in the list, unless a tail pointer is maintained.

Visualizing Insertion at the End



- **Before insertion:** The node with data 30 is the last node and its `next` pointer is NULL.
- **After traversal:** The pointer `temp` stops at node 30 because `temp->next` is NULL.
- **Linking:** The `next` pointer of node 30 is redirected from NULL to point to the address of New Node (40).
- **Terminal change:** New Node (40) is now the terminal node pointing to NULL.

C Implementation: Insertion at the End

```c
void insertAtEnd(struct Node** head_ref, int new_
    // 1. Allocate node
    struct Node* new_node = (struct Node*)malloc(
    if (new_node == NULL) return;

    new_node->data = new_data;
    new_node->next = NULL; // New node will be th

    // 2. If list is empty, make new node the hea
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }

    // 3. Otherwise, traverse to the last node
    struct Node* last = *head_ref;
    while (last->next != NULL) {
```

Performance Summary and Trade-Offs

Summary

Understanding the efficiency and runtime characteristics of basic singly linked list insertions.

- **Insert at Beginning:** $O(1)$ time complexity, $O(1)$ auxiliary space. Optimal for push-only structures like LIFO stacks.
- **Insert at End (No Tail Pointer):** $O(N)$ time complexity, $O(1)$ auxiliary space. High penalty for large lists due to traversal.
- **Insert at End (With Tail Pointer):** $O(1)$ time complexity. Optimization requires holding and updating a secondary 'tail' reference pointer.
- **Memory overhead:** Every insertion allocates 1 structural node containing a data type and a pointer (8 bytes on 64-bit platforms).

Data Structures (DI03000021)

GTU Diploma Engineering

Structural Overview: The Singly Linked List Node

Singly Linked List Node

In C/C++, a singly linked list node is represented as a structure with a data payload and a pointer linking it to the next node.

- The struct representation is typically: `struct Node { int data; Node* next; };`
- Dynamic memory allocation is performed using `'new Node()'` in C++ or `'malloc(sizeof(Node))'` in C.
- Visualization: A contiguous block of memory logically split into payload (data) and address container (next link).
- Understanding dereferencing (e.g. `node->next`) is crucial for manipulating the links.

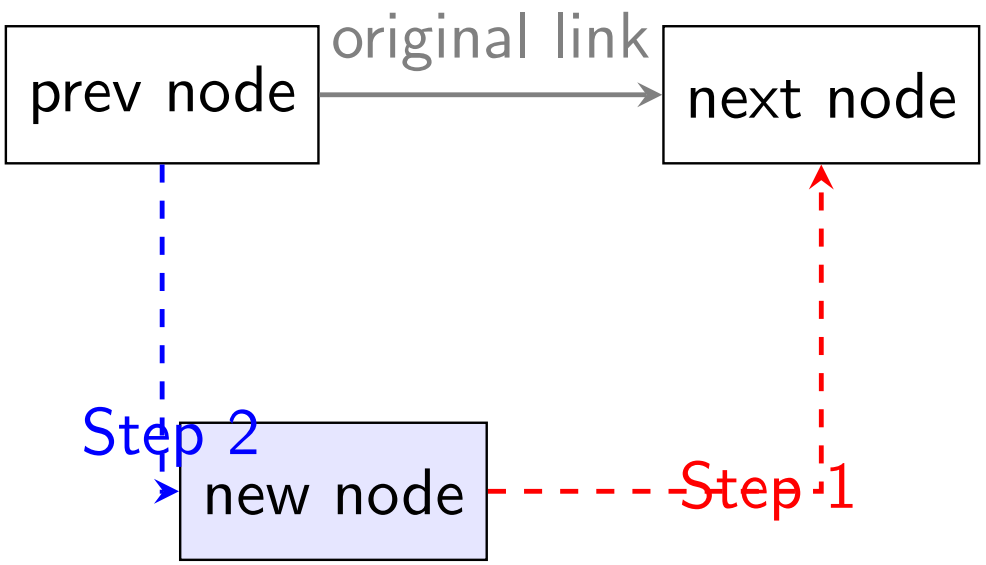
Inserting AFTER a Given Node: Concept & Logic

Objective

The process of inserting a new node immediately after a specified target node (`prev_node`) in the list.

- Verify first that the target node '`prev_node`' is not NULL to avoid undefined pointer behavior.
- Allocate a new node dynamically and populate its data field: `new_node->data = val`.
- Set the new node's next pointer to point to the node originally following `prev_node`: `new_node->next = prev_node->next`.
- Re-route the `prev_node`'s next pointer to point directly to the new node: `prev_node->next = new_node`.
- This sequence is critical. Reversing steps 3 and 4 will orphan the rest of the linked list and cause memory leaks.

Diagram: Inserting After a Node



- The diagram demonstrates the order-critical pointers updates for insert-after operation.
- Step 1: The `'new_node->next'` pointer is assigned the address of `'next_node'` (which was `prev_node->next`).
- Step 2: The `'prev_node->next'` pointer is updated to hold the address of the newly allocated `'new_node'`.
- The operation is completed in $O(1)$ time complexity as we already possess a direct pointer to `'prev_node'`.

Implementation: C++ Function for Insert After

```
void insertAfter(Node* prev_node, int new_data) {
    if (prev_node == nullptr) {
        std::cout << "The given previous node cannot be
        return;
    }
    Node* new_node = new Node();
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}
```

- Time Complexity: $O(1)$ auxiliary time, as no traversal is necessary.
- Space Complexity: $O(1)$ auxiliary space, only allocating a single node.
- Robustness: The check `prev_node == nullptr` prevents segmentation faults and dereferencing errors.

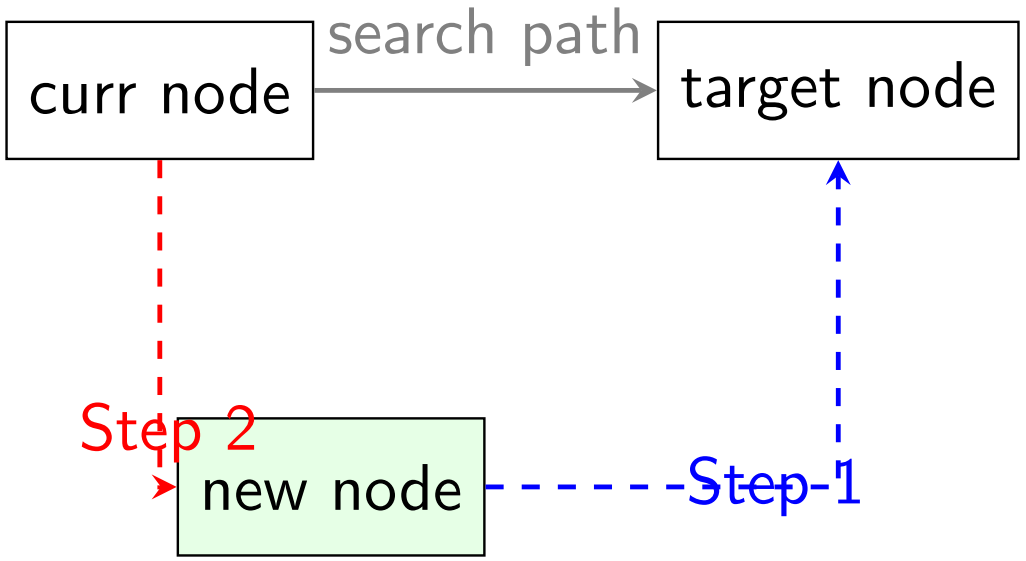
Inserting BEFORE a Given Node: Concept & Logic

Objective

The process of inserting a new node directly before a specified target node (`next_node`) in the list.

- Case 1: If the list is empty, insertion before is impossible; return early or raise an error.
- Case 2: If the target `'next_node'` is the head of the list, this is equivalent to inserting at the head.
- Case 3: For other nodes, we must traverse from the head to find the predecessor node `'curr'` (where `curr->next == next_node`).
- Finding the predecessor is necessary because singly linked lists only possess forward pointer references.

Diagram: Inserting Before a Node



- Unlike insertion 'after', insertion 'before' in a singly linked list requires locating the predecessor node.
- We search sequentially starting at `'head'` until we reach `'curr'` such that `'curr->next == target'`.
- Once found, we update the pointers: `'new_node->next = target'` and `'curr->next = new_node'`.
- This linear scan makes the overall insertion operation run in $O(N)$ time in the worst case.

Implementation: C++ Function for Insert Before

```
void insertBefore(Node** head_ref, Node* next_node, int
    if (head_ref == nullptr || *head_ref == nullptr || n
        return;
    if (*head_ref == next_node) {
        Node* new_node = new Node();
        new_node->data = new_data;
        new_node->next = *head_ref;
        *head_ref = new_node;
        return;
    }
    Node* curr = *head_ref;
    while (curr != nullptr && curr->next != next_node) {
        curr = curr->next;
    }
    if (curr == nullptr) return;
    Node* new_node = new Node();
    new_node->data = new_data;
```

Complexity Analysis Comparison

Operation	Best Case Time	Worst Case Time	Space Cor
Insert After	$O(1)$	$O(1)$	$O(1)$ aux
Insert Before	$O(1)$ (at head)	$O(N)$	$O(1)$ aux

- Insert After is highly efficient $O(1)$ because we already hold the pointer to the preceding node.
- Insert Before is inefficient $O(N)$ because we must traverse from head to find the predecessor node.
- Both operations consume $O(1)$ auxiliary space as only a single node is dynamically created.
- Note: In a Doubly Linked List, Insert Before becomes $O(1)$ because we can access `target->prev` directly.

Key Takeaways and Common Pitfalls

Summary Guidelines

Crucial guidelines for safe pointer manipulations in Linked Lists.

- Always update `new_node->next` before reassigning `predecessor->next` to avoid dereferencing lost nodes.
- Boundary cases (empty list, target is head, target is tail, target not present) must be systematically handled.
- Avoid memory leaks: ensures that every dynamically allocated node is either linked or deleted.
- Understand the trade-offs of Singly vs. Doubly linked lists for relative insertion tasks.

Data Structures (DI03000021)

GTU Diploma Engineering

Problem Definition & Objectives

Problem Statement

Given a pointer to the head of a sorted singly linked list (in ascending order) and a new data value, insert a new node containing this value such that the list remains sorted.

- **Input:** Head pointer of a sorted linked list, and the key/value to insert.
- **Output:** Updated head pointer of the sorted linked list containing the new node.
- **Constraint:** The ascending sorted order (`node.data` $\leq$ `node.next.data`) must be preserved.
- **Key Challenge:** We must find the correct insertion spot before modifying any pointers, which differs from inserting at a known position.

Case Analysis: Insertion Positions

To implement this robustly, we must classify the insertion into three logical scenarios based on the state of the list and the value of the new node.

- **Case 1: Empty List.** The list has no nodes (`head` is `NULL`). The new node becomes the head and only node, pointing to `NULL`.
- **Case 2: Insertion at the Head.** The new value is smaller than or equal to the current head's value. The new node must become the new head.
- **Case 3: Insertion in the Middle or End.** The new value is greater than the head's value. We must traverse the list to find the correct location, inserting either between two nodes or after the tail.

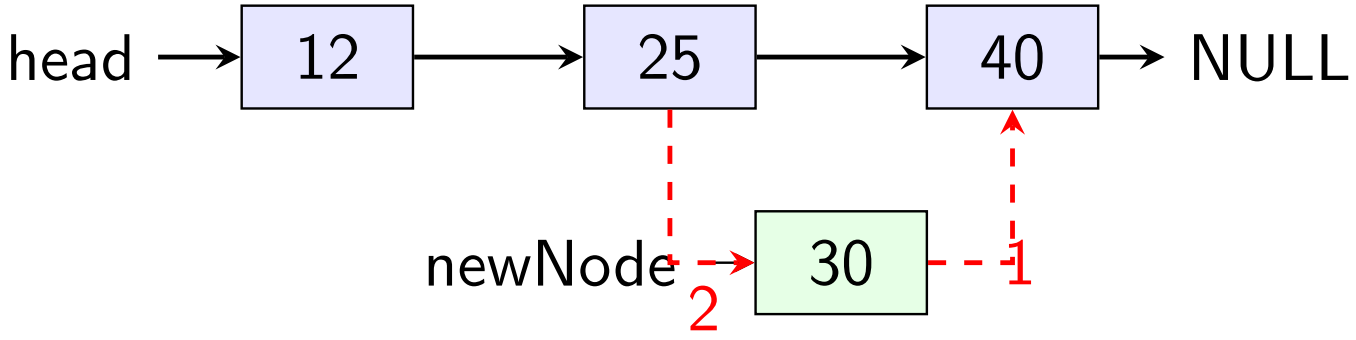
Algorithmic Steps (Pseudocode)

Algorithm: INSERT-SORTED-LINKED-LIST(head

1. Create a new node `newNode` with `data = value`, `next = NULL`.
2. If `head` is `NULL` or `value < head.data` then:
   - `newNode.next = head`
   - `return newNode`
3. Set `current = head`
4. While `current.next` is not `NULL` and `current.next.data < value` do:
   - `current = current.next`
5. `newNode.next = current.next`
6. `current.next = newNode`
7. Return `head`

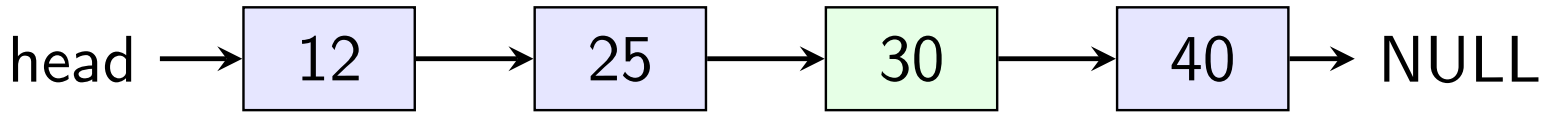
- **Steps 1 & 2** handle Cases 1 and 2 directly by updating the

Visualizing the Insertion (Before)



- We wish to insert a new node with value 30 into the sorted list $[12 \rightarrow 25 \rightarrow 40]$.
- We traverse starting at the head (12). Since $25 < 30$, we move to 25.
- At node 25, we look ahead: `current.next` is 40. Since 40 is not < 30, we stop traversal at node 25.
- Node 25 is the predecessor, and node 40 is the successor for our new node 30.

Visualizing the Insertion (After)



- After resolving pointers, node 25 points to node 30, and node 30 points to node 40.
- The list's sorted order is correctly maintained: $12 \rightarrow 25 \rightarrow 30 \rightarrow 40$.
- Notice that the tail pointer of node 40 and head pointer of 12 remain unchanged.

C++ Implementation Details

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};

Node* insertSorted(Node* head, int val) {
    Node* newNode = new Node(val);
    if (head == nullptr || val < head->data) {
        newNode->next = head;
        return newNode;
    }
    Node* current = head;
    while (current->next != nullptr && current->n
        current = current->next;
    }
    newNode->next = current->next;
```

Complexity Analysis

Complexity Summary (N

Time Complexity:

- **Best Case:** $\mathcal{O}(1)$ (Insertion at head or empty list)
- **Worst Case:** $\mathcal{O}(N)$ (Insertion at tail or close to tail)
- **Average Case:** $\mathcal{O}(N)$ (Traversing $N/2$ nodes on average)

Space Complexity: $\mathcal{O}(1)$ Auxiliary Space

- Best Case occurs when the list is empty or the new value is less than the head's value, requiring only a constant number of operations.
- Worst Case occurs when the new value is greater than all existing elements, requiring a full traversal of the list to reach the end.
- No extra data structures are created; only a single node is

Double Pointer Optimization

```
void insertSortedDoublePointer(Node** headRef, in
    Node* newNode = new Node(val);
    Node** current = headRef;
    while (*current != nullptr && (*current)->dat
        current = &((*current)->next);
    }
    newNode->next = *current;
    *current = newNode;
}
```

- This optimization avoids treating head insertion as a special case by using a double pointer (pointer to pointer).
- The variable `current` points to the `next` field of the predecessor node, or the `head` pointer itself.
- When the loop terminates, `current` is the address of the pointer that needs to be updated.
- Splicing becomes a simple two-step process: assign

Key Takeaways & Summary

Summary of Key Takeaways

- **Pre-conditions:** The list must already be in sorted order.
- **Traversal Strategy:** Look one step ahead (using `current->next`) or use double pointers to capture the predecessor.
- **Boundaries:** Empty lists and head changes are common sources of edge cases.
- **Efficiency:** Highly efficient $\mathcal{O}(1)$ auxiliary space, but linear $\mathcal{O}(N)$ time complexity due to sequential traversal constraints.

- Understanding this algorithm is essential for understanding insertion sort on linked lists.
- Double pointer manipulation is a powerful C/C++ skill that

Data Structures (DI03000021)

GTU Diploma Engineering

Anatomy of Deletion Operations

Overview

Overview of memory management and pointer manipulation in node deletion.

- Deleting a node requires bypassing the target node by linking its predecessor directly to its successor.
- Memory management is vital: dynamic memory allocation (`new`/`malloc`) must be paired with deallocation (`delete`/`free`) to prevent memory leaks.
- Dangling pointers must be avoided by setting unused pointer variables to null after freeing memory.
- We must always handle special boundary cases, such as empty lists or lists containing only a single element.

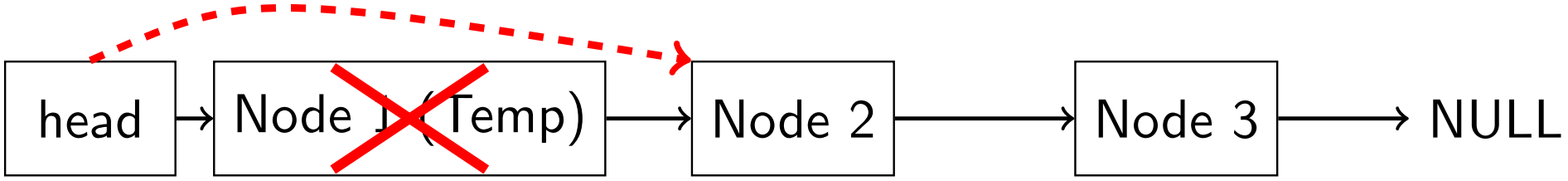
Deleting the First Node: Algorithmic Steps

Head Deletion Logic

Step-by-step logic to delete the head node from a singly linked list.

- **Step 1:** Check if the list is empty (`head == NULL`). If so, execution terminates immediately (underflow error).
- **Step 2:** Create a temporary pointer `'temp'` that stores the current head node: `temp = head`.
- **Step 3:** Update the head pointer to refer to the second node: `head = head->next`.
- **Step 4:** Safe deallocation: release the memory allocated to the node pointed by `'temp'` using `delete` or `free`.
- **Complexity:** This operation runs in $\mathcal{O}(1)$ time complexity as it requires no traversal.

Visualizing Head Deletion



- The `head` pointer originally points to Node 1.
- The `head` pointer is updated to bypass Node 1 and point directly to Node 2.
- A temporary pointer is used to reference Node 1 during pointer update.
- Red cross lines indicate the node memory being freed from the heap.

Code Implementation: deleteFirst

```
void deleteFirst(Node*& head) {
    if (head == nullptr) return;
    Node* temp = head;
    head = head->next;
    delete temp;
}
```

- The parameter `head` is passed by reference (`Node*&`) to allow the function to modify the caller's head variable.
- The check `if (head == nullptr)` protects against dereferencing null pointers.
- In a single-element list, `head->next` is `nullptr`, which correctly updates `head` to `nullptr` upon deletion.

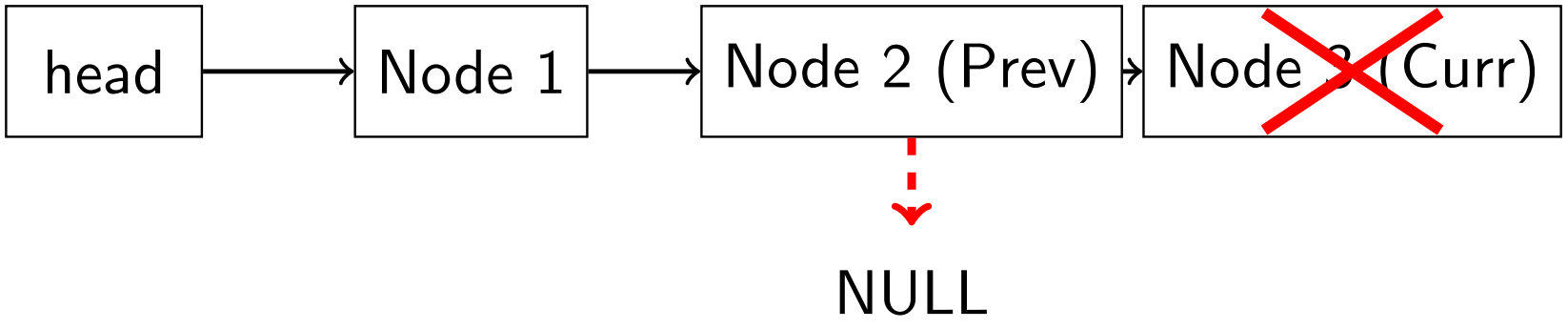
Deleting the Last Node: Algorithmic Steps

Tail Deletion Logic

Step-by-step logic to delete the tail node from a singly linked list.

- **Step 1:** Check if the list is empty (`head == NULL`). If true, return immediately (underflow).
- **Step 2:** Check if the list has a single node (`head->next == NULL`). If true, delete head, set `head = NULL`, and return.
- **Step 3:** Initialize two pointers: `'curr'` pointing to head, and `'prev'` pointing to `NULL`.
- **Step 4:** Traverse the list until `curr->next` is `NULL`. Update `prev = curr` and `curr = curr->next` at each step.
- **Step 5:** Set `prev->next = NULL` to sever the link to the last node, then delete `'curr'` to free memory.

Visualizing Tail Deletion



- The pointer `'curr'` traverses all the way to Node 3 (the tail).
- The pointer `'prev'` lags behind, pointing to Node 2 (the penultimate node).
- Node 2's next field is modified to point to `NULL` instead of Node 3.
- Memory allocated to Node 3 is successfully deallocated.

Code Implementation: deleteLast

```
void deleteLast(Node*& head) {
    if (head == nullptr) return;
    if (head->next == nullptr) {
        delete head;
        head = nullptr;
        return;
    }
    Node* prev = nullptr;
    Node* curr = head;
    while (curr->next != nullptr) {
        prev = curr;
        curr = curr->next;
    }
    prev->next = nullptr;
    delete curr;
}
```

Complexity Analysis & Comparison

Performance Comparison

Comparing `deleteFirst` and `deleteLast` across different list structures.

- **deleteFirst:** Has a time complexity of $\mathcal{O}(1)$ in all configurations because the head node is always directly accessible.
- **deleteLast (Singly Linked List):** Has a time complexity of $\mathcal{O}(N)$ since traversing to the penultimate node is required.
- **deleteLast (Singly Linked List with Tail pointer):** Still $\mathcal{O}(N)$, because having a tail pointer does not help us access the predecessor node.
- **deleteLast (Doubly Linked List with Tail pointer):** Can be optimized to $\mathcal{O}(1)$ time by leveraging `tail->prev` to bypass the final node.
- **Space Complexity:** $\mathcal{O}(1)$ auxiliary space for both operations

Summary & Best Practices

Key Takeaways

Key takeaways for implementing safe node deletion operations.

- Order of operations is critical: copy the node address before overwriting the pointer that references it.
- Always deallocate memory when removing a node to prevent long-term memory leaks in your programs.
- Check your pointers: dereferencing head or `head->next` without null validation leads to runtime segmentation faults.
- Design trade-offs: choose between singly and doubly linked lists based on how frequently tail deletion is performed.

Data Structures (DI03000021)

GTU Diploma Engineering

Structure of a Singly Linked List Node

Singly Linked List Node

A node in a singly linked list contains two main parts: a data field to store the element, and a next pointer to reference the subsequent node in the sequence.

```
struct Node {
    int data;
    Node* next;
    Node(int val) : data(val), next(nullptr) {}
};
```

- The data field can store any standard data type or object.
- The next pointer points to the address of the next Node object in memory.
- A `nullptr` or `NULL` pointer in the next field of the last node designates the tail/end of the list.

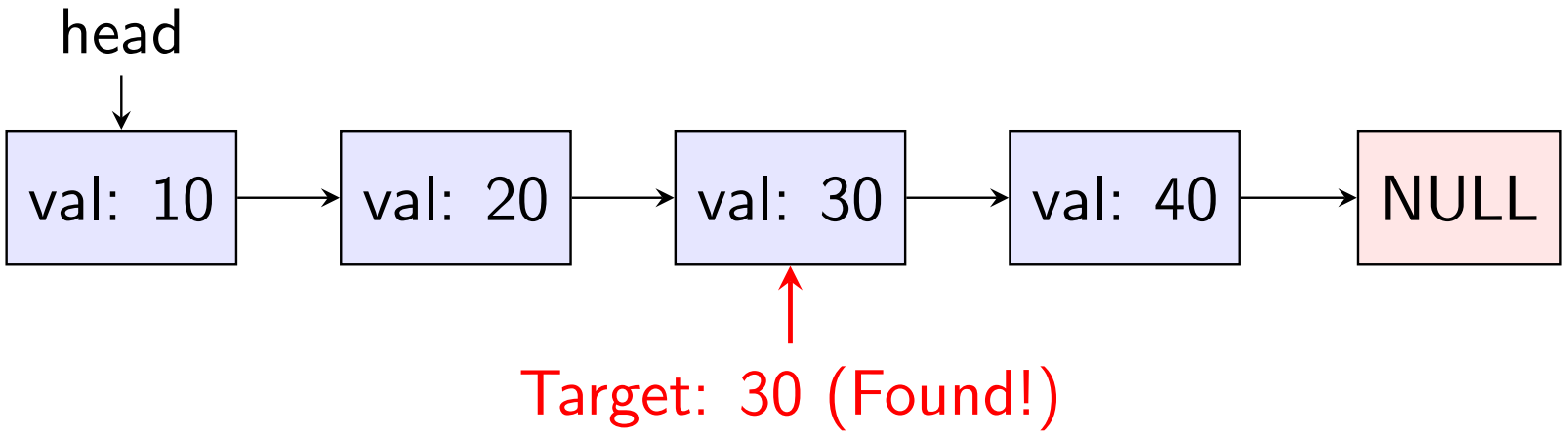
Searching a Node: Conceptual Approach

Search Concept

Searching in a linked list involves traversing the list from the head node down to the tail, comparing the target value against the data field of each node. Since we cannot perform random access (like `array[i]`), we must inspect nodes sequentially using a pointer variable.

- **Initialization:** Create a temporary pointer variable, usually named `current` or `temp`, initialized to point to the head node.
- **Traversal Loop:** While the `current` pointer is not `NULL`, check if `current->data` is equal to the target key.
- **Match Found:** If there is a match, return the node's pointer or its index (position).
- **No Match:** Move `current` to `current->next` and repeat. If `current` becomes `NULL`, the key is not present in the list.

Visualizing the Search Process



- The search starts at the head (value 10) and follows next pointers.
- At value 20, comparison fails, pointer advances to node 30.
- At node 30, `current->data == target (30)`, search terminates successfully.
- If the target were 50, the traversal would reach `NULL`, indicating a search miss.

C++ Implementation of Search Function

```
bool searchNode(Node* head, int key) {
    Node* current = head;
    while (current != nullptr) {
        if (current->data == key) {
            return true;
        }
        current = current->next;
    }
    return false;
}
```

- Iterative search function returns boolean `true` if the key is found, `false` otherwise.
- Always check for edge cases: an empty list (`head == nullptr`) will immediately return `false`.
- The assignment `current = current->next` changes the pointer's location to the next node.

Search Complexity Analysis

Complexity Analysis

Time Complexity:

- Best Case: $\mathcal{O}(1)$ — when the target node is the head of the list.
- Worst Case: $\mathcal{O}(n)$ — when the target is at the tail or not present.
- Average Case: $\mathcal{O}(n)$ — when the target is in the middle of the list.

Space Complexity:

- Iterative: $\mathcal{O}(1)$ auxiliary space.
- Recursive: $\mathcal{O}(n)$ space due to recursion call stack frame allocation.

- Since memory addresses are non-contiguous, binary search

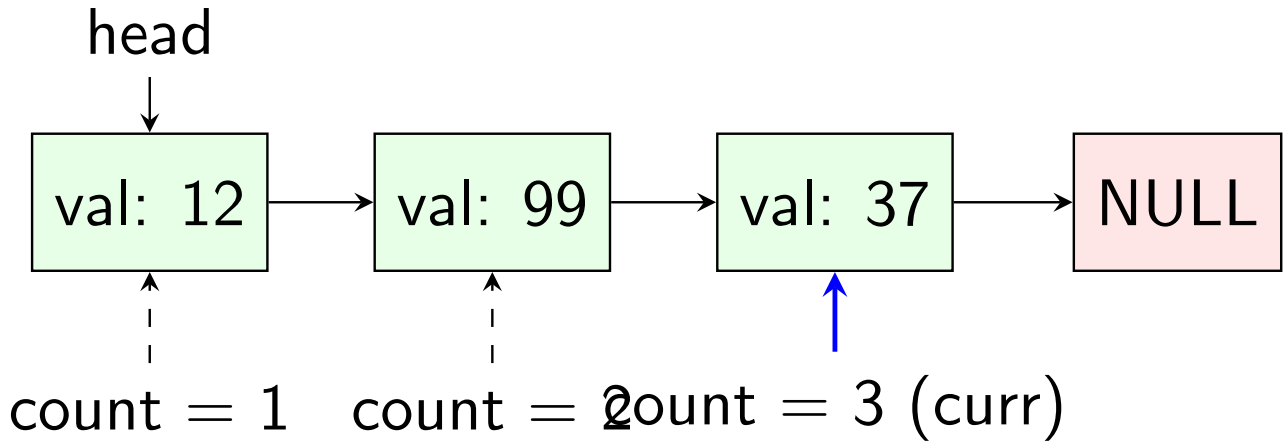
Counting Nodes in a Linked List

Counting Concept

To count the total number of nodes in a linked list, we must traverse the list from the head to the tail, incrementing a counter variable at each step until we reach the terminating null pointer.

- Initialize a local integer variable `count` to zero.
- Initialize a traversal pointer `current` to the `head` pointer.
- In a loop, while `current` is not `nullptr`, increment `count` by 1 and advance `current` to `current->next`.
- Once `current` equals `nullptr`, exit the loop and return the accumulated `count`.

Visualizing Traversal for Counting



- At node 1 (val: 12), count is updated to 1.
- At node 2 (val: 99), count is incremented to 2.
- At node 3 (val: 37), count is incremented to 3.
- When current reaches `NULL`, the count of 3 is returned, indicating 3 nodes exist in the list.

C++ Code for Counting Nodes

```
int countNodes(Node* head) {
    int count = 0;
    Node* current = head;
    while (current != nullptr) {
        count++;
        current = current->next;
    }
    return count;
}
```

- An empty list (`head == nullptr`) will skip the while loop entirely and return 0, which is correct.
- Time Complexity is $\mathcal{O}(n)$ because we must touch every node exactly once to count it.
- Space Complexity is $\mathcal{O}(1)$ for the iterative implementation.
- Recursive count option: `return (head == nullptr) ? 0 : 1 + countNodes(head->next);`

Summary of Operations and Complexities

Summary

Both searching and counting in a linked list require complete traversal of the active nodes.

- **Search Time:** $\mathcal{O}(n)$ worst-case, $\mathcal{O}(1)$ best-case.
- **Count Time:** $\mathcal{O}(n)$ always, unless tracking list size as a class property.
- **Space Complexity:** $\mathcal{O}(1)$ for both iterative approaches.

- Tracking size as a member variable in a Wrapper Linked List class changes the count complexity to $\mathcal{O}(1)$.
- Unlike arrays, neither search nor size calculations can utilize indexing offset calculations.
- Proper null checks at every step are essential to prevent runtime segmentation faults or null-pointer dereferences.
- Understanding these baseline linear operations is critical for

Data Structures (DI03000021)

GTU Diploma Engineering

Linear vs. Hierarchical Structures

Overview

Linear structures like arrays, linked lists, stacks, and queues establish a sequential 1-to-1 relationship. Trees represent hierarchical 1-to-many relationships, structuring data in nested levels.

- In a linear structure, every element (except first/last) has a unique predecessor and successor.
- In a hierarchical structure, a node has a single predecessor (parent) but can have multiple successors (children).
- Hierarchical representations are essential for modeling directories, organization charts, XML/HTML documents, and decision processes.

Formal Definition of a General Tree

General Tree

A General Tree T is a finite, non-empty set of nodes establishing a hierarchical parent-child relation.

- Formal Definition: A tree T is partitioned into $1 + k$ disjoint subsets, where $k \geq 0$.
- One subset contains a single unique node r, designated as the root of T.
- The remaining k subsets are trees themselves: $T_1, T_2, \ldots, T_k$, which are called the subtrees of r.
- Unlike binary trees, a general tree imposes no constraint on the maximum number of subtrees (out-degree) a node can have.

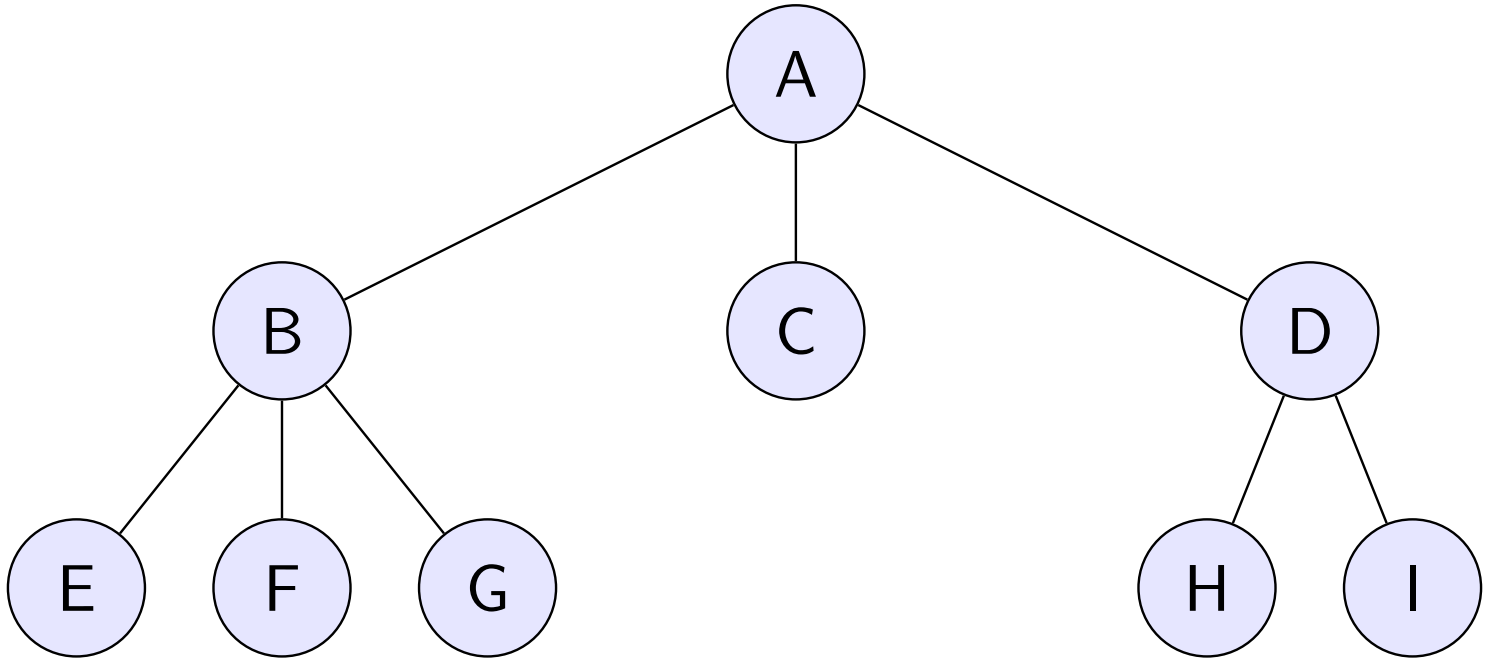
Anatomy of a Tree: Key Terminology

Terminology

To analyze and traverse trees, we define standard terms for nodes, edges, and subtrees.

- **Root**: The top node in a tree with no parent. Every non-empty tree has exactly one root.
- **Child & Parent**: If node u connects directly to node v (moving away from root), u is the parent and v is the child.
- **Siblings**: Nodes that share the same parent node.
- **Leaf Node (External Node)**: A node with no children (degree of 0).
- **Internal Node**: A node with one or more children (degree > 0).

Visualizing a General Tree Structure



- A is the root node of the tree, possessing three children: B, C, and D.
- B, C, and D are siblings. C is a leaf node, whereas B and D are internal nodes.
- The subtree rooted at B has three leaf nodes: E, F, and G.
- The tree contains 9 nodes and 8 edges, demonstrating the property of $E = N - 1$.

Structural Properties and Math Metrics

Properties

Mathematical guidelines govern the structures of all general trees.

- **Property 1**: A general tree containing N nodes always contains exactly $N - 1$ edges.
- **Depth of a node** v: The length of the unique path from the root to v. Depth(root) $= 0$.
- **Height of a node** v: The length of the longest path from v to a leaf. Height(leaf) $= 0$.
- **Height of a tree** T: The maximum depth of any node in T, which equals the height of the root node.
- **Path**: A sequence of nodes $v_1, v_2, \ldots, v_k$ such that v_i is the parent of v_{i+1}.

Left-Child Right-Sibling Representation

LCRS Representation

Representing nodes with a variable number of child pointers is inefficient. The Left-Child Right-Sibling (LCRS) representation maps any general tree into a binary tree structure.

- Each node in the LCRS representation contains only two pointers: `leftChild` and `rightSibling`.
- `leftChild` points to the first/leftmost child of the node in the general tree.
- `rightSibling` points to the immediate sibling to its right.
- Memory efficiency: Standardizes node structure to C++: `struct Node { Data val; Node* leftChild; Node* rightSibling; };`

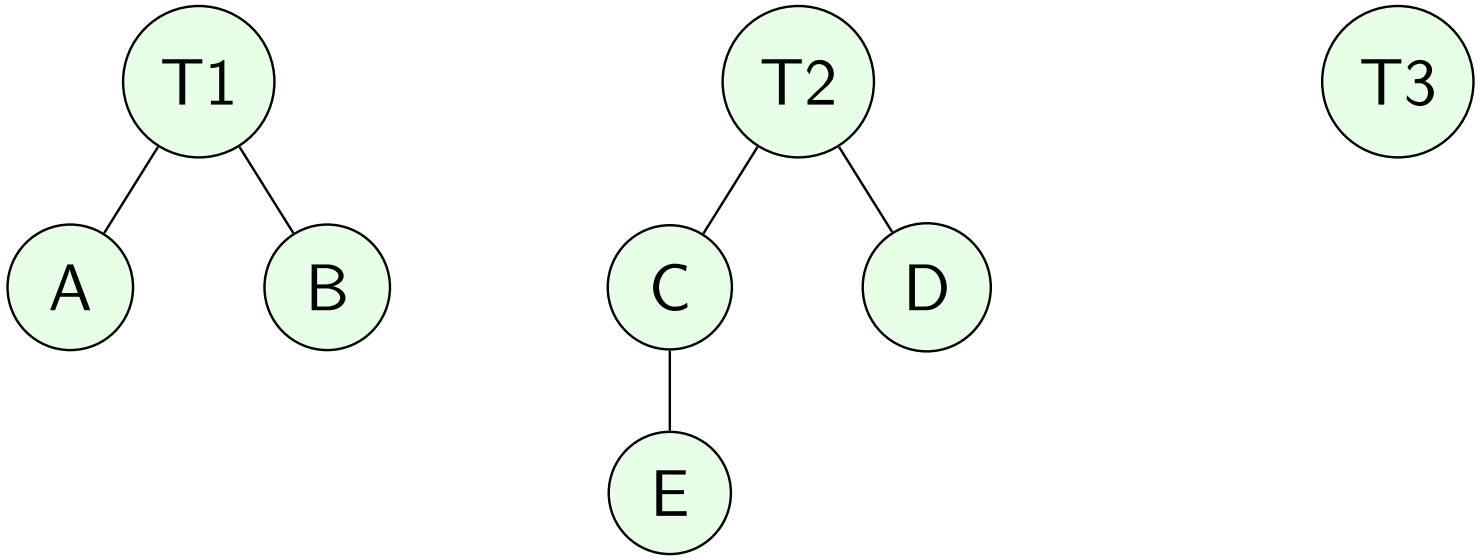
Forest: Definition and Conversion

Forest

A Forest is an ordered set of zero or more disjoint trees.

- Removing the root node from a general tree yields a forest consisting of the root's subtrees.
- Conversely, any forest can be transformed into a single general tree by introducing a new dummy root node and setting the roots of the forest's trees as children of this new root.
- Forest traversals: Pre-order and Post-order traversals are defined recursively over the sequence of constituent trees.

Visualizing a Forest of Disjoint Trees



- This forest consists of three disjoint trees: T_1, T_2, and T_3.
- T_1 has two leaf children (A and B). T_2 has child C (with grandchild E) and child D.
- T_3 is a single-node tree representing a root-only hierarchy.
- The disjoint union of these trees behaves as a forest before any root merger occurs.

Summary of Core Concepts

Summary

A summary of general tree concepts, forest topologies, and node attributes.

- A general tree represents a hierarchy with a single root and arbitrary degrees per node.
- Trees always contain $N - 1$ edges and are connected, acyclic graphs.
- LCRS (Left-Child Right-Sibling) allows representing general trees uniformly using binary-like nodes.
- A forest is a set of disjoint trees that can be converted into a general tree by adding a single parent node.

Data Structures (DI03000021)

GTU Diploma Engineering

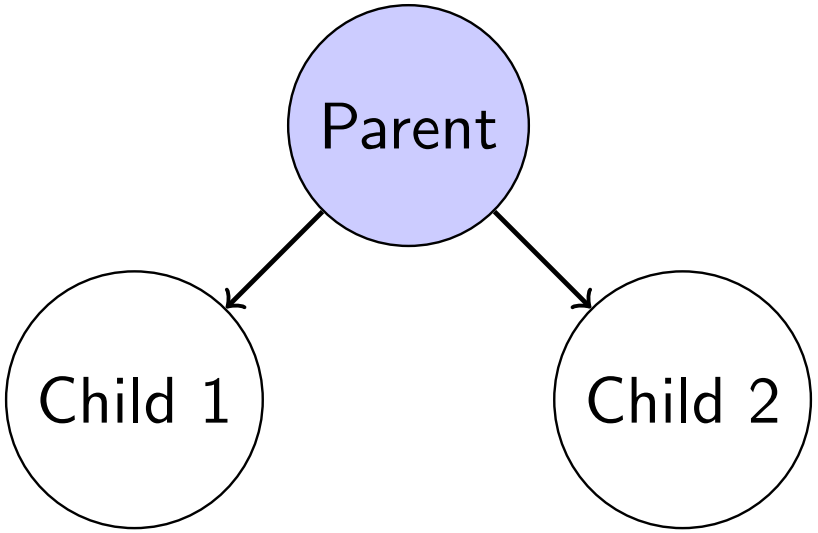
Introduction to Tree Terminology

Tree Terminology

A tree is a non-linear, hierarchical data structure consisting of nodes connected by directed or undirected edges. Unlike linear structures (arrays, linked lists), a tree has a single entry point called the root and does not contain cycles.

- **Node**: The fundamental element containing data and references/pointers to other nodes.
- **Edge**: The connection representing the parent-child relationship between two nodes.
- **Degree**: The number of subtrees or children connected to a specific node.
- **Height of a Tree**: The length of the longest path from the root to a leaf node.

The Concept of a Parent Node



- A parent node is any node (except the root) that has a directed edge pointing to another node.
- The root node has no parent node, representing the top of the hierarchy.
- In a binary tree, a parent can have at most two children: left child and right child.
- In a general tree, a parent node can have any number of children (bounded by the node's degree).

Algorithm to Find a Parent Node

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

Node* findParent(Node* root, Node* target) {
    if (root == NULL || root == target) return NU
    if (root->left == target || root->right == ta
    Node* leftSearch = findParent(root->left, tar
    if (leftSearch != NULL) return leftSearch;
    return findParent(root->right, target);
}
```

- To find the parent of a target node in a binary tree, we perform a recursive preorder or postorder traversal.
- Time Complexity: $O(N)$ in the worst case (unbalanced tree),

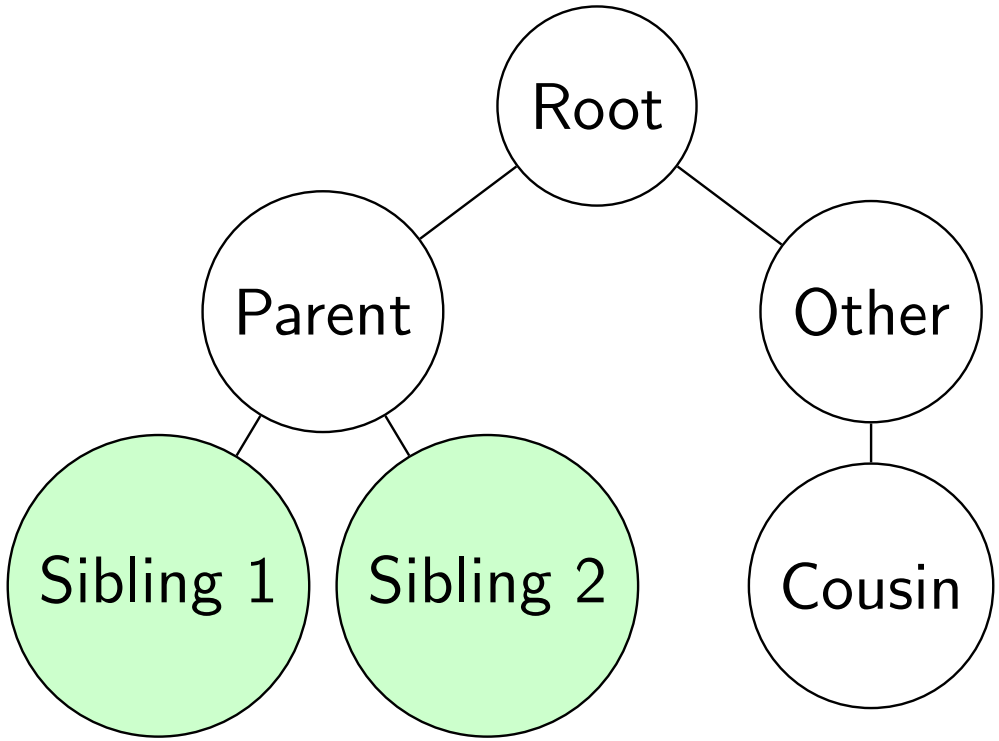
Sibling Nodes in a Tree

Sibling Nodes

Nodes that share the exact same parent node are defined as sibling nodes. Siblings are at the same depth level in the hierarchy, but the converse is not always true (nodes at the same depth are not necessarily siblings).

- Two nodes A and B are siblings if parent$(A) ==$ parent(B) and $A \neq B$.
- In a binary tree, a node can have at most one sibling.
- Sibling relationships are crucial for implementing tree rebalancing operations (e.g., in AVL or Red-Black trees).
- Cousins are nodes at the same level but with different parent nodes.

Visualizing Sibling Relationships



- Nodes ‘Sibling 1’ and ‘Sibling 2’ share ‘Parent’ as their parent, hence they are siblings.
- Node ‘Cousin’ is at the same level (depth 2) as Sibling 1 and Sibling 2, but has ‘Other’ as its parent.
- Understanding the difference between siblings and cousins is fundamental to operations like uncle-node checks in Red-Black Trees.

Leaf Nodes (External Nodes)

Leaf Node

A leaf node (or external node) is a node that does not have any children. It resides at the very bottom boundaries of the tree structure.

- Mathematical definition: A node u is a leaf if $\text{degree}(u) = 0$.
- Leaf nodes represent the termination of paths starting from the root node.
- In expression trees, leaf nodes store operands (numbers or variables) while internal nodes store operators.
- Leaf node count property: In a non-empty binary tree, the number of leaf nodes L is always equal to the number of nodes with two children plus one ($L = N_2 + 1$).

Coding Leaf Node Operations

```
bool isLeaf(Node* node) {
    if (node == NULL) return false;
    return (node->left == NULL && node->right ==
}

int countLeaves(Node* root) {
    if (root == NULL) return 0;
    if (isLeaf(root)) return 1;
    return countLeaves(root->left) + countLeaves(
}
```

- Checking leaf status requires validating that both `left` and `right` pointers are null pointers.
- The recursive leaf counting algorithm runs in $O(N)$ time as it must visit every node exactly once.
- The recursion depth is proportional to the tree height H, requiring $O(H)$ auxiliary space.

Complexity of Tree Operations

Complexity Overview

Analyzing search, insertion, and deletion complexity for parent, sibling, and leaf queries across different tree balances.

- **Finding Sibling**: $O(1)$ if parent pointer is present, otherwise $O(N)$ in a general binary tree.
- **Leaf Nodes Search**: $O(N)$ traversal is required to find all leaf nodes since they are distributed across the tree boundaries.
- In a Binary Search Tree (BST), finding the parent of a target node during insertion takes $O(\log N)$ average time, $O(N)$ worst-case.
- Maintaining explicit parent references increases memory by one pointer per node, but simplifies backward traversals.

Summary and Key Takeaways

Summary

Reviewing the fundamental tree nodes: Parent, Sibling, and Leaf, and their practical implications in computer science.

- **Parent**: The immediate ancestor node. Root has no parent. Important for tracking lineages.
- **Sibling**: Nodes that share the same parent node. Crucial for self-balancing trees.
- **Leaf**: A terminal node with no child nodes (degree 0). Represents the base case for recursive traversals.
- Understanding these relationships forms the foundation for advanced graph and tree algorithms like Heap, Segment Trees, and B-Trees.

Data Structures (DI03000021)

GTU Diploma Engineering

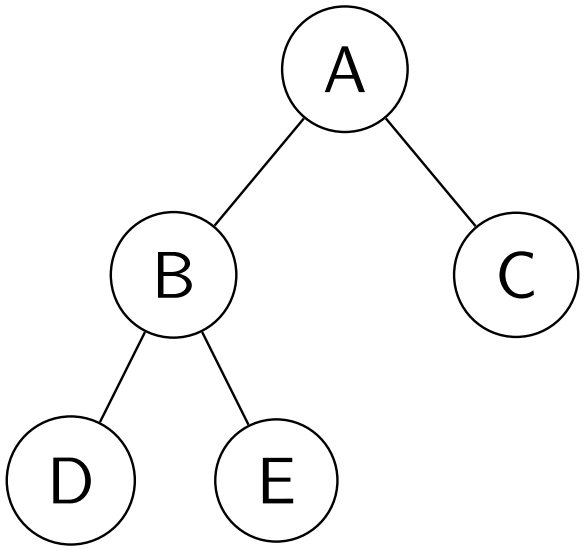
Defining Node Degree in Undirected Graphs

Degree of a Vertex in an Undirected Graph

The degree of a vertex v in an undirected graph, denoted as $\deg(v)$, is the number of edges incident to v.

- The degree of a vertex v in an undirected graph, denoted as $\deg(v)$, is the number of edges incident to v.
- A self-loop on a vertex contributes 2 to the degree of that vertex.
- Vertices of degree 0 are called isolated vertices (having no incident edges).
- Vertices of degree 1 are called pendant vertices or leaf vertices (connected to exactly one neighbor).

Node Degree in Tree Structures



- In rooted trees, the degree of a node is defined as the number of its children (not its total incident edges).
- The degree of the tree is the maximum degree of any node in that tree (e.g., a Binary Tree has a maximum degree of 2).
- In the diagram, Node A has degree 2 (children B, C), Node B has degree 2 (children D, E), and Nodes C, D, E have degree 0 (leaves).
- Note: If treated as an undirected graph, the root A would have degree 2, internal node B would have degree 3 (1 parent + 2 children), and leaves would have degree 1.

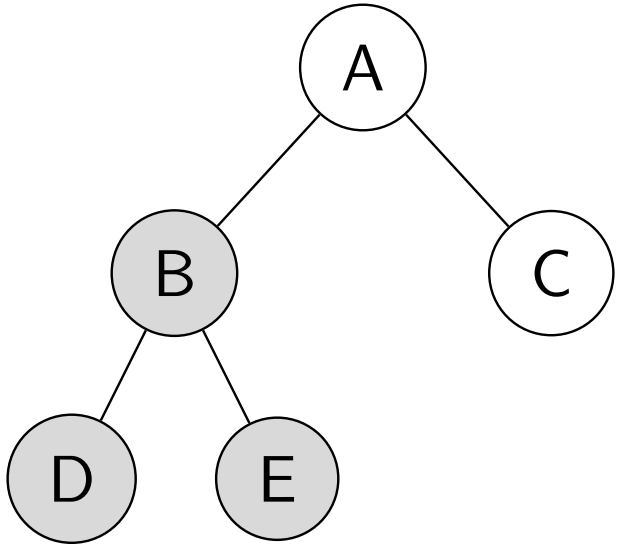
The Concept of Subtrees

Subtree Definition

A subtree is a tree structure consisting of a node in a tree and all of its descendants. Formally, for a tree $T = (V, E)$ and a node $r \in V$, the subtree rooted at r, denoted T_r, contains r as its root and all nodes v such that the path from the overall root of T to v passes through r.

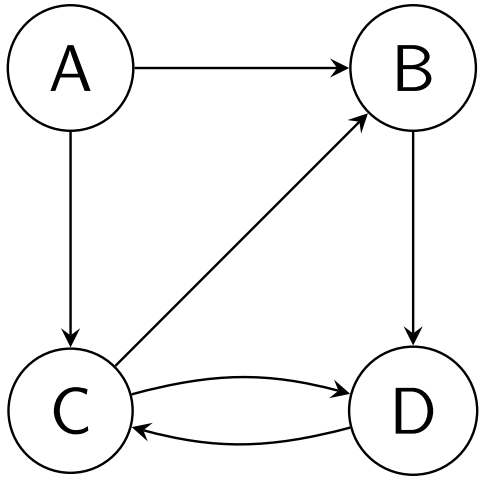
- A subtree is a tree structure consisting of a node in a tree and all of its descendants.
- Formally, for a tree $T = (V, E)$ and a node $r \in V$, the subtree rooted at r, denoted T_r, contains r as its root and all nodes v such that the path from the overall root of T to v passes through r.
- Subtrees are recursive: every node in a tree is the root of its own subtree.
- Recursive algorithms (like DFS, tree traversals, height

Visualizing Subtrees in a Binary Tree



- The shaded nodes (B, D, E) form the subtree T_B rooted at node B.
- Node B serves as the local root for this subtree, behaving exactly like an independent tree.
- Recursive algorithms process T_B independently before propagating the results back to parent node A.
- In binary search trees (BST), the subtree T_B containing left children contains values strictly less than A, while T_C contains values strictly greater.

Directed Graphs: In-Degree and Out-Degree



- In a directed graph (digraph), edges have a direction, meaning degree is split into two components.
- In-degree of a vertex v, denoted $\deg^-(v)$, is the number of incoming edges to v.
- Out-degree of a vertex v, denoted $\deg^+(v)$, is the number of outgoing edges from v.
- In the diagram: $\deg^-(A) = 0$ (source node), $\deg^+(A) = 2$. $\deg^-(B) = 2$, $\deg^+(B) = 1$. $\deg^-(C) = 2$, $\deg^+(C) = 2$. $\deg^-(D) = 2$, $\deg^+(D) = 1$.

The Handshaking Lemma

Fundamental Theorem of Graph Theory

For any undirected graph $G = (V, E)$, the sum of the degrees of all vertices is twice the number of edges:

$$\sum_{v \in V} \deg(v) = 2|E|$$

- For any undirected graph $G = (V, E)$, the sum of the degrees of all vertices is twice the number of edges: $\sum_{v \in V} \deg(v) = 2|E|$.
- Corollary: Every undirected graph contains an even number of vertices of odd degree.
- For a directed graph $G = (V, E)$, the sum of the in-degrees equals the sum of the out-degrees, which equals the total number of edges: $\sum_{v \in V} \deg^-(v) = \sum_{v \in V} \deg^+(v) = |E|$.
- This conservation property is central to network flow

Algorithmic Representation and Complexity

- Adjacency Matrix: Storing a graph as a $V \times V$ matrix. Finding the degree of v requires checking an entire row/column: $O(V)$ time complexity.
- Adjacency List: Storing a graph as an array of lists. Out-degree of v is the size of its list: $O(1)$ time if size is cached, or $O(\deg^+(v))$ by traversal.
- To find the in-degree of all vertices in an adjacency list representation, we must scan all adjacency lists, taking $O(V + E)$ time unless an explicit in-degree array is maintained.
- Maintaining an in-degree array dynamically allows $O(1)$ lookup at the cost of updates during edge insertions/deletions ($O(1)$ extra cost).

Implementation of Degree Counting

- Let `adj[u]` be the list of outgoing neighbors from `u`. We want to construct an in-degree array.
- Algorithm:
  1. Initialize `in_degree` array of size V with 0.
  2. Initialize `out_degree` array of size V with 0.
  3. For each vertex `u` from 0 to $V - 1$:
     - `out_degree[u] = adj[u].size()`
     - For each neighbor `v` in `adj[u]`: `in_degree[v]++`
- Time Complexity: $O(V + E)$ since we visit every vertex once and traverse every edge exactly once.
- Space Complexity: $O(V)$ to store the in-degree and out-degree arrays.

Summary and Practical Applications

- Tree Degree: Measures maximum children per node. Used to define binary, ternary, and B-trees.
- Subtrees: Crucial for divide-and-conquer algorithms, dynamic programming on trees, and tree rotations in self-balancing trees (AVL, Red-Black).
- Graph Degrees: In-degree and Out-degree are essential for Topological Sorting (Kahn's Algorithm relies directly on in-degrees).
- Handshaking Lemma: Guarantees fundamental invariant checks for graph construction and validation.

Data Structures (DI03000021)

GTU Diploma Engineering

Introduction to Tree Hierarchies

Tree Hierarchy

A tree T is an acyclic connected graph. In a rooted tree, we designate one vertex as the root, which establishes a parent-child hierarchy. Understanding levels, depth, and paths is fundamental for analyzing tree-based algorithms like BFS, DFS, and AVL tree balancing.

- A tree with N nodes has exactly $N - 1$ edges.
- The root node is the unique node with no parent.
- Every node except the root has a unique parent.
- Descendants and ancestors are defined relative to paths from the root.

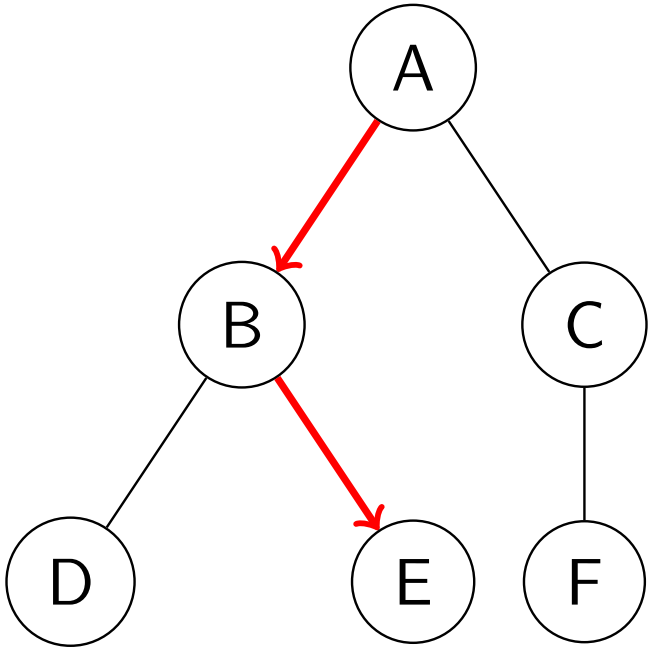
Formal Definition of a Path

Path in a Tree

A path in a tree is a sequence of distinct vertices $v_0, v_1, \ldots, v_k$ such that (v_i, v_{i+1}) is an edge for all $0 \leq i < k$. The length of a path is the number of edges it contains, which is k.

- In a tree, there is a unique simple path between any two vertices.
- The path from the root to any node v defines v's ancestors.
- Path length is measured in terms of edges, not vertices (a path with k edges has $k + 1$ vertices).
- The height of a node is the length of the longest downward path to a leaf.

Visualizing Paths and Depth



- The path A $\rightarrow$ B $\rightarrow$ E has length 2 (edges: (A,B) and (B,E)).
- Depth of A is 0, depth of B and C is 1, depth of D, E, F is 2.
- TikZ shows the hierarchical tree structure with highlighted path in red.
- The number of edges on the path from root to node equals the node's depth.

Understanding Node Depth

Node Depth

The depth of a node v in a rooted tree is the length of the unique path from the root to v. It represents how far down the tree the node resides relative to the starting point.

- The depth of the root node is always 0.
- The depth of any child node c of parent p is $\text{depth}(c) = \text{depth}(p) + 1$.
- Depth is calculated recursively during traversals (e.g., in Pre-order traversal).
- Finding depth of a node takes $O(d)$ time where d is its depth, or $O(N)$ worst-case.

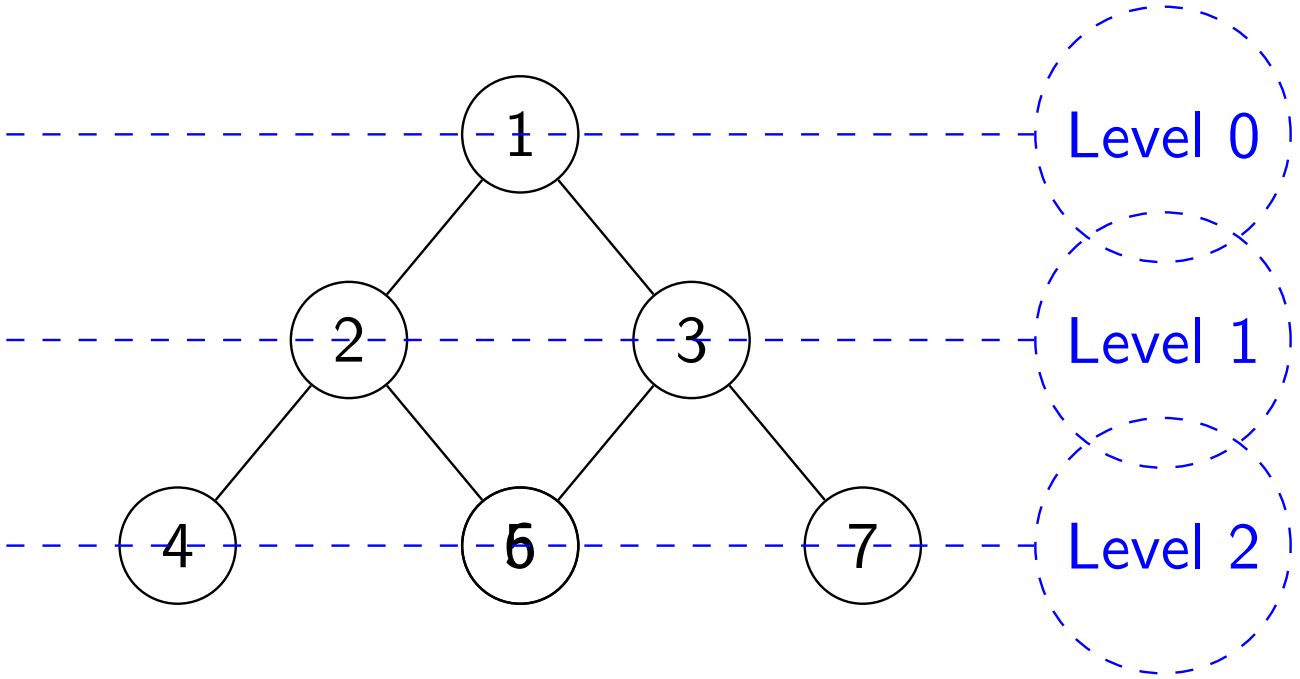
Defining Levels in Trees

Tree Levels

The level of a node is equivalent to its depth. The set of all nodes at a given depth d constitutes Level d of the tree. Level-order traversal processes nodes level by level from top to bottom.

- Level 0 contains only the root node.
- Level i contains all nodes at distance i from the root.
- The maximum level in a tree of size N is $N - 1$ (skewed tree) and minimum is $\lfloor \log_2 N \rfloor$.
- Breadth-First Search (BFS) is typically used for level-order processing using a Queue data structure.

Level-Order Graph Representation



- Nodes 2 and 3 are on Level 1; nodes 4, 5, 6, and 7 are on Level 2.
- Level-order traversal visits nodes in the order: 1, 2, 3, 4, 5, 6, 7.
- A queue is used to enqueue children as each parent is dequeued.
- Time complexity of Level-Order traversal is $O(N)$ since each node is visited once.

Recursive Algorithm for Depth & Height

Algorithm: Computing Depth

```
depth(node, target, current_depth):
    if node is null: return -1
    if node == target: return current_depth
    left = depth(node.left, target, current_depth + 1
    if left != -1: return left
    return depth(node.right, target, current_depth +
```

- This algorithm performs a depth-first search tracking depth in a parameter.
- Space complexity is $O(H)$ where H is the tree height, due to call stack overhead.
- Time complexity is $O(N)$ in the worst case as we might visit all nodes.
- Height can be computed as:

Practical Applications in Computer Science

Applications

Level and depth concepts are vital for:

1. Balanced trees (AVL, Red-Black) which keep height $O(\log N)$,
2. Lowest Common Ancestor (LCA) algorithms using binary lifting,
3. Computer Graphics where scene graphs represent objects hierarchically.

- In AVL trees, the balance factor of node N is $\text{height}(L) - \text{height}(R)$.
- LCA algorithms often align node depths first before traversing upwards.
- Network routing algorithms use hop counts (path length) to optimize packets.

Summary of Key Tree Metrics

Summary of Metrics

We covered: Path (sequence of edges connecting nodes), Depth (number of edges from root to node), Level (set of nodes at the same depth). Understanding these is essential for efficient graph theory and database indexing structures.

- Path length equals the number of edges, not nodes.
- Depth is top-down (root is 0); height is bottom-up (leaves are 0).
- Level-order traversal maps directly to Breadth-First Search (BFS).
- These metrics determine the complexity of search, insertion, and deletion operations.

Data Structures (DI03000021)

GTU Diploma Engineering

Fundamental Tree Definitions: Nodes, Paths, and Levels

A

tree $T = (V, E)$ is an acyclic connected graph. Properties are defined relative to a designated root node r.

- **Path:** A sequence of distinct nodes where each consecutive pair is connected by an edge. Path length is the number of edges.
- **Depth of a node u:** The length of the unique path from the root r to u. By definition, $\text{depth}(r) = 0$.
- **Level of a node u:** Defined as $\text{depth}(u) + 1$, representing the generation tier of the node within the hierarchy.
- **Height of a node u:** The length of the longest downward path from u to a leaf descendant.
- **Height of Tree T:** The height of the root node r, which is

Computing Tree Height Recursively

```
int height(Node* node) {
    if (node == nullptr) {
        return -1;
    }
    int leftHeight = height(node->left);
    int rightHeight = height(node->right);
    return 1 + std::max(leftHeight, rightHeight);
}
```

- **Base Case:** If the node is null, it represents an empty subtree, returning a height of -1.
- **Recurrence Relation:** The height of node u is 1 plus the maximum height of its left and right subtrees.
- **Time Complexity:** $\mathcal{O}(N)$ where N is the number of nodes, as every node is visited exactly once.
- **Space Complexity:** $\mathcal{O}(H)$ where H is the height of the tree, representing the maximum call stack frames. Worst case $\mathcal{O}(N)$

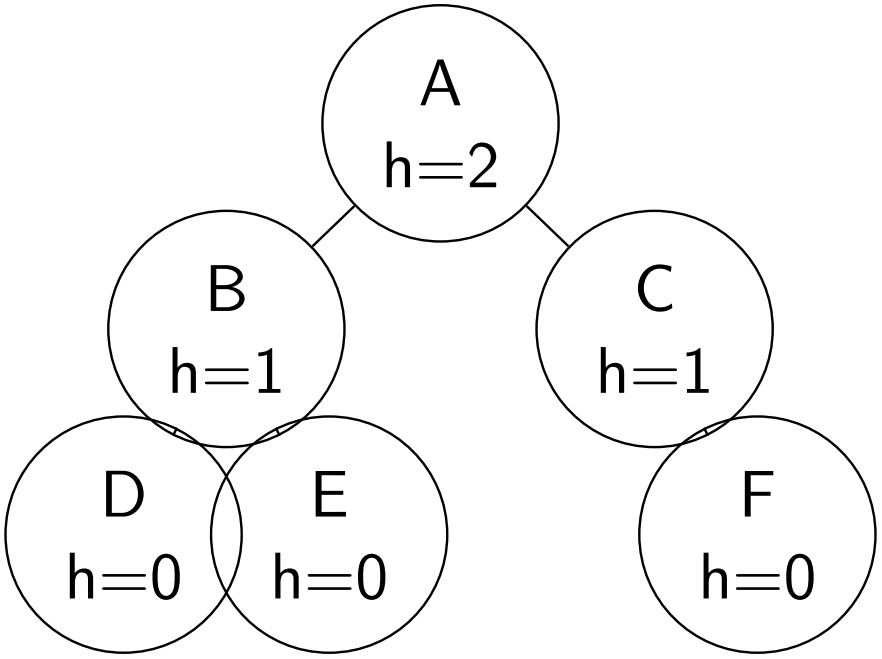
Binary Trees: Structural Properties and Constraints

A

Binary Tree is a tree where each node has at most two children, designated as left and right.

- **Maximum nodes at level L:** A binary tree can have at most 2^L nodes at depth L.
- **Maximum total nodes for height H:** A perfect binary tree of height H contains exactly $2^{H+1} - 1$ nodes.
- **Minimum total nodes for height H:** A degenerate/skewed binary tree of height H contains exactly $H + 1$ nodes.
- **Height Bounds:** For any binary tree with N nodes, the height H satisfies the inequality: $\lfloor \log_2(N) \rfloor \leq H \leq N - 1$.
- **Leaf Node Relation:** In any non-empty binary tree, if n_0 is the number of leaf nodes and n_2 is the number of nodes with two children, then $n_0 = n_2 + 1$.

Visualizing a Binary Tree and Node Heights



- Root node A has height 2, as the longest path to a leaf is A $\rightarrow$ B $\rightarrow$ D (length 2).
- Node B has height 1, derived from $1 + \max(\text{height}(D), \text{height}(E)) = 1 + 0 = 1$.
- Node C has height 1, derived from $1 + \max(\text{height}(\text{NULL}), \text{height}(F)) = 1 + \max(-1, 0) = 1$.
- Leaf nodes D, E, and F have height 0 because they have no children (left and right subtrees are NULL).

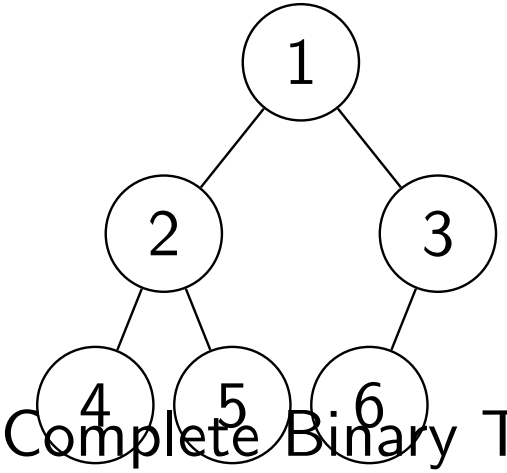
The Complete Binary Tree

A

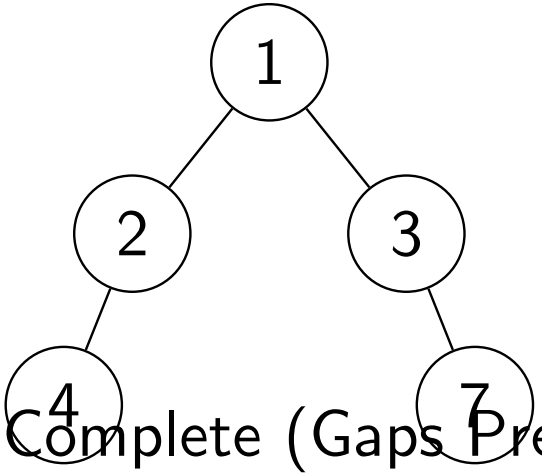
Complete Binary Tree is a binary tree that is completely filled at all levels except possibly the lowest, which is filled from left to right.

- **Strict filling:** Every level $d < H$ contains exactly 2^d nodes, leaving no gaps in the upper structure.
- **Left alignment:** At the bottom level H, all nodes are positioned as far left as possible.
- **Height Guarantee:** A complete binary tree of N nodes has height $H = \lfloor \log_2 N \rfloor$, ensuring logarithmic height for all operations.
- **Balanced nature:** The difference in height between the left and right subtrees of any node is at most 1, making it automatically balanced.
- **Use cases:** Serves as the structural foundation for binary

Comparing Complete and Non-Complete Binary Trees



Complete Binary Tree



NOT Complete (Gaps Present)

- **Left Tree (Complete):** Level 0 and 1 are full. Level 2 is filled from left to right (indices 4, 5, 6) with no gaps.
- **Right Tree (Non-Complete):** Gaps exist because node 2 has no right child while node 3 has a right child 7 without a left child.

Array-Based Representation of Complete Binary Trees

A

Complete Binary Tree of size N can be stored compactly in a 1D array without explicit pointers.

- **Zero-based indexing:** The root node is stored at index 0.
- **Left Child Index:** For a parent node at index i, its left child is located at index $2i + 1$.
- **Right Child Index:** For a parent node at index i, its right child is located at index $2i + 2$.
- **Parent Index:** For a child node at index i (where $i > 0$), its parent is located at index $\lfloor (i - 1)/2 \rfloor$.
- **Spatial Locality:** Storing elements contiguously in memory eliminates pointer overhead (8 bytes per node on 64-bit systems) and improves CPU cache hit rates.

Validating Node Connections in Array Representation

```
bool hasLeftChild(int i, int n) {
    return (2 * i + 1) < n;
}

bool hasRightChild(int i, int n) {
    return (2 * i + 2) < n;
}

int getParent(int i) {
    if (i <= 0) return -1; // Root has no parent
    return (i - 1) / 2;
}
```

- **Index Limits:** Because the array size is N, indices range from 0 to $N-1$. Any computed child index $\geq N$ is invalid.
- **Leaf check:** Node i is a leaf if and only if $2i+1 \geq N$ (has no left child).

Key Takeaways: Height and Structural Variations

S

ummary of tree types and height characteristics: Balanced vs Skewed, Array vs Pointer.

- Height is the primary metric determining search, insertion, and deletion complexity: $\mathcal{O}(\log N)$ in balanced trees vs $\mathcal{O}(N)$ in skewed trees.
- Binary Trees restrict out-degree to 2, establishing standard algebraic bounds for node and leaf distribution.
- Complete Binary Trees enforce strict level-filling, guaranteeing logarithmic height and enabling pointer-free array representations.
- Array representations optimize storage and memory locality but are only efficient when the tree is complete, otherwise wasting space on empty slots.

Data Structures (DI03000021)

GTU Diploma Engineering

Definition of a Strict Binary Tree

Strict Binary Tree

A Strict Binary Tree (or Full Binary Tree) is a binary tree in which every node has either 0 or 2 children.

- No node in a strict binary tree can have exactly one child; all internal nodes must have two children.
- Formally, for any node v in a strict binary tree T, $\deg(v) \in \{0, 2\}$ where $\deg(v)$ is the degree (number of children) of node v.
- Often used in representing mathematical expressions (expression trees) where operators are internal nodes and operands are leaves.
- Contrast with a Complete Binary Tree (where all levels are filled except possibly the last, which is filled left-to-right).

Strict Binary Tree Properties

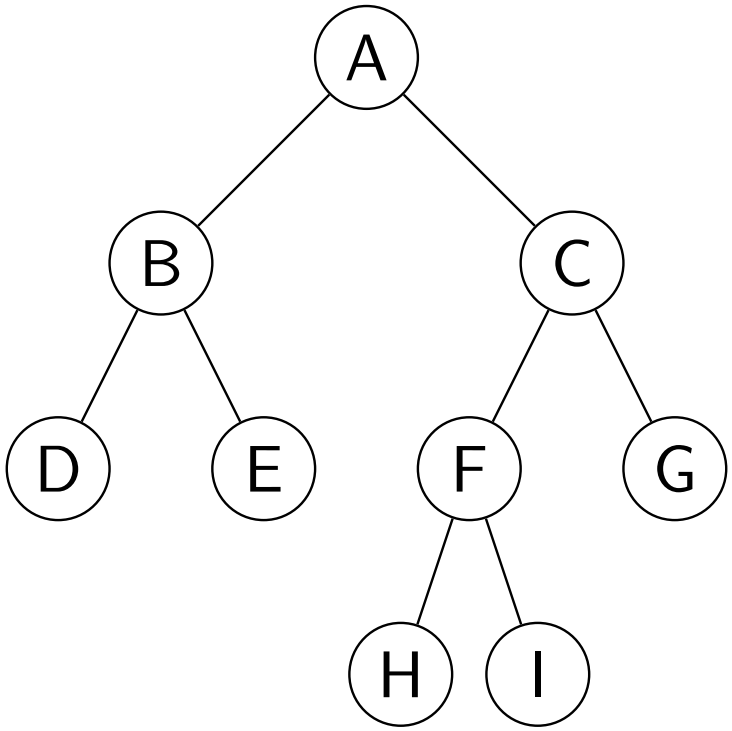
Key Property: Leaves vs. Internal Nodes

In any strict binary tree, the number of leaf nodes (L) is always one more than the number of internal nodes (I):

$$L = I + 1$$

- Let L be the number of leaf nodes (nodes with 0 children) and I be the number of internal nodes (nodes with 2 children).
- **Theorem:** In any strict binary tree, the number of leaf nodes is always one more than the number of internal nodes: $L = I + 1$.
- **Proof by Induction:** A tree with 1 node has $L = 1, I = 0$ $(1 = 0 + 1)$. Replacing a leaf node with an internal node and two leaves increases I by 1 and L by 1 $(-1 + 2 = +1)$. Thus, the relation holds.
- **Total Nodes:** $N = L + I = (I + 1) + I = 2I + 1$. Hence, the total number of nodes in a strict binary tree is always odd.

Diagram of a Strict Binary Tree



- This diagram illustrates a strict binary tree where every internal node (A, B, C, F) has exactly two children.
- Leaf nodes (D, E, H, I, G) have zero children.
- Notice that the number of internal nodes $I = 4$, and the number of leaf nodes $L = 5$. $L = I + 1$ holds.
- Total nodes $N = 9$, which is indeed an odd number.

General Trees vs. Binary Trees

Motivation for Conversion

Understanding structural limitations and representation challenges between General Trees and Binary Trees.

- A **General Tree** is a tree where each node can have an arbitrary, unrestricted number of children (zero or more).
- Representing general trees using standard node structures with variable-sized child arrays or linked lists is memory-inefficient and complex.
- A **Binary Tree** restricts the number of children per node to at most two (left and right child), allowing for a uniform representation.
- To store and process general trees efficiently, we often convert them into equivalent binary tree structures.

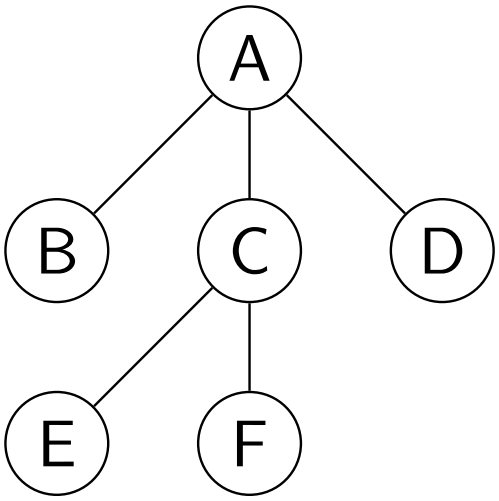
Conversion Algorithm: Left-Child Right-Sibling (LCRS)

Left-Child Right-Sibling (LCRS) Conversion

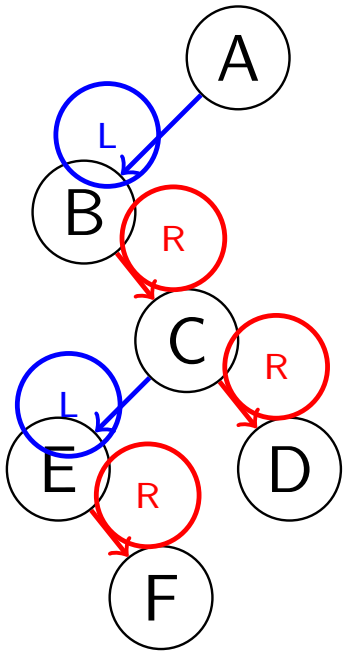
The standard representation technique to convert a General Tree into an equivalent Binary Tree structure.

- **Algorithm Rules:**
  1. For each node, its left pointer in the binary tree points to its first child (leftmost child) in the general tree.
  2. Its right pointer in the binary tree points to its immediate next sibling in the general tree.
- If a node has no children in the general tree, its left pointer in the binary tree is `null`.
- If a node has no next sibling in the general tree, its right pointer in the binary tree is `null`.
- This conversion is reversible; the original general tree structure can be uniquely reconstructed from the resulting binary tree.

LCRS Conversion Visualization



General Tree



Converted Binary Tree (LCRS)

- **Left diagram:** General tree where node A has children B, C, D, and node C has children E, F.
- **Right diagram:** Converted Binary Tree. Left-links (L, blue

Tree Traversal: Inorder Traversal

Inorder Traversal (LVR)

Inorder traversal visits the left subtree, the root node, and then the right subtree.

- **Recursive Definition:**
  1. Traverse the left subtree in inorder.
  2. Visit the root node.
  3. Traverse the right subtree in inorder.
- For a Binary Search Tree (BST), inorder traversal visits nodes in ascending, sorted order of their keys.
- **Time Complexity:** $O(N)$ where N is the total number of nodes, as each node is visited exactly once.
- **Space Complexity:** $O(H)$ where H is the height of the tree, representing the maximum call stack depth ($O(\log N)$ for balanced trees, $O(N)$ for skewed trees).

Tree Traversal: Preorder Traversal

Preorder Traversal (VLR)

Preorder traversal visits the root node first, followed by the left subtree, and then the right subtree.

- **Recursive Definition:**
  1. Visit the root node.
  2. Traverse the left subtree in preorder.
  3. Traverse the right subtree in preorder.
- Preorder traversal is commonly used to create a copy of the tree, or to serialize a tree structure for storage.
- In an expression tree, preorder traversal produces the prefix notation (Polish notation) of the expression.
- Like Inorder, it runs in $O(N)$ time complexity and $O(H)$ auxiliary space complexity.

C++ Implementation of Traversals

- **Node Definition:**

```
struct Node {
    int data;
    Node* left;
    Node* right;
    Node(int val) : data(val), left(nullptr),
};
```

- **Inorder and Preorder Functions:**

```
void inorder(Node* root) {
    if (root == nullptr) return;
    inorder(root->left);
    std::cout << root->data << '-';
    inorder(root->right);
}

void preorder(Node* root) {
```

Data Structures (DI03000021)

GTU Diploma Engineering

BST Fundamentals & Postorder Traversal

Binary Search Tree (BST)

A Binary Search Tree (BST) is a node-based binary tree data structure where each node has at most two children. The BST property states that for any node N, the keys of all nodes in its left subtree are less than N's key, and the keys of all nodes in its right subtree are greater than N's key.

- Postorder traversal visits the nodes in the order: Left subtree, Right subtree, and then Root (L–R–N).
- It is a bottom-up traversal technique, meaning leaf nodes are processed before their parents.
- Crucial application: Used in deleting/destroying a tree because we must delete child nodes before deleting the parent node to prevent memory leaks.
- Time Complexity: $O(N)$ to visit all nodes; Auxiliary Space

Searching Algorithm in BST

Algorithm: BST_Search(root

1. If `root` is `Null` or `root.key == key`, return `root`.
2. If `key < root.key`, return `BST_Search(root.left, key)`.
3. Else, return `BST_Search(root.right, key)`.

- Searching takes advantage of the sorted property of the BST to prune the search space by half at each step.
- Iterative search is preferred in practice because it reduces recursion stack overhead to $O(1)$ space complexity.
- Average-case Time Complexity: $O(\log N)$ where N is the number of nodes in a balanced BST.
- Worst-case Time Complexity: $O(N)$ when the tree degenerates into a skewed binary tree (linked list structure).

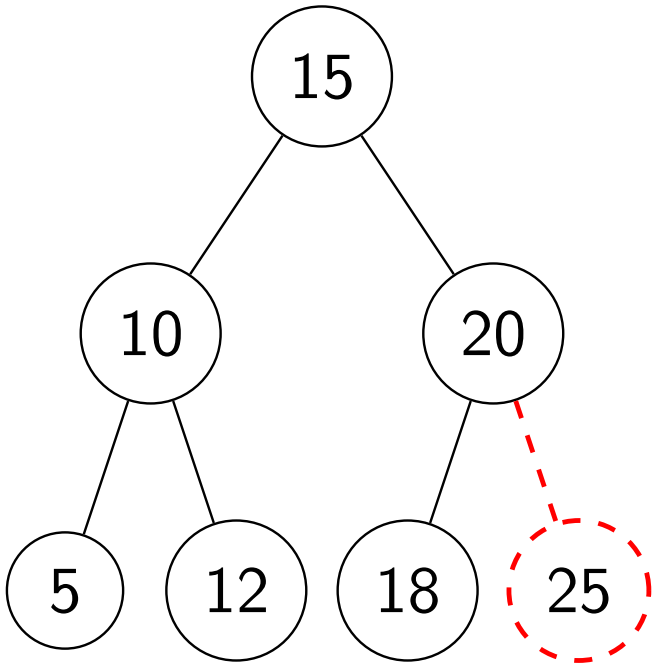
Insertion of a Node in BST

Algorithm: BST_Insert(root

1. If `root` is `Null`, return `new Node(key)`.
2. If `key < root.key`, `root.left = BST_Insert(root.left, key)`.
3. Else if `key > root.key`, `root.right = BST_Insert(root.right, key)`.
4. Return `root`.

- Insertion in a BST is always performed at a leaf position, maintaining the tree's BST invariant.
- The search path is followed until a `Null` pointer is reached, where the new node is then appended.
- Duplicate keys can either be ignored, stored in a frequency counter within the node, or inserted consistently into one of the subtrees.

Step-by-Step BST Insertion Diagram



- The diagram shows the insertion of key '25' into an existing BST.
- Search path: $25 > 15$ (go right to 20), $25 > 20$ (go right). Since the right child of 20 is `Null`, we insert it there.
- The red dashed node and edge highlight the newly inserted leaf element.
- All ancestor nodes preserve the BST property: Left nodes $<$ Parent $<$ Right nodes.

Deleting Nodes with 0 or 1 Child

BST Deletion: Cases 1 & 2

Deletion in a BST involves three scenarios depending on the number of children the target node possesses:

- **Case 1 (0 children):** Target node is a leaf. Set parent's pointer to `Null` and free node memory.
- **Case 2 (1 child):** Bypass target node by linking its parent directly to its single child.

- Case 1 example: Deleting node 5 in the previous tree. Its parent (10)'s left child pointer is set to `Null`.
- Case 2 example: If deleting a node with only a right child, the parent links directly to this right child.
- Memory management: In languages like C/C++, explicit deallocation (`free`/`delete`) must be called on the deleted node's memory.

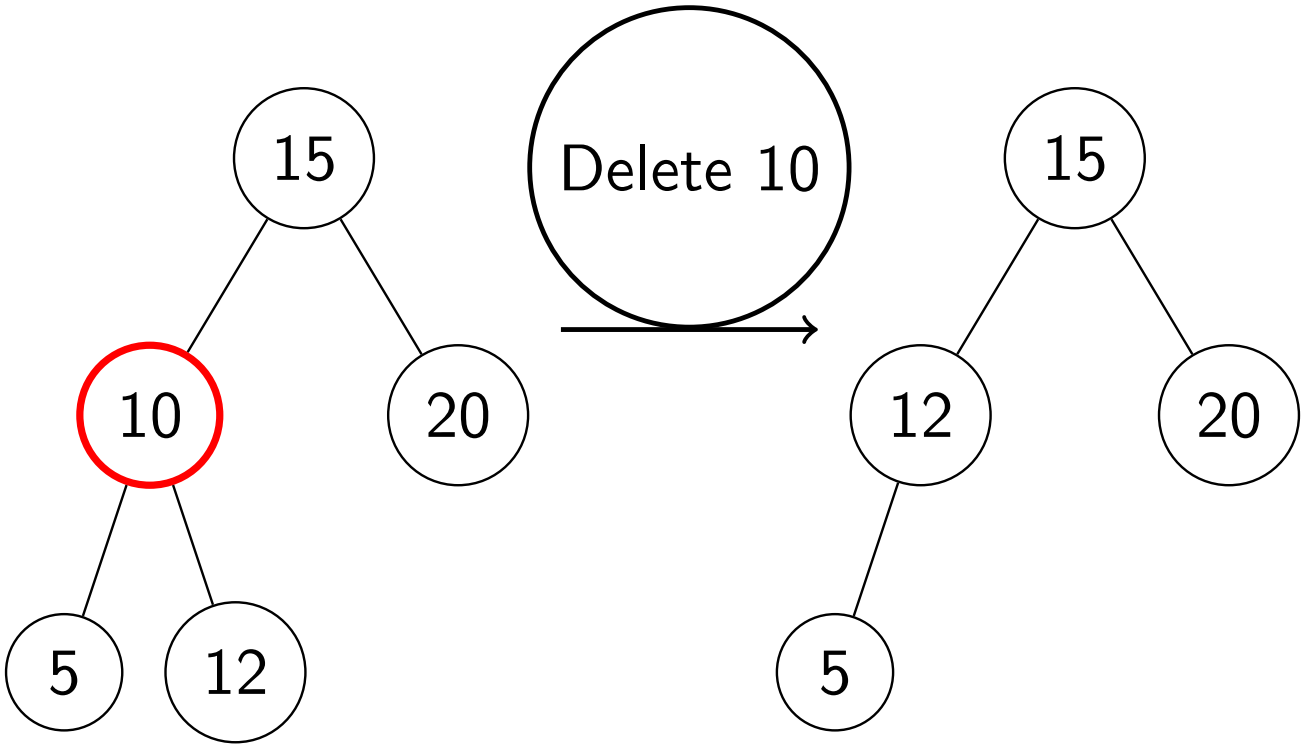
Deleting Nodes with Two Children

BST Deletion: Case 3 (Two Children)

1. Find the inorder successor (smallest node in right subtree) or inorder predecessor (largest node in left subtree).
2. Copy successor's/predecessor's key to the target node.
3. Recursively delete the successor/predecessor node (guaranteed to have at most 1 child).

- The inorder successor is found by going to the right child, then traversing left as far as possible.
- The inorder predecessor is found by going to the left child, then traversing right as far as possible.
- This reduction guarantees that step 3 will always resolve to a simpler Case 1 or Case 2 deletion.
- Replacing with either successor or predecessor ensures that the

Deletion of a Node with Two Children



- This diagram illustrates the deletion of node ‘10’ which has two children (5 and 12).
- We locate its inorder successor, which is the smallest node in the right subtree: node ‘12’.
- We replace node 10’s value with 12, then delete the original node 12 (a leaf node case).
- The BST property is fully preserved in the resulting tree shown

Graph Fundamentals: The Vertex

Graph & Vertex Definition

A Graph $G = (V, E)$ is a non-linear data structure consisting of a finite set of Vertices (or Nodes) V, and a set of Edges E that connect pairs of vertices.
Vertex: A fundamental unit of which graphs are formed. It represents an entity or coordinate.

- The degree of a vertex in an undirected graph is the number of edges incident to it. Handshaking Lemma: Sum of degrees $= 2 \cdot |E|$.
- In directed graphs, each vertex has an in-degree (incoming edges) and an out-degree (outgoing edges).
- Isolated Vertex: A vertex with degree 0. Pendent Vertex: A vertex with degree 1.
- Vertices are typically stored in memory as indices in an Adjacency Matrix ($O(1)$ lookup) or as head nodes in an

Summary of Lecture 32

Lecture Overview

Tree-based insertion/deletion mechanisms and introduction to Graph fundamentals.

- BST Operations: Search, Insert, and Delete depend heavily on Tree Height H.
- Postorder Traversal: Left $\rightarrow$ Right $\rightarrow$ Node processing pattern.
- Graph Vertex: Fundamental node unit in network structures.
- For balanced BSTs, operations complete in $O(\log N)$ time, whereas skewed BSTs degenerate to $O(N)$.
- Postorder traversal is essential for recursive processes that work from bottom up, such as tree deallocation.
- Graph vertex forms the basic data point representation, which can be connected via directed or undirected edges.
- Next Lecture: Graph representations (Adjacency Matrix vs. Adjacency List) and Graph Traversals (DFS and BFS).

Data Structures (DI03000021)

GTU Diploma Engineering

Foundations: The Anatomy of a Graph

A

graph $G = (V, E)$ consists of a set of vertices V and a set of edges E. Edges represent topological relationships between vertices, which can be directed or undirected.

- An edge $e = (u, v)$ is defined by its endpoints. In a directed graph (digraph), u is the source (tail) and v is the target (head).
- **Undirected Graph**: Edges are unordered pairs $\{u, v\}$, implying bidirectional traversal.
- **Directed Graph**: Edges are ordered pairs (u, v), restricting traversal from u to v.
- **Weighted Edges**: A function $w : E \to \mathbb{R}$ maps edges to numerical values representing costs, distances, capacities, or latencies.

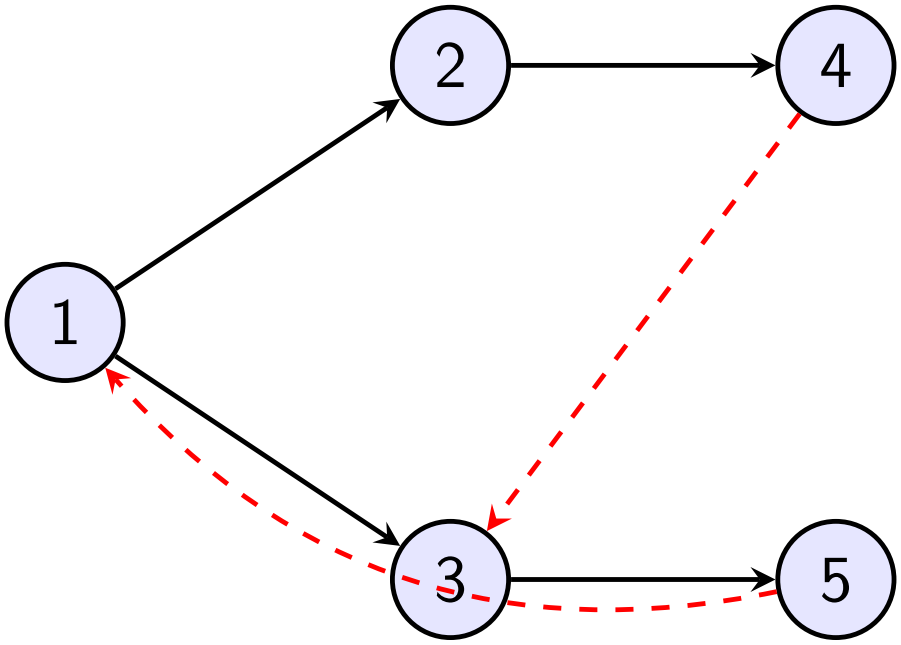
Depth-First Search (DFS) Edge Classification

R

unning DFS on a graph constructs a DFS forest. We classify edges into four categories based on the discovery time ($d[u]$) and finishing time ($f[u]$) of vertices.

- **Tree Edges**: Edges in the depth-first forest. Edge (u, v) is a tree edge if v was first discovered by exploring edge (u, v).
- **Back Edges**: Edges (u, v) connecting a vertex u to an ancestor v in a DFS tree. This includes self-loops.
- **Forward Edges**: Non-tree directed edges (u, v) connecting a vertex u to a descendant v in a DFS tree.
- **Cross Edges**: All other edges. They connect vertices in different trees or vertices with no ancestor/descendant relationship in the same tree.

Visualizing DFS Edge Classification



- **Tree Edges** (black solid arrows): $(1, 2)$, $(1, 3)$, $(2, 4)$, $(3, 5)$ are traversed to discover new vertices.
- **Back Edge** (red dashed bend arrow): $(5, 1)$ points from descendant 5 to ancestor 1, indicating a cycle.
- **Cross Edge** (red dashed straight arrow): $(4, 3)$ connects vertex 4 to 3 in different subtrees (3 is already finished when exploring from 4).

Paths: Definitions and Properties

A

path P of length k from vertex u to u' in $G = (V, E)$ is a sequence $\langle v_0, v_1, \ldots, v_k \rangle$ of vertices such that $u = v_0$, $u' = v_k$, and $(v_{i-1}, v_i) \in E$ for $i = 1, 2, \ldots, k$.

- **Simple Path**: A path where all vertices $v_0, v_1, \ldots, v_k$ are distinct.
- **Path Length**: Number of edges in the path (which is k). In weighted graphs, it is the sum of edge weights.
- **Reachability**: A vertex v is reachable from u (denoted $u \rightarrow^* v$) if there exists a path from u to v.
- **Subpath**: A subsequence of vertices in a path is itself a path. Subpaths of shortest paths are also shortest paths (optimal substructure property).

Shortest Path Routing Algorithms

G

iven a source vertex s, we compute shortest paths using distinct algorithms suited for specific edge types and constraints.

- **Breadth-First Search (BFS)**: Computes shortest paths in unweighted graphs in $O(V + E)$ time.
- **Dijkstra's Algorithm**: Single-source shortest paths on graphs with non-negative edge weights. Time complexity: $O(V \log V + E)$ with Fibonacci heaps.
- **Bellman-Ford Algorithm**: Resolves single-source shortest paths on graphs with negative edge weights in $O(V \cdot E)$ and identifies negative cycles.
- **Floyd-Warshall Algorithm**: All-pairs shortest paths using dynamic programming in $O(V^3)$ time.

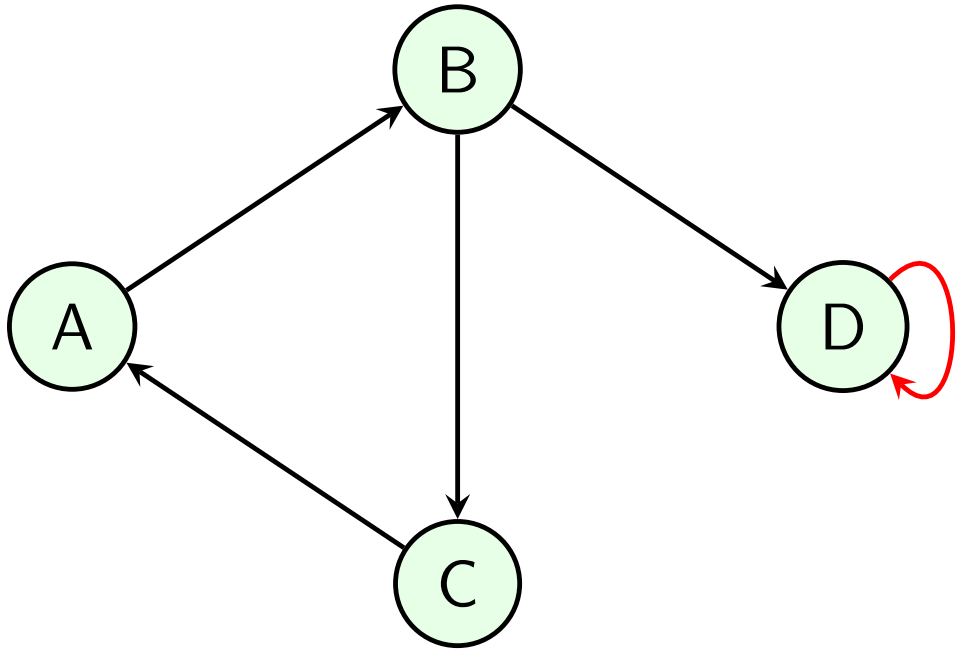
Cycles: Definitions and Classification

A

cycle is a path $\langle v_0, v_1, \ldots, v_k \rangle$ where $v_0 = v_k$ and length $k \geq 1$. If $v_1, v_2, \ldots, v_k$ are distinct, the cycle is simple.

- **Self-Loop**: A cycle of length 1, formed by an edge (v, v) from a vertex to itself.
- **Directed Acyclic Graph (DAG)**: A directed graph containing no directed cycles. Essential for representing dependency hierarchies.
- **Hamiltonian Cycle**: A simple cycle that visits every vertex in G exactly once (NP-complete to find).
- **Eulerian Cycle**: A cycle that traverses every edge in G exactly once (exists if and only if every vertex has an even degree in undirected graphs).

Visualizing Cycle and Path Structures



- **Directed Cycle**: The loop $A \to B \to C \to A$ forms a simple directed cycle of length 3.
- **Self-Loop**: Vertex D has a red self-loop of length 1.
- **Path**: $A \to B \to D$ is a simple path of length 2 from A to D.
- **Reachability**: D is reachable from A, B, and C, but none of A, B, or C are reachable from D.

Algorithms for Cycle Detection

C

ycle detection is a fundamental operation before topological sorting or deadlock resolution.

- **Undirected Cycle Detection**: Use Union-Find (Disjoint Set Union). For each edge (u, v), if $\text{Find}(u) = \text{Find}(v)$, a cycle is detected; otherwise $\text{Union}(u, v)$. Complexity: $O(E \cdot \alpha(V))$.
- **Directed Cycle Detection (DFS)**: Color vertices: White (unvisited), Gray (actively visiting), Black (finished). If we encounter a Gray vertex during traversal, a back edge (cycle) exists. Complexity: $O(V + E)$.
- **Topological Sort (Kahn's Algorithm)**: Computes in-degrees. If the count of processed vertices $< |V|$, a directed cycle exists. Complexity: $O(V + E)$.
- **Applications**: Wait-For Graph (WFG) analysis in Operating Systems for deadlock detection.

Complexity and Data Structure Summary

S

electing the optimal graph representation depends on the density of edges and the frequency of edge lookup operations.

- **Edge Query Time** (Is $(u, v) \in E$?): $O(1)$ in Adjacency Matrix; $O(\deg(u))$ in Adjacency List.
- **Space Complexity**: $O(V^2)$ for Adjacency Matrix (optimal for dense graphs); $O(V + E)$ for Adjacency List (optimal for sparse graphs).
- **DFS/BFS Traversal**: $O(V^2)$ for Matrix representation; $O(V + E)$ for List representation.
- **All-Pairs Shortest Path**: Floyd-Warshall $O(V^3)$ time complexity and $O(V^2)$ space complexity.

Data Structures (DI03000021)

GTU Diploma Engineering

Mathematical Formulation of Graphs

A

graph G is defined as an ordered pair $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges (or links) representing connections between vertices.

- $V = \{v_1, v_2, \ldots, v_n\}$ defines the entity set, where cardinality $|V|$ is the order of the graph.
- E represents relationships, where cardinality $|E|$ is the size of the graph.
- In undirected graphs, E consists of unordered pairs of vertices: $e = \{u, v\}$.
- In directed graphs, E consists of ordered pairs of vertices: $e = (u, v)$, representing a one-way path.

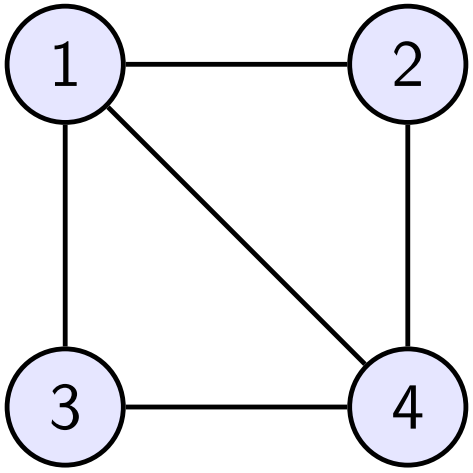
Undirected Graphs: Definition & Properties

I

n an undirected graph, edges have no orientation. Relations are symmetric: if there is an edge between u and v, then u is adjacent to v and v is adjacent to u.

- Edges are symmetric unordered sets: $\{u, v\}$ is equivalent to $\{v, u\}$.
- The degree of a vertex v, $\deg(v)$, is the number of edges incident to it.
- Handshaking Lemma: $\sum_{v \in V} \deg(v) = 2|E|$. The sum of degrees is always even.
- Maximum number of edges in a simple undirected graph with n vertices is $n(n-1)/2$, which is $O(n^2)$.

Undirected Graph Topology



- Vertex set $V = \{1, 2, 3, 4\}$ with size $|V| = 4$.
- Edge set $E = \{\{1, 2\}, \{1, 3\}, \{1, 4\}, \{2, 4\}, \{3, 4\}\}$ with size $|E| = 5$.
- Degrees: $\deg(1) = 3$, $\deg(2) = 2$, $\deg(3) = 2$, $\deg(4) = 3$.
- Sum of degrees $= 3 + 2 + 2 + 3 = 10$, verifying $2 \cdot |E| = 10$.

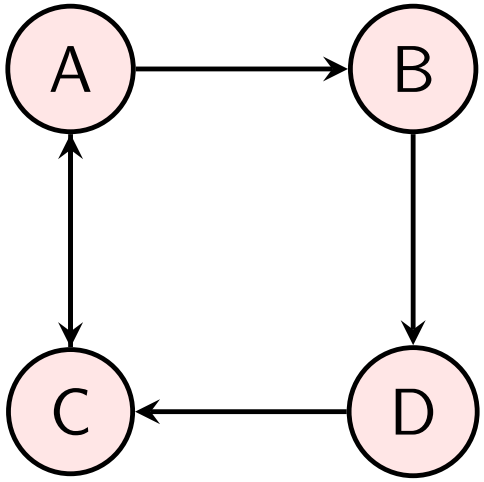
Directed Graphs (Digraphs): Formal Theory

I

n a directed graph, every edge has an orientation, going from a source/origin vertex to a target/destination vertex.

- Edges are ordered pairs: (u, v) is a directed edge from u (tail) to v (head). Note that $(u, v) \neq (v, u)$.
- In-degree of v, $\text{indeg}(v)$, is the number of edges entering v. Out-degree of v, $\text{outdeg}(v)$, is the number of edges leaving v.
- Directed Handshaking: The sum of in-degrees equals the sum of out-degrees, which equals $|E|$.
- Maximum number of edges in a simple digraph with n vertices is $n(n-1)$, which is $O(n^2)$.

Directed Graph Topology



- Vertex set $V = \{A, B, C, D\}$ with size $|V| = 4$.
- Directed edge set $E = \{(A, B), (A, C), (B, D), (D, C), (C, A)\}$ with size $|E| = 5$.
- Degrees: $\text{indeg}(A) = 1$, $\text{outdeg}(A) = 2$; $\text{indeg}(C) = 2$, $\text{outdeg}(C) = 1$.
- Sum of $\text{indeg}(v) = 1\ (\text{A}) + 1\ (\text{B}) + 2\ (\text{C}) + 1\ (\text{D}) = 5 = |E|$.

Graph Connectivity Properties

C

onnectivity describes whether nodes are reachable from each other through paths in the graph topology.

- An undirected graph is Connected if there exists an undirected path between every pair of vertices.
- A directed graph is Strongly Connected if a directed path exists from any node u to any node v, and vice versa.
- A directed graph is Weakly Connected if its underlying undirected representation (ignoring direction) is connected.
- Tarjan's and Kosaraju's algorithms find Strongly Connected Components (SCCs) in $O(V + E)$ time.

Complete Graphs: Analysis and Density

A

complete graph is a simple graph where an edge exists between every single pair of distinct vertices, representing maximum edge density.

- An undirected complete graph on n vertices is denoted by K_n.
- Every vertex in K_n has degree exactly $n-1$.
- Total number of edges in K_n is exactly $n(n-1)/2$, meaning K_n is $O(n^2)$ dense.
- Complete graphs serve as the worst-case scenario for space complexity in adjacency lists: $O(V^2)$.

Weighted Graphs: Costs and Traversal

A

weighted graph associates a numerical value (weight, cost, distance, or capacity) with each edge in the edge set.

- Mathematically defined as a triple $G = (V, E, W)$ where $W : E \to \mathbb{R}$ is the weight function.
- Edge weights can represent physical distances, latency, or capacity constraints in flow networks.
- Dijkstra's single-source shortest path algorithm operates on positive weighted graphs in $O((V + E) \log V)$ time.
- Bellman-Ford algorithm handles negative weights and detects negative cycles in $O(V \cdot E)$ time.

G

raph representations dictate space complexity and runtime efficiency of core graph traversal algorithms.

- Adjacency Matrix: Space complexity is $O(V^2)$. Edge existence checks are fast $O(1)$, making it ideal for dense graphs.
- Adjacency List: Space complexity is $O(V + E)$. Ideal for sparse graphs; traversing neighbors takes $O(\deg(u))$ time.
- Breadth-First Search (BFS) and Depth-First Search (DFS) run in $O(V + E)$ time complexity.
- Directed Acyclic Graphs (DAGs) can be Topologically Sorted in $O(V + E)$ time using Kahn's algorithm or DFS.

Data Structures (DI03000021)

GTU Diploma Engineering

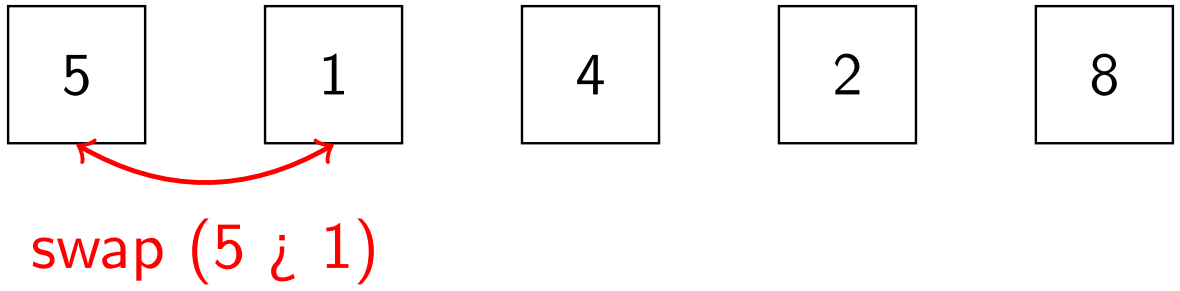
What is Bubble Sort?

B

ubble Sort is a straightforward, comparison-based sorting algorithm where adjacent elements are compared and swapped if they are in the wrong order.

- It is named 'Bubble Sort' because smaller elements 'bubble' to the top (beginning) of the list, while larger elements sink to the bottom (end).
- It is an in-place and stable sorting algorithm.
- Highly intuitive but inefficient for large datasets.

Bubble Sort: One Step Visualization



- We compare the first two elements: 5 and 1.
- Since 5 is greater than 1, we swap them.
- The array becomes [1, 5, 4, 2, 8].
- The next comparison will compare 5 and 4.

Bubble Sort Pseudocode

```
procedure bubbleSort(A : list of sortable items)
    n := length(A)
    repeat
        swapped := false
        for i := 1 to n-1 inclusive do
            if A[i-1] > A[i] then
                swap(A[i-1], A[i])
                swapped := true
            end if
        end for
        n := n - 1
    until not swapped
end procedure
```

- The 'swapped' boolean flag tracks whether any elements were swapped in the pass.
- If a pass completes with no swaps, the list is sorted, allowing

Full Execution Trace Example

Input: [5, 1, 4, 2, 8]
Pass 1: Compare (5,1)→swap; (5,4)→swap; (5,2)→swap; (5,8)→no swap. State: [1, 4, 2, 5, 8]
Pass 2: Compare (1,4)→no swap; (4,2)→swap; (4,5)→no swap. State: [1, 2, 4, 5, 8]
Pass 3: Compare (1,2)→no swap; (2,4)→no swap. Swapped is false. State: [1, 2, 4, 5, 8]

- During Pass 1, 8 is verified as sorted at the last index.
- During Pass 2, 5 is verified as sorted at the second-to-last index.
- During Pass 3, no swaps occur, so the algorithm terminates early.
- Total comparisons: $4 + 3 + 2 = 9$ without optimization, but optimized it stops early.

Time Complexity Analysis

Time Complexity of Bubble Sort:

- Worst Case: $\mathcal{O}(n^2)$
- Average Case: $\mathcal{O}(n^2)$
- Best Case: $\mathcal{O}(n)$

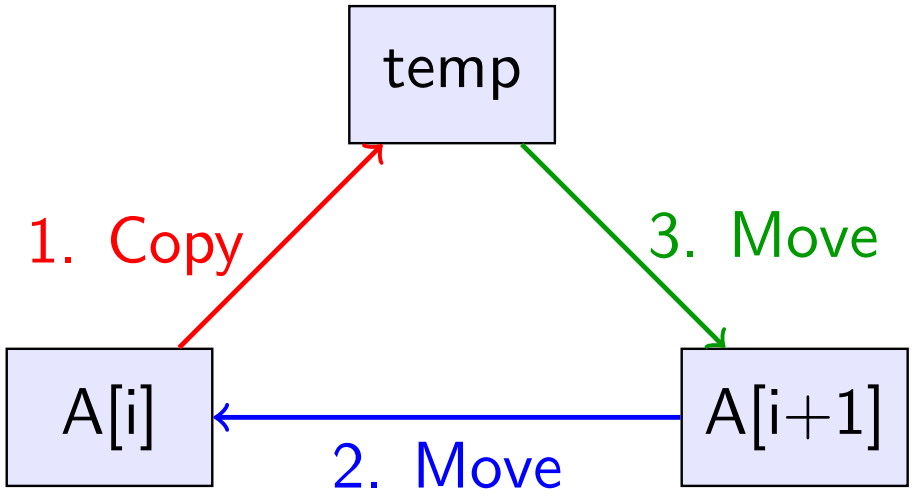
- Worst case occurs when the array is sorted in reverse order: requires $n(n-1)/2$ comparisons and swaps.
- Average case requires quadratic comparisons and swaps due to random distribution of elements.
- Best case occurs when the array is already sorted: requires only 1 pass with $(n-1)$ comparisons and 0 swaps.
- Bubble sort is inefficient for large datasets due to its quadratic growth.

Space Complexity and Auxiliary Memory

Space Complexity: $\mathcal{O}(1)$ Auxiliary Memory

- Bubble Sort is an 'in-place' sorting algorithm.
- It only requires a constant amount of extra memory space for temporary swap variables.
- No dynamic memory allocation is performed during execution.
- This makes it highly space-efficient, though time-inefficient.

The Swapping Mechanism



- To swap two elements, we must temporarily store one of them to prevent it from being overwritten.
- Step 1 copies `A[i]` to a temporary variable 'temp'.
- Step 2 overwrites `A[i]` with the value of `A[i+1]`.
- Step 3 copies the value in 'temp' back into `A[i+1]`.

Optimized C++ Implementation

```
void bubbleSort(int arr[], int n) {
    bool swapped;
    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
                swapped = true;
            }
        }
        if (!swapped) {
            break;
        }
    }
}
```

Algorithmic Properties & Summary

Bubble Sort Characteristics:

- **Stability:** Stable
- **In-place:** Yes
- **Adaptive:** Yes (with optimization)

- Stable sorting algorithm: elements with equal keys maintain their relative order.
- Adaptive: takes advantage of pre-sorted components, reducing time on nearly-sorted data.
- Suffers from the 'turtles and rabbits' problem where small values at the end move very slowly to the front.
- Rarely used in practice, but serves as an excellent educational tool for sorting fundamentals.

Data Structures (DI03000021)

GTU Diploma Engineering

Core Intuition of Selection Sort

Selection Sort

Selection Sort conceptually divides the input array into two segments: a sorted subarray on the left and an unsorted subarray on the right.

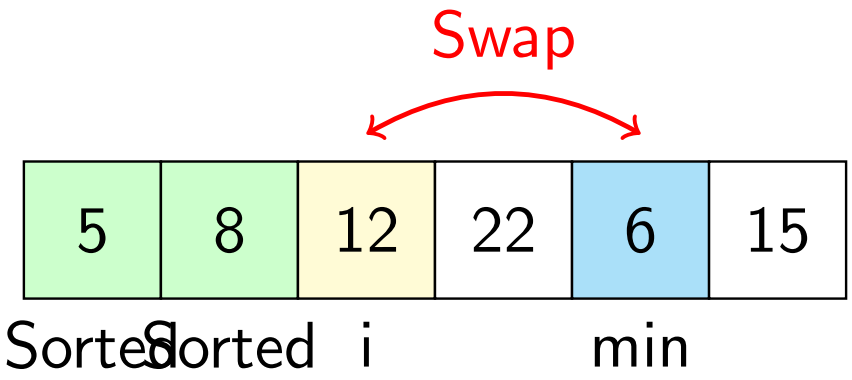
- Initially, the sorted subarray is empty, and the unsorted subarray contains all the elements.
- The algorithm works in passes. In each pass, it scans the unsorted subarray to find the minimum (or maximum) element.
- It then swaps this minimum element with the first element of the unsorted subarray.
- This increases the sorted subarray's length by one and decreases the unsorted subarray's length by one, progressing from left to right.

The Selection Sort Algorithm (Pseudocode)

```
Algorithm SelectionSort(A, n):
  for i = 0 to n - 2 do:
    min_idx = i
    for j = i + 1 to n - 1 do:
      if A[j] < A[min_idx] then:
        min_idx = j
    if min_idx != i then:
      swap(A[i], A[min_idx])
```

- The outer loop variable 'i' represents the boundary between the sorted and unsorted portions.
- The inner loop variable 'j' scans the unsorted portion from index 'i + 1' to 'n - 1'.
- We maintain 'min_idx' to track the position of the smallest element encountered so far.
- At the end of the inner loop, a swap is performed only if the index of the minimum element ('min_idx') is different from 'i'.

Visualizing an Iteration (Swap Step)



- The green cells (indices 0 and 1) show elements that are already sorted and in their final positions.
- The current boundary element '`i`' is at index 2 (value 12).
- The inner loop searches the unsorted portion (indices 2 to 5) and finds the minimum element '6' at index 4.
- A swap is initiated between index 2 and index 4, placing '6' into the sorted part and moving '12' further right.

Step-by-Step Execution Trace

Trace Overview

Tracing array: $[29, 10, 14, 37, 13]$ (Size $n = 5$)

- Pass 1: Min is 10. Swap 10 & 29 $\rightarrow [10, 29, 14, 37, 13]$
- Pass 2: Min is 13. Swap 13 & 29 $\rightarrow [10, 13, 14, 37, 29]$
- Pass 3: Min is 14. No swap needed $\rightarrow$ $[10, 13, 14, 37, 29]$
- Pass 4: Min is 29. Swap 29 & 37 $\rightarrow [10, 13, 14, 29, 37]$

- Initial state: unsorted array $[29, 10, 14, 37, 13]$.
- In Pass 1, the minimum element in the entire array is 10. It swaps with the first element (29).
- In Pass 2, index $i = 1$ (value 29). The minimum of $[29, 14, 37, 13]$ is 13 at index 4. Swap occurs.
- In Pass 3, index $i = 2$ (value 14). The minimum of $[14, 37, 29]$ is 14 itself. No swap happens.

C++ Implementation

```
void selectionSort(int arr[], int n) {
    for (int i = 0; i < n - 1; ++i) {
        int minIdx = i;
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[minIdx]) {
                minIdx = j;
            }
        }
        if (minIdx != i) {
            int temp = arr[i];
            arr[i] = arr[minIdx];
            arr[minIdx] = temp;
        }
    }
}
```

- The outer loop variable 'i' runs from 0 up to 'n - 2', making

Complexity Analysis

Complexity Summary

Time Complexity:

- Best Case: $O(n^2)$
- Worst Case: $O(n^2)$
- Average Case: $O(n^2)$

Space Complexity:

- Auxiliary Space: $O(1)$
- Total Space: $O(n)$

- Comparisons: The total number of comparisons is $(n-1)+(n-2)+\cdots+1 = n(n-1)/2$, which is $O(n^2)$. This is constant regardless of how the input array is ordered.
- Swaps: The worst-case and average-case number of swaps is $O(n)$ (specifically, at most $n-1$ swaps).

Algorithmic Properties: Stability & Adaptability

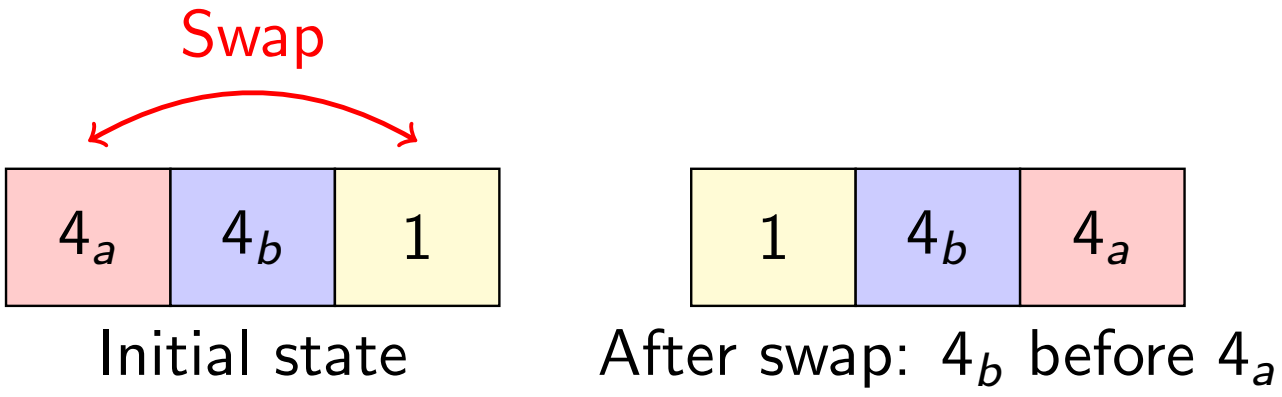
Properties

Selection Sort is:

- **Unstable:** Can change the relative order of equal elements.
- **Non-adaptive:** The number of comparisons remains $O(n^2)$ even if the array is already sorted.

- A sort is stable if equal elements retain their original relative order. Selection Sort is unstable because it performs long-distance swaps.
- Selection Sort is non-adaptive because the inner loop must always scan to the end of the unsorted subarray to guarantee the minimum is found.
- However, its minimal number of memory writes ($O(n)$ swaps) is a key advantage when writing to memory is highly expensive

Unstable Behavior Visualized



- Consider the input array $[4_a, 4_b, 1]$, where 4_a and 4_b are equal values but 4_a appears before 4_b.
- In the first iteration, the minimum element found is 1 at index 2.
- We swap 1 with the first element of the unsorted portion (4_a at index 0).
- The array becomes $[1, 4_b, 4_a]$. Since 4_b now appears before 4_a, the original relative order is lost, proving Selection Sort is unstable.

Comparison with Other $O(n^2)$ Algorithms

Algorithm Comparison Table

- **Bubble Sort:** $O(n^2)$ comp, $O(n^2)$ swaps (Stable, Adaptive)
- **Insertion Sort:** $O(n^2)$ comp, $O(n^2)$ shifts (Stable, Adaptive)
- **Selection Sort:** $O(n^2)$ comp, $O(n)$ swaps (Unstable, Non-adaptive)

- Selection Sort is preferred over Bubble Sort because it performs a maximum of $O(n)$ swaps compared to $O(n^2)$ swaps in Bubble Sort.
- Insertion Sort is generally superior to Selection Sort for nearly sorted arrays, as Insertion Sort runs in $O(n)$ time in the best case.
- Use Selection Sort when auxiliary memory is limited ($O(1)$)

Data Structures (DI03000021)

GTU Diploma Engineering

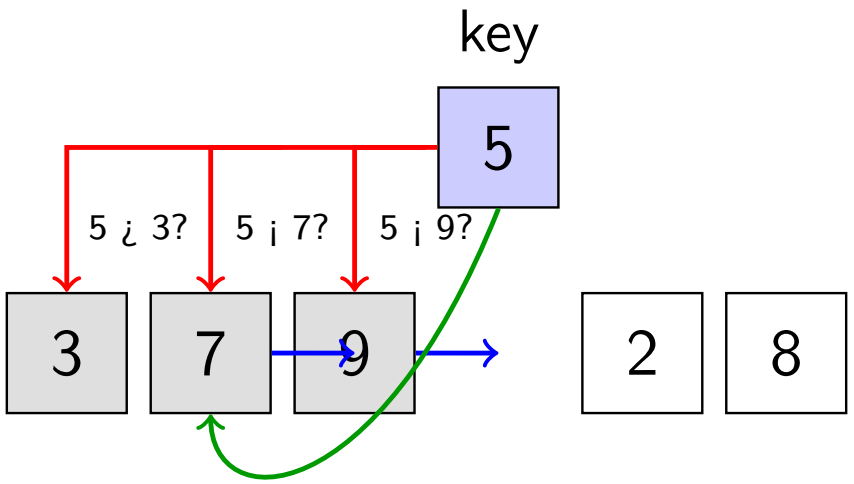
Conceptual Framework & Card Analogy

Conceptual Framework

How we sort cards in our hands: we take a card from the unsorted deck and place it into its correct position relative to the cards we already hold.

- Divide the array conceptually into two segments: a sorted prefix and an unsorted suffix.
- Initially, the sorted prefix contains only the first element (index 0), which is trivially sorted.
- In each step, pick the first element of the unsorted suffix (the 'key').
- Compare the key with elements in the sorted prefix from right to left.
- Shift larger elements one position to the right to make space for the key.
- Insert the key at its correct relative sorted position, expanding

Visualizing a Single Insertion Step



- The sorted partition currently holds $[3, 7, 9]$. The unsorted partition begins with key $= 5$.
- Red arrows represent comparisons: key (5) is compared with 9, then 7, and finally 3.
- Blue arrows represent shifts: 9 and 7 are shifted one position to the right as they are larger than 5.
- Comparison with 3 returns false ($5 > 3$), halting the shift loop.
- The green path indicates the insertion of key (5) into the vacant position (index 1).

Insertion Sort Pseudocode

Algorithm Pseudocode

```
INSERTION-SORT(A, n):
for j = 1 to n - 1
    key = A[j]
    // Insert A[j] into the sorted sequence A[0..j-1]
    i = j - 1
    while i >= 0 and A[i] > key
        A[i + 1] = A[i]
        i = i - 1
    A[i + 1] = key
```

- The outer loop variable j starts at index 1 and goes up to $n - 1$, tracking the first element of the unsorted segment.
- The `key` variable temporarily holds the element to be inserted, preventing it from being overwritten during shifts.
- The inner loop variable i starts at $j - 1$ and moves backwards

Python & C++ Implementations

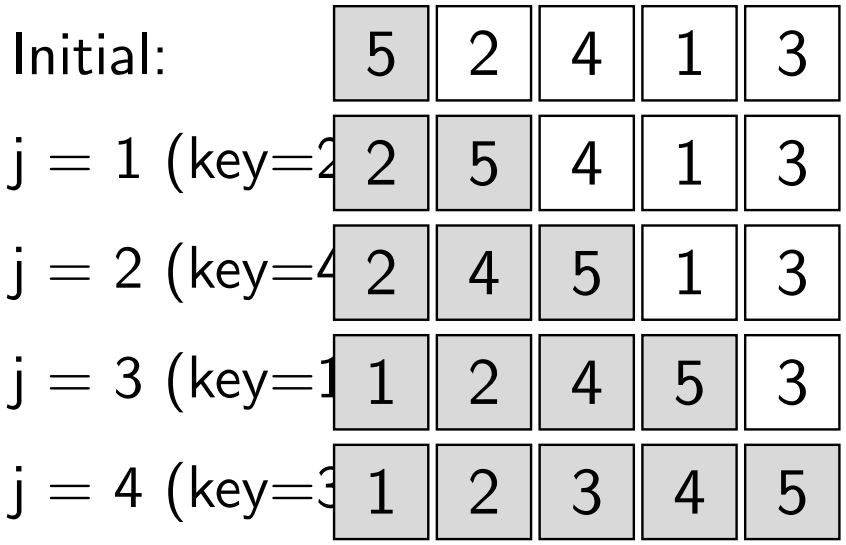
Python:

```
def insertion_sort(arr):
    for j in range(1, len(arr)):
        key = arr[j]
        i = j - 1
        while i >= 0 and arr[i] > key:
            arr[i + 1] = arr[i]
            i -= 1
        arr[i + 1] = key
```

C++:

```
void insertionSort(int arr[], int n) {
    for (int j = 1; j < n; ++j) {
        int key = arr[j];
        int i = j - 1;
        while (i >= 0 && arr[i] > key) {
            arr[i + 1] = arr[i];
```

Execution Walkthrough of Insertion Sort



- Shaded elements indicate the sorted portion of the array after each outer loop iteration.
- At $j = 1$, the key 2 is compared to 5 and inserted before it.
- At $j = 2$, the key 4 is compared to 5 (shifted right) and then to 2 (no shift, inserted at index 1).
- At $j = 3$, the key 1 is smaller than all sorted elements, shifting 5, 4, and 2, then inserted at index 0.
- At $j = 4$, the key 3 is compared to 5 and 4 (shifted), then inserted at index 2, resulting in a fully sorted array.

Time Complexity Analysis

Time Complexity Summary

- **Best Case:** $O(n)$ comparisons, $O(1)$ shifts (Array already sorted)
- **Worst Case:** $O(n^2)$ comparisons and shifts (Array sorted in reverse order)
- **Average Case:** $O(n^2)$ comparisons and shifts

- In the Best Case, the array is already sorted. The inner loop condition '`A[i] > key`' is checked once per iteration and immediately fails, resulting in exactly $(n-1)$ comparisons.
- In the Worst Case, the array is in reverse order. The inner loop must shift every element in the sorted prefix, resulting in $\sum_{i=1}^{n-1} i = \frac{n(n-1)}{2} = O(n^2)$ comparisons and shifts.
- In the Average Case, each element is compared with half of the elements in the sorted prefix, leading to roughly $\frac{n(n-1)}{4}$

Space Complexity & Stability

Space & Stability Profile

- **Auxiliary Space:** $O(1)$
- **Stability:** Stable (Preserves relative order of duplicate elements)
- **In-place:** Yes (Modifies the input array directly without extra copy)

- Insertion Sort is an in-place sorting algorithm. It only requires a constant amount of extra memory space for variables '`key`', '`i`', and '`j`'.
- Stability is maintained because the inner loop condition is '`A[i] > key`' rather than '`A[i] >= key`'.
- If an element equals the key, the inner loop terminates, and the key is placed immediately after the identical element, preserving relative order.

Insertion Sort vs. Other Elementary Sorts

Comparison Table

- **Insertion Sort:** Best $O(n)$, Avg $O(n^2)$, Worst $O(n^2)$, Stable: Yes, In-place: Yes
- **Selection Sort:** Best $O(n^2)$, Avg $O(n^2)$, Worst $O(n^2)$, Stable: No, In-place: Yes
- **Bubble Sort:** Best $O(n)$, Avg $O(n^2)$, Worst $O(n^2)$, Stable: Yes, In-place: Yes

- Unlike Selection Sort, Insertion Sort is adaptive; its run-time depends on the initial level of sorting in the array.
- Selection Sort always performs $O(n^2)$ comparisons because it must scan the remaining unsorted suffix to find the minimum.
- While Bubble Sort with an optimized flag can also achieve $O(n)$ in the best case, Insertion Sort generally performs fewer swaps/shifts.

Key Takeaways and Advanced Features

Summary

- Insertion Sort is highly effective for nearly sorted or small datasets.
- It is adaptive, stable, in-place, and online.
- Often serves as the base-case optimizer for divide-and-conquer algorithms.

- An important feature of Insertion Sort is that it is an online algorithm: it can sort a list as it receives it, element by element.
- This makes it highly suitable for streaming data inputs where the complete dataset is not available at the start.
- It is also extremely cache-friendly due to sequential memory access patterns during comparisons and shifts.
- In practice, library implementations of QuickSort or MergeSort

Data Structures (DI03000021)

GTU Diploma Engineering

Core Concepts of Quick Sort

Quick Sort

Quick Sort is an efficient, in-place, comparison-based sorting algorithm developed by Tony Hoare in 1959. It employs a divide-and-conquer strategy to sort arrays or lists.

- **Divide:** Choose a pivot element from the array.
- **Partition:** Reorder the array so that all elements less than the pivot come before it, and all elements greater than the pivot come after it. After partitioning, the pivot is in its final position.
- **Conquer:** Recursively apply the above steps to the sub-arrays of smaller elements and larger elements.
- **Base Case:** Sub-arrays of size 0 or 1 are already sorted, which terminates the recursion.

Partitioning Schemes

Partitioning Overview

The partition algorithm is the core of Quick Sort. The two most common partition schemes are Lomuto Partition Scheme and Hoare Partition Scheme.

- **Lomuto Partition:** Easier to implement. Typically chooses the last element as the pivot. Maintains an index i to track the boundary of elements $\leq$ pivot. Scans the array with index j from low to high-1, swapping when $A[j] \leq$ pivot.
- **Hoare Partition:** More efficient in practice than Lomuto. Uses two indices starting at the ends of the array, moving toward each other until they find a pair of elements out of order with respect to the pivot. Performs fewer swaps on average.
- **Stability:** Both standard partitioning schemes are unstable, meaning they do not preserve the relative order of equal elements without additional space.

Lomuto Partition Implementation

```
int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);
    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(&arr[i], &arr[j]);
        }
    }
    swap(&arr[i + 1], &arr[high]);
    return (i + 1);
}
```

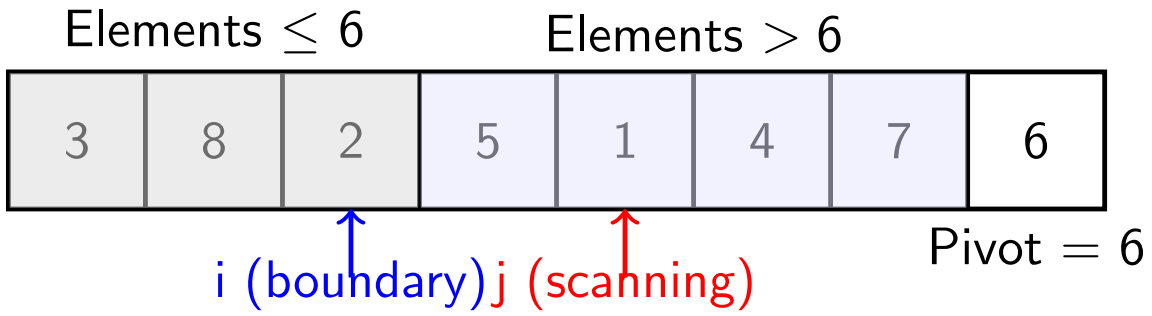
- The pivot is selected as the last element of the subarray, `arr[high]`.
- The index `i` points to the last element smaller than or equal to the pivot.

Quick Sort Algorithm Implementation

```
void quickSort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quickSort(arr, low, pi - 1);
        quickSort(arr, pi + 1, high);
    }
}
```

- Checks if the partition bounds are valid (`low < high`) to prevent infinite recursion on empty or single-element subarrays.
- Finds the partitioning index `pi` such that `arr[pi]` is in its correct sorted position.
- Recursively sorts the elements before the partition point (`low` to `pi - 1`).
- Recursively sorts the elements after the partition point (`pi + 1` to `high`).

Visualizing Lomuto Partitioning



- Initial array: [3, 8, 2, 5, 1, 4, 7, 6] with pivot = 6.
- Index i tracks the boundary of elements smaller than the pivot.
- Index j scans through the array from left to right.
- When $arr[j] <$ pivot, i is incremented, and $arr[i]$ is swapped with $arr[j]$.

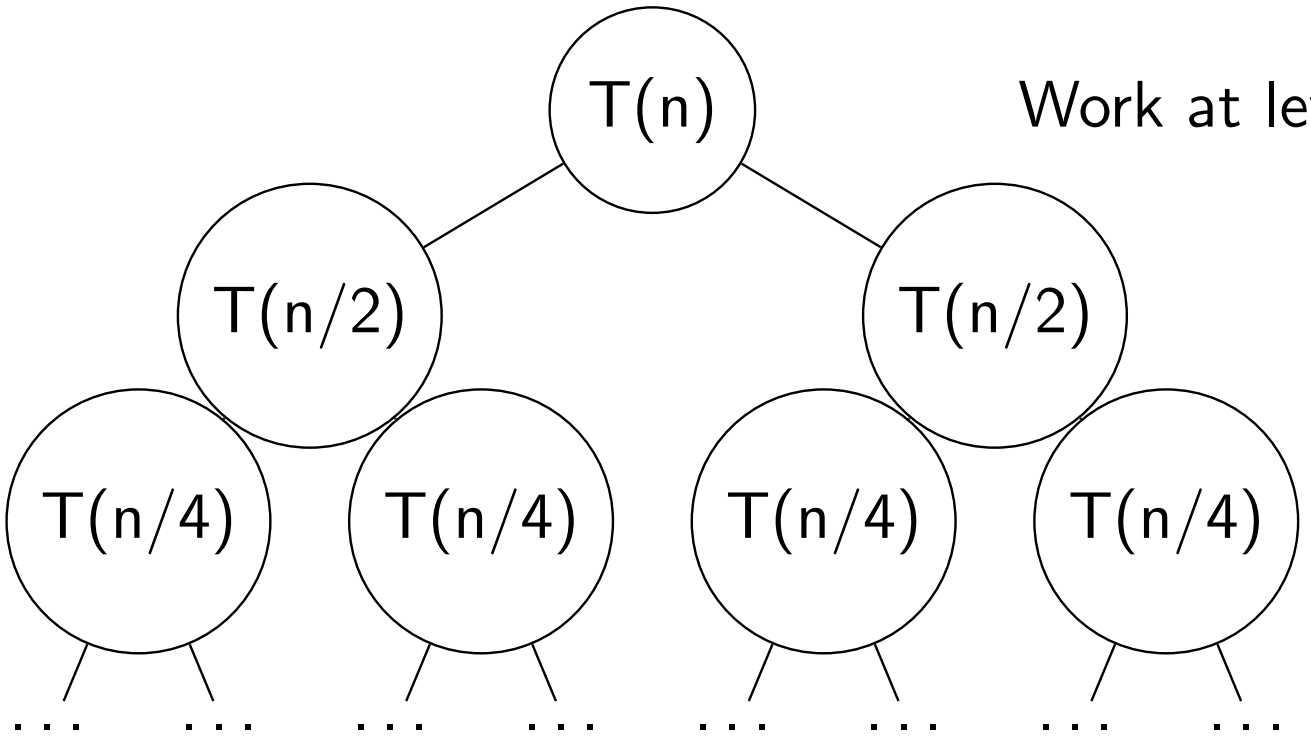
Selecting the Right Pivot

Pivot Selection Impact

The efficiency of Quick Sort heavily depends on the choice of the pivot element. A poor pivot choice can lead to worst-case $O(n^2)$ time complexity.

- **Always pick the first or last element:** Simple to implement, but leads to worst-case performance $O(n^2)$ if the array is already sorted or reverse-sorted.
- **Random pivot selection:** Choosing a random element as pivot guarantees $O(n \log n)$ expected time complexity on any input, preventing adversaries from triggering the worst case.
- **Median-of-three:** Choose the median of the first, middle, and last elements of the partition. This provides a very good approximation of the true median and improves performance by 20–30% in practice.

Quick Sort Recursion Tree



Work at level: $O(n)$

Work at level: $O(n)$

Total Depth: $\log_2 n$

Work at level:

- In the best and average case, the pivot splits the array into two nearly equal halves.
- The recursion depth is bounded by $O(\log n)$.
- At each level of the tree, the partitioning step takes linear time $O(n)$ total across all subproblems.
- Summing the work over all levels gives a total time complexity

Quick Sort Complexity & Trade-offs

Complexity Analysis

We analyze the time and space complexity of Quick Sort under different scenarios and compare it with other sorting algorithms.

- **Best Case:** $O(n \log n)$ when the pivot always splits the array into two equal halves ($T(n) = 2T(n/2) + O(n)$).
- **Average Case:** $O(n \log n)$ with random pivots, even with moderately unbalanced splits like 9-to-1 ratio.
- **Worst Case:** $O(n^2)$ when the pivot is always the smallest or largest element, producing highly skewed partitions ($T(n) = T(n-1) + O(n)$).
- **Space Complexity:** $O(\log n)$ auxiliary space on average for call stack. Can reach $O(n)$ in the worst case if not optimized via tail recursion elimination.

Advanced Optimizations

Performance Optimizations

Modern implementations of Quick Sort use several key optimizations to achieve peak performance in system libraries (e.g., glibc, Java's Arrays.sort).

- **Hybrid Sorting (Introsort):** Starts with Quick Sort, but switches to Heap Sort if the recursion depth exceeds a threshold (typically $2 \log n$) to guarantee $O(n \log n)$ worst-case performance.
- **Small Subarray Cutoff:** Switches to Insertion Sort for very small subarrays (e.g., size ≤ 10) because Insertion Sort has lower constant factors for small n.
- **Dual-Pivot Quick Sort:** Uses two pivots to partition the array into three parts. This reduces the number of cache misses and memory scans, and is the default sorting algorithm in Java 7+ for primitive types.

Data Structures (DI03000021)

GTU Diploma Engineering

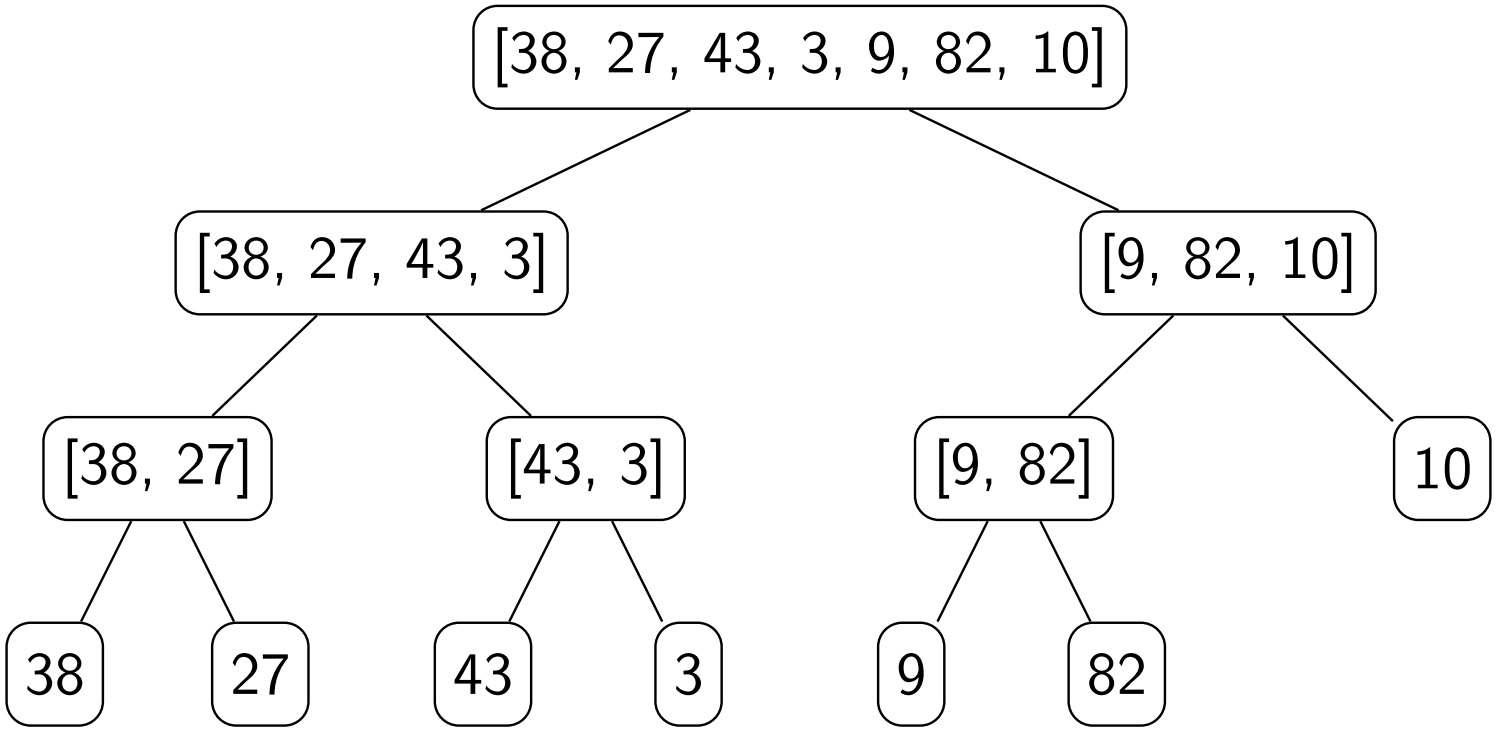
The Divide and Conquer Paradigm

Merge Sort

Merge Sort is a comparison-based sorting algorithm that uses the divide-and-conquer paradigm to achieve a guaranteed $O(N \log N)$ time complexity. It works by dividing the input array, sorting the halves recursively, and merging them.

- **Divide:** Split the N-element sequence into two subsequences of size $N/2$.
- **Conquer:** Sort the two subsequences recursively using merge sort.
- **Combine:** Merge the two sorted subsequences to produce the sorted answer.

Recursive Division of Array Elements



- Recursive splits continue until base-case subarrays of size 1 are reached.
- The division phase requires $O(\log N)$ steps.
- No elements are compared or swapped during the division phase; we merely compute mid $= l + (r - l)/2$.

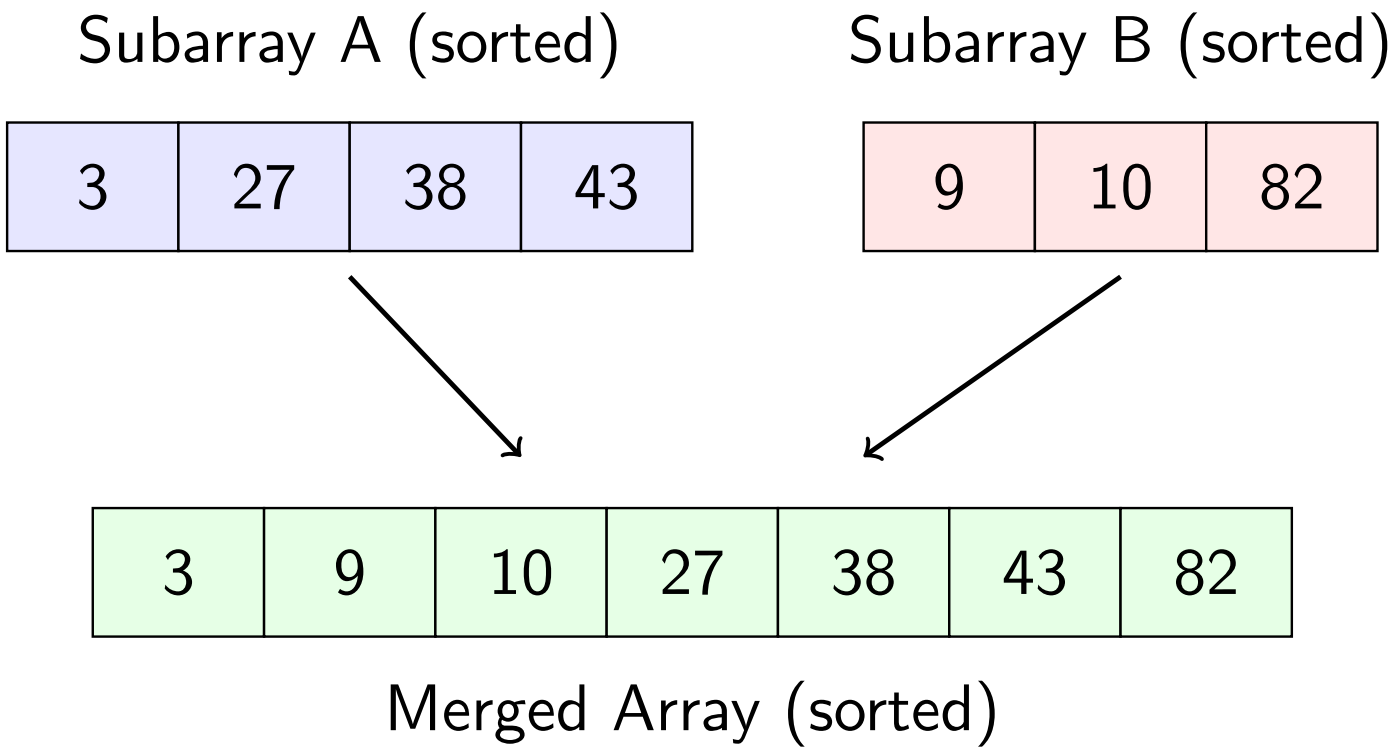
The Merge Procedure

Merge Operation

The core computational work of Merge Sort occurs in the merging step, where two sorted subarrays are combined into a single sorted array. This requires two pointer traversals and auxiliary space.

- Pointers i and j traverse the left and right subarrays, comparing elements.
- The smaller element is copied to a temporary auxiliary array, and its pointer is incremented.
- If one subarray is completed, the remaining elements of the other subarray are copied in bulk.
- The sorted temporary array is copied back into the original array space.

Visualizing the Merge Step



- Compares elements at pointer positions (e.g. 3 vs 9) to determine output order.
- Produces a single sorted array containing all elements from both subarrays.
- Preserves stable sorting order because if elements are equal, the element from the left subarray is chosen first.

Recursive Implementation in C++

```
void mergeSort(int arr[], int l, int r) {
  if (l < r) {
    int m = l + (r - l) / 2;
    mergeSort(arr, l, m);
    mergeSort(arr, m + 1, r);
    merge(arr, l, m, r);
  }
}
```

- The parameter `l` represents the left index and `r` represents the right index of the subarray.
- To prevent arithmetic overflow with large arrays, compute the midpoint using `l + (r - l) / 2`.
- Recursive calls partition the array until single-element segments (`l == r`) are reached.

The Merge Helper Function Implementation

```
void merge(int arr[], int l, int m, int r) {
  int n1 = m - l + 1;
  int n2 = r - m;
  std::vector<int> L(n1), R(n2);
  for (int i = 0; i < n1; i++) L[i] = arr[l + i];
  for (int j = 0; j < n2; j++) R[j] = arr[m + 1 +
  int i = 0, j = 0, k = l;
  while (i < n1 && j < n2) {
    if (L[i] <= R[j]) {
      arr[k++] = L[i++];
    } else {
      arr[k++] = R[j++];
    }
  }
  while (i < n1) arr[k++] = L[i++];
  while (j < n2) arr[k++] = R[j++];
}
```

Time Complexity Analysis

Recurrence Relation

The time complexity is calculated using the recurrence relation: $T(N) = 2T(N/2) + O(N)$. Solving this yields a constant time complexity of $O(N \log N)$ for best, worst, and average cases.

- Divide phase takes $O(1)$ time.
- Conquer phase solves two subproblems of size $N/2$, taking $2T(N/2)$ time.
- Combine (merge) phase takes linear $O(N)$ time.
- Since the recursion tree has $\log(N)$ levels and each level performs $O(N)$ work, the total time is always $O(N \log N)$.

Space Complexity and Memory Trade-offs

Memory Overhead

Standard Merge Sort requires $O(N)$ auxiliary space. This memory overhead is its primary disadvantage compared to in-place sorting algorithms like Quick Sort.

- Auxiliary space of $O(N)$ is required for storing temporary subarrays during merging.
- Call stack space of $O(\log N)$ is required to manage recursive call frames.
- In-place merge sort variants exist but significantly increase time complexity coefficients and implementation complexity.

Applications, Advantages & Disadvantages

Applications

Merge Sort is widely used in systems where stability is required, or when sorting linked structures and external datasets.

- Highly preferred for sorting Linked Lists because merge operations require only pointer manipulation with $O(1)$ extra space.
- Ideal for External Sorting (handling datasets larger than RAM) because of its sequential access patterns during merge steps.
- Disadvantage: Heavy memory overhead and high copying cost make it slower than Quick Sort for typical in-memory arrays.

Data Structures (DI03000021)

GTU Diploma Engineering

Core Philosophy of Radix Sort

Radix Sort

Radix Sort is a non-comparative integer sorting algorithm that avoids pairwise key comparisons. It operates by grouping keys by the individual digits that share the same significant position and value.

- Process keys digit-by-digit, either from Least Significant Digit (LSD) or Most Significant Digit (MSD).
- Requires an auxiliary, stable sorting algorithm (commonly Counting Sort) to sort the keys at each digit level.
- Bypasses the theoretical $\Omega(N \log N)$ lower bound constraint of comparison-based sorting algorithms.
- Exhibits linear time complexity when the number of digits per key is relatively constant.

Stable Subroutine: Counting Sort

```
void countingSort(int arr[], int n, int exp) {
    int output[n];
    int count[10] = {0};
    for (int i = 0; i < n; i++)
        count[(arr[i] / exp) % 10]++;
    for (int i = 1; i < 10; i++)
        count[i] += count[i - 1];
    for (int i = n - 1; i >= 0; i--) {
        output[count[(arr[i] / exp) % 10] - 1] =
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
```

- Digits are isolated using integer division and modulo arithmetic: `(arr[i] / exp) % 10`.

The LSD Radix Sort Driver Function

```
void radixSort(int arr[], int n) {
    int m = getMax(arr, n);
    for (int exp = 1; m / exp > 0; exp *= 10)
        countingSort(arr, n, exp);
}
```

- LSD (Least Significant Digit) Radix Sort starts by finding the maximum element to determine the number of digits (d).
- The loop runs d times, scaling `exp` by a factor of 10 in each iteration (1s place, 10s place, 100s place, etc.).
- If the maximum value has d digits, the time complexity of the driver is $\mathcal{O}(d \cdot (n + k))$.
- Auxiliary space complexity is $\mathcal{O}(n + k)$ due to the temporary arrays used in the Counting Sort subroutine.

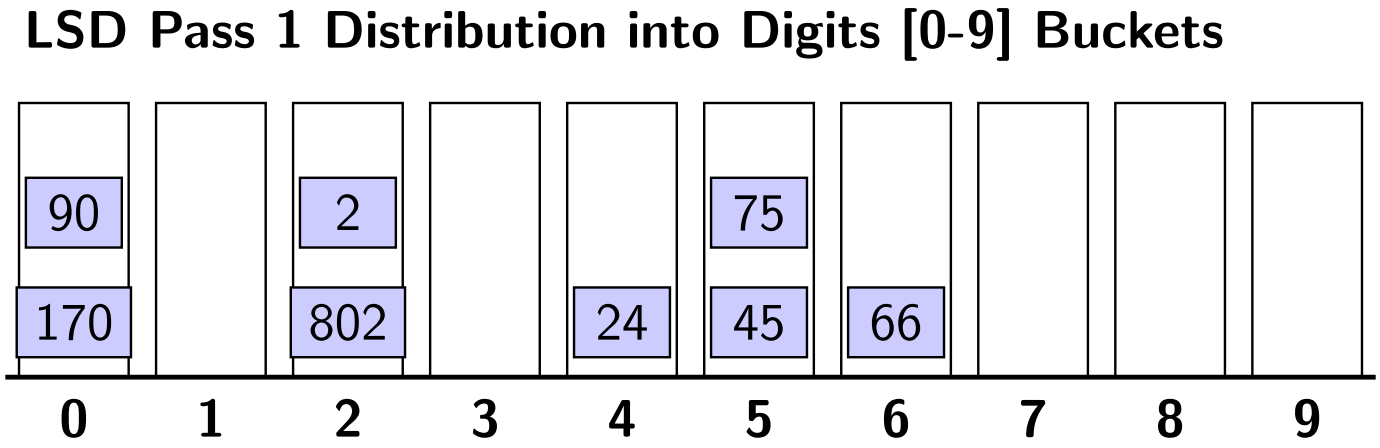
LSD Radix Sort Step-by-Step Example

Input Array

Input Array: [170, 45, 75, 90, 802, 24, 2, 66]
We sort this list based on base-10 digits, from right to left (LSD first).

- Pass 1 (1s Place): Sort by units digit. [170, 90, 802, 2, 24, 45, 75, 66]. Stability keeps 170 before 90, and 802 before 2.
- Pass 2 (10s Place): Sort by tens digit. [802, 2, 24, 45, 66, 170, 75, 90]. Note that 2 has a tens digit of 0.
- Pass 3 (100s Place): Sort by hundreds digit. [2, 24, 45, 66, 75, 90, 170, 802]. Complete sorted order is achieved.

LSD Pass 1 Bucket Distribution



- Visual representation of elements distributed into 10 buckets based on their least significant digit (1s place).
- Stacking order in the buckets shows the stable distribution before collecting back into the main array.
- Buckets 1, 3, 7, 8, and 9 remain empty in this pass.

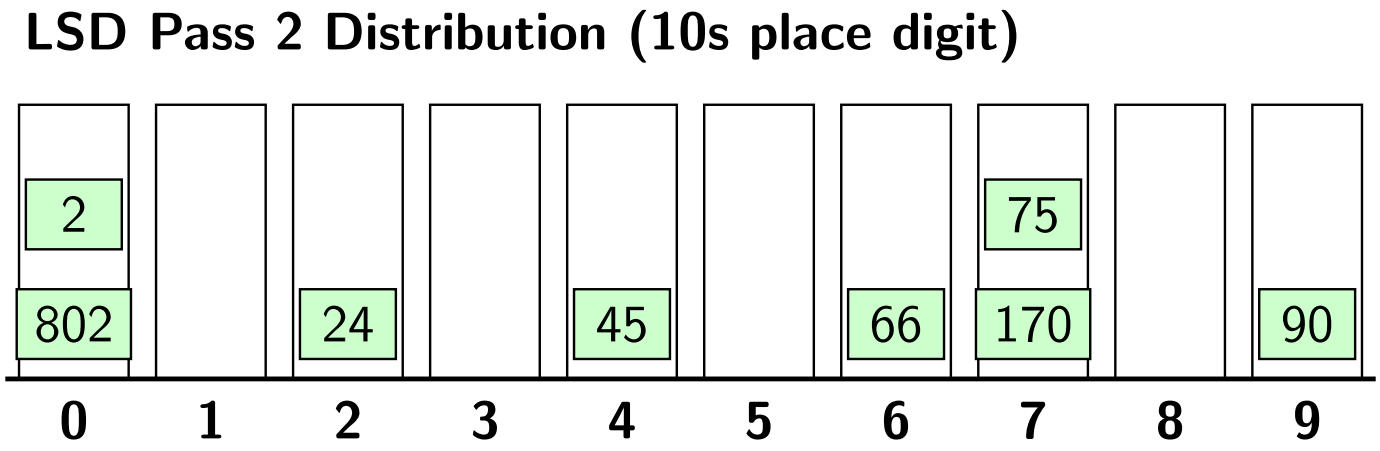
Complexity Analysis & Performance

Parameters

Let n be the number of keys, d be the number of digits in the keys, and k be the radix (base of the representation).

- Time Complexity: $\mathcal{O}(d \cdot (n + k))$ for both average and worst cases.
- If keys are in the range $[0, n^c - 1]$ for some constant c, and we choose $k = n$, Radix Sort runs in $\mathcal{O}(n)$ linear time.
- Space Complexity: $\mathcal{O}(n + k)$ auxiliary memory space needed to hold output and count structures.
- Practical considerations: Radix Sort has higher constant factors than Quick Sort, and it does not make optimal use of CPU cache locality.

LSD Pass 2 Bucket Distribution



- Visual representation of elements distributed into buckets based on the tens digit.
- Single digit elements like 2 have a tens digit of 0, grouping them with 802.
- Collecting from left to right yields: [802, 2, 24, 45, 66, 170, 75, 90].

LSD Radix Sort vs. MSD Radix Sort

Comparison Overview

Radix sorting algorithms are categorized by the direction of processing: Least Significant Digit (LSD) vs. Most Significant Digit (MSD).

- LSD Radix Sort: Iterates from right-to-left. Extremely simple to implement iteratively, requires a stable sub-sorting algorithm.
- MSD Radix Sort: Iterates from left-to-right. Partitioning begins with the most significant digit.
- MSD is typically recursive and uses divide-and-conquer to sort partitioned buckets individually.
- MSD can sort variable-length keys (such as words/strings) efficiently, terminating early on unique prefixes.

Summary & Practical Applications

Summary

Radix Sort provides linear sorting capabilities in domains where keys can be decomposed into stable integer digit representations.

- Excellent for sorting fixed-width values like IP addresses, 32-bit integers, zip codes, and dates.
- Forms the theoretical foundation of mechanical card-sorting devices used historically.
- Limitation: Unsuitable for general object sorting where comparison keys cannot be cleanly decomposed into distinct components.
- Requires $\mathcal{O}(N)$ extra space which may restrict deployment in low-memory embedded devices.

Data Structures (DI03000021)

GTU Diploma Engineering

Searching Basics and Problem Definition

T

he searching problem: Given a collection of N elements and a target key K, find an index i such that $A[i] = K$. If K is not present, return a sentinel value, typically -1.

- **Internal searching:** The collection resides completely in main memory (RAM).
- **External searching:** The collection resides on disk/secondary storage, requiring block access.
- **Static vs Dynamic datasets:** Static arrays favor binary search, whereas dynamic lists may require other structures like search trees.
- **Key types:** Searches can be on primitive values or complex object records using a primary key.

Linear Search: Algorithm and Intuition

L

inear search (or sequential search) starts at the beginning of the collection and checks each element sequentially until the target key is found or the end of the collection is reached.

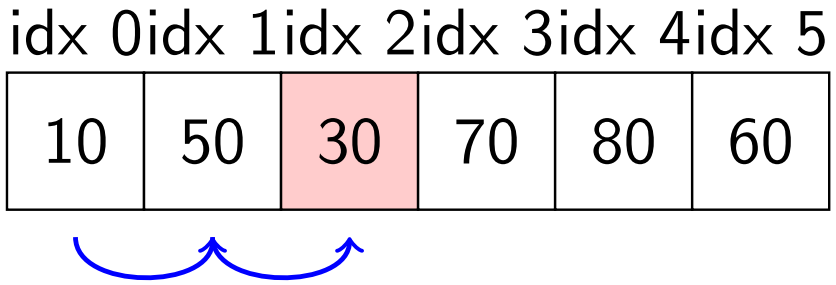
- **Order-agnostic:** Does not require the array elements to be sorted.
- **Data structures:** Can be implemented on arrays, singly linked lists, and doubly linked lists.
- **Cache friendliness:** High spatial locality due to contiguous memory access in arrays.
- **Comparisons:** In the worst case, it makes exactly N comparisons.

Linear Search: C++ Implementation

```cpp
int linearSearch(int arr[], int n, int target) {
    for (int i = 0; i < n; i++) {
        if (arr[i] == target) {
            return i; // Target found, return ind
        }
    }
    return -1; // Target not found
}
```

- **Time Complexity:** Best Case is $\mathcal{O}(1)$ (found at index 0); Worst Case is $\mathcal{O}(N)$ (not found or at index $N-1$).
- **Average Case:** $\mathcal{O}(N)$ since on average it checks $N/2$ elements.
- **Space Complexity:** $\mathcal{O}(1)$ auxiliary space (only uses a few loop variables).
- **Loop termination condition:** `i < n` ensures we do not read past array boundaries.

Visualizing Linear Search



Searching for target = 30. Compares 10, then 50, then matches 30.

- Sequential scan starts at index 0.
- Index 0: $10 \neq 30$ (continue search).
- Index 1: $50 \neq 30$ (continue search).
- Index 2: $30 == 30$ (match found, return index 2).

Binary Search: Divide-and-Conquer Paradigm

B

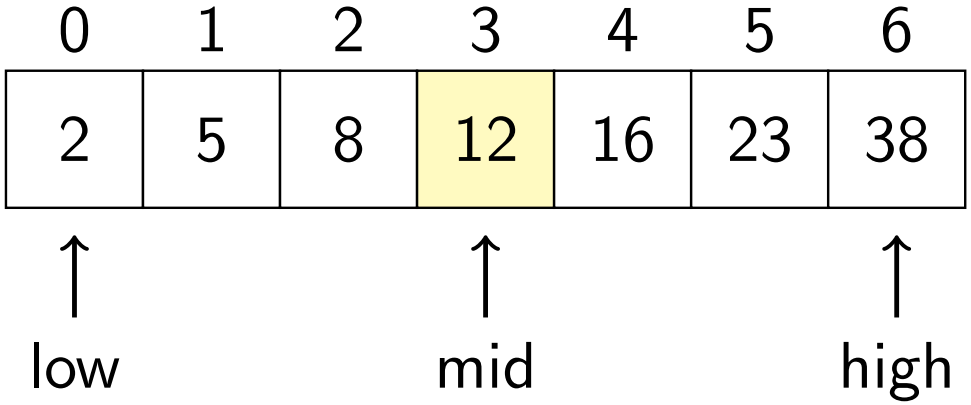
inary search is an interval-halving search algorithm. It relies on the sorting property: if the target is less than the middle element, it must reside in the lower half; if it is greater, it must reside in the upper half.

- **Precondition:** The sequence must be sorted (ascending or descending order).
- **Random Access:** Requires $\mathcal{O}(1)$ access time to elements (e.g., array). Cannot be efficiently implemented on standard Linked Lists (takes $\mathcal{O}(N)$ to find mid).
- **Decision tree:** The logic is modelled as a binary decision tree of height $\mathcal{O}(\log N)$.
- **Interval reduction:** Every comparison reduces the search space by half.

Binary Search: Iterative Implementation

```
int binarySearch(int arr[], int n, int target) {
    int low = 0;
    int high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target) {
            return mid; // Element found
        }
        if (arr[mid] < target) {
            low = mid + 1; // Search right half
        } else {
            high = mid - 1; // Search left half
        }
    }
    return -1; // Element not found
}
```

Visualizing Binary Search



Target = 23. mid (12) ¡ 23. Next range is [mid+1, high] (indices 4 t

- Initial search space: index 0 to 6. `mid` is index 3 (value 12).
- Since $12 < 23$, the target must be in the right half.
- Update: `low` index shifts to 4. New search space is indices 4 to 6.
- Subsequent `mid` at index 5 (value 23) matches target exactly.

Binary Search: Recursive Implementation

```
int recursiveBinarySearch(int arr[], int low, int
    if (low > high) {
        return -1; // Base case: not found
    }
    int mid = low + (high - low) / 2;
    if (arr[mid] == target) {
        return mid; // Base case: found
    }
    if (arr[mid] < target) {
        return recursiveBinarySearch(arr, mid + 1
    } else {
        return recursiveBinarySearch(arr, low, mi
    }
}
```

- **Call Stack Space:** $\mathcal{O}(\log N)$ auxiliary space due to recursive stack frames.

Algorithm Comparison and Summary

C

omparison of Searching Algorithms:

- **Linear Search:** Time $\mathcal{O}(N)$, Space $\mathcal{O}(1)$, Precondition: None.
- **Binary Search:** Time $\mathcal{O}(\log N)$, Space $\mathcal{O}(1)$ iterative / $\mathcal{O}(\log N)$ recursive, Precondition: Sorted collection.
- **Selection rule:** Choose Linear Search for unsorted arrays of small size ($N < 100$) or linked lists. Choose Binary Search for large sorted collections or collections searched frequently.

- Binary Search is exponentially faster for large N: $\mathcal{O}(\log N)$ vs $\mathcal{O}(N)$.
- **Sorting Overhead:** If a collection is unsorted and only searched once, Linear Search $\mathcal{O}(N)$ is faster than sorting

Data Structures (DI03000021)

GTU Diploma Engineering

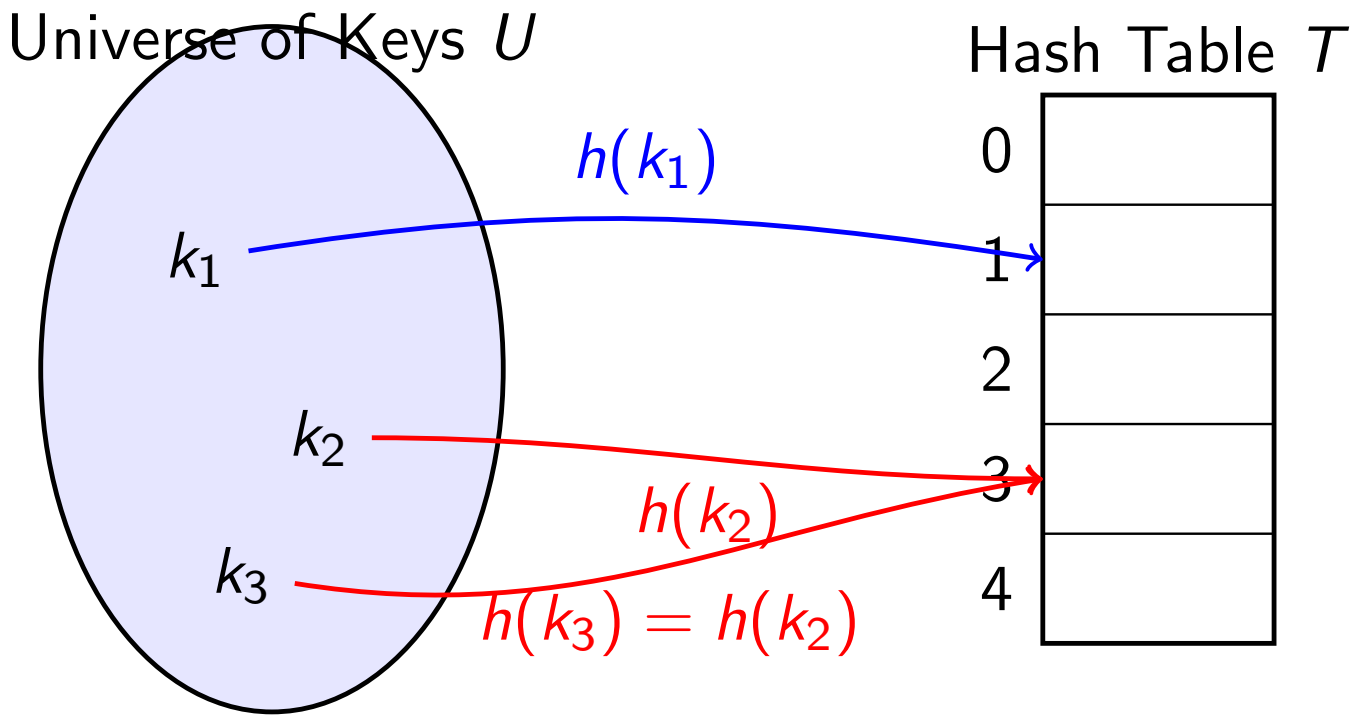
What is a 'Key' in Hashing?

I

n hashing, a key is the input data that uniquely identifies a record or value in a collection. It undergoes a mathematical transformation to map to an index in a hash table.

- Keys can be of various data types: integers, strings, custom objects, or composite data.
- The domain of all possible keys is called the keyspace (U). Typically, $|U| \gg N$, where N is the number of keys actually stored.
- The hashing process maps a key $k \in U$ to a bucket/slot index $h(k)$ in the range $[0, M-1]$, where M is the hash table size.
- The hash function $h(k)$ must be deterministic: for any given key k, it must always produce the same slot index $h(k)$.

Visualizing Keyspace Mapping



- The universe of possible keys U is generally much larger than the slot capacity M of the hash table.
- A collision occurs when two distinct keys $k_i \neq k_j$ hash to the exact same slot, i.e., $h(k_i) = h(k_j)$.
- The diagram shows k_2 and k_3 colliding at slot 3, while k_1 maps uniquely to slot 1.
- Since $|U| > M$, the Pigeonhole Principle guarantees that collisions are mathematically inevitable.

Essential Properties of a Key Hash Function

T

o achieve efficient $O(1)$ average-case operations, a hash function must satisfy several key criteria.

- **1. Determinism:** A given key k must always map to the same slot index $h(k)$ during the program's lifecycle.
- **2. Fast Computation:** Calculating $h(k)$ should be $O(1)$ or linear in terms of the key's size, keeping overhead minimal.
- **3. Uniform Distribution:** It must map keys to slots uniformly, minimizing clustering and collisions (Simple Uniform Hashing Assumption).
- **4. High Avalanche Effect:** A tiny change in the key (e.g., flipping a single bit) should produce a completely different hash value.

Hashing Integer Keys

I

ntegers are the most straightforward keys to hash. There are two primary classical methods: the Division Method and the Multiplication Method.

- **Division Method:** $h(k) = k \bmod M$. For best results, M should be a prime number not close to a power of 2 or 10.
- Choosing prime M helps distribute keys more uniformly when there are patterns (like common factors) in the input keys.
- **Multiplication Method:** $h(k) = \lfloor M(kA \bmod 1) \rfloor$, where A is a constant in the range $0 < A < 1$.
- Knuth suggested using the Golden Ratio conjugate: $A = (\sqrt{5} - 1)/2 \approx 0.6180339887$, which works exceptionally well regardless of M.

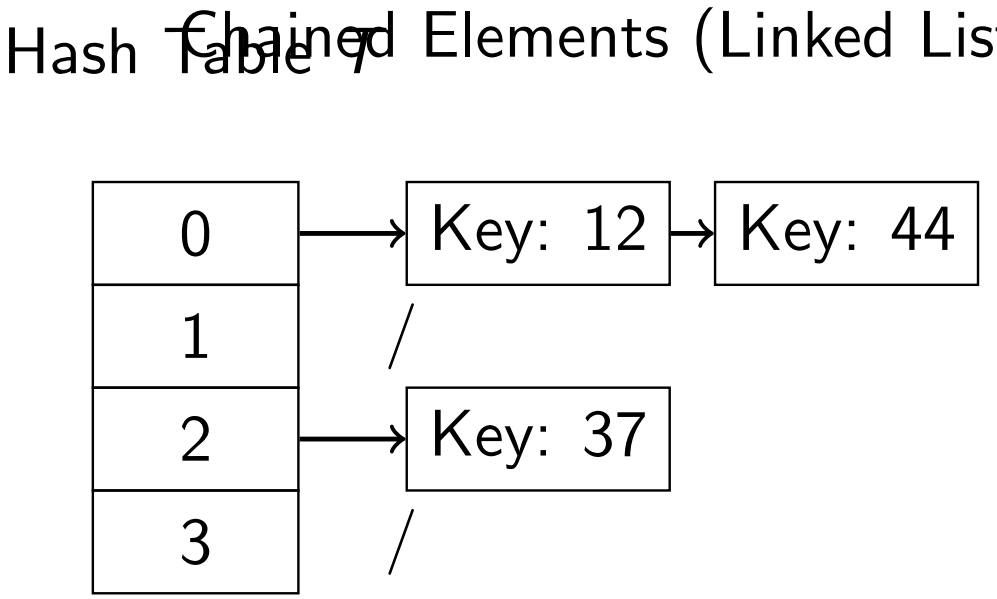
Hashing Variable-Length String Keys

S

trings are variable-length and require processing each character. A simple sum of ASCII values is a poor hash function because anagrams collide.

- **Polynomial Rolling Hash:** $h(s) = (\sum_{i=0}^{L-1} s[i] \cdot p^i) \bmod M$, where p is a prime and M is a large prime or power of 2.
- For English lowercase strings, $p = 31$ or $p = 33$ are standard choices, representing the base of the alphabet plus a small margin.
- Java's `String.hashCode()` uses a similar formula: $h(s) = \sum_{i=0}^{L-1} s[i] \cdot 31^{L-1-i}$.
- The rolling hash can be computed efficiently in $O(L)$ time using Horner's rule to avoid exponentiation overhead: `hash = (hash * p + s[i]) % M`.

Collision Resolution: Separate Chaining



- Separate chaining stores colliding keys in an auxiliary data structure, typically a singly linked list.
- Worst-case search time becomes $O(K)$ where K is the chain length. If N keys are uniformly distributed, average chain length is $\alpha = N/M$ (load factor).
- With a good hash function, the average search and delete operations take $O(1 + \alpha)$ time.
- Drawbacks include pointer overhead and poor cache locality compared to open addressing techniques.

Cryptographic vs. Non-Cryptographic Hashing

D

epending on the application, keys are hashed using either cryptographic or non-cryptographic hash functions. They have distinct design goals.

- **Non-Cryptographic** (e.g., MurmurHash, FNV-1a, xxHash): Optimized purely for speed and uniform distribution in hash tables.
- **Cryptographic** (e.g., SHA-256, SHA-3, bcrypt): Designed to be computationally infeasible to invert (one-way property).
- **Pre-image resistance:** Given $h(k)$, it should be extremely hard to find k.
- **Collision resistance:** It must be practically impossible to find any two different keys $k_1 \neq k_2$ such that $h(k_1) = h(k_2)$.

The Birthday Paradox and Collisions

A

common misconception is that collisions are rare if the hash space is large. Probability theory proves otherwise.

- The Birthday Paradox shows that in a room of 23 people, there is a $> 50\%$ chance that two share a birthday ($M = 365$).
- In general, if we map N keys to M slots, a collision is highly likely when $N \approx \sqrt{M}$.
- For a 32-bit hash function ($M = 2^{32} \approx 4.3 \times 10^9$), a collision is expected with 50% probability after hashing only $\approx 77,000$ keys.
- This mathematical reality underlines the absolute necessity of robust collision resolution strategies in all hash tables.

Summary and Key Takeaways

U

nderstanding how keys map to hash values is critical to designing efficient, secure, and robust systems.

- Select hash functions tailored to the key type (e.g., MurmurHash for strings/binary data, multiplication method for integers).
- Keep the load factor $\alpha = N/M$ low (typically $\alpha < 0.7$ for open addressing, $\alpha < 1.0$ for chaining) by resizing/rehashing.
- Always ensure your custom objects override both `hashCode()` and `equals()` consistently (e.g., in Java).
- Consider security requirements: if user-input keys can trigger worst-case $O(N)$ attacks (Hash DoS), use randomized seed hashing.

Data Structures (DI03000021)

GTU Diploma Engineering

Direct-Address Tables and the Need for Hashing

Direct-Address Table: Works when the universe of keys U is small. Each key $k \in U$ maps directly to slot k in array T, providing $O(1)$ worst-case time complexity for Insert, Delete, and Search operations.

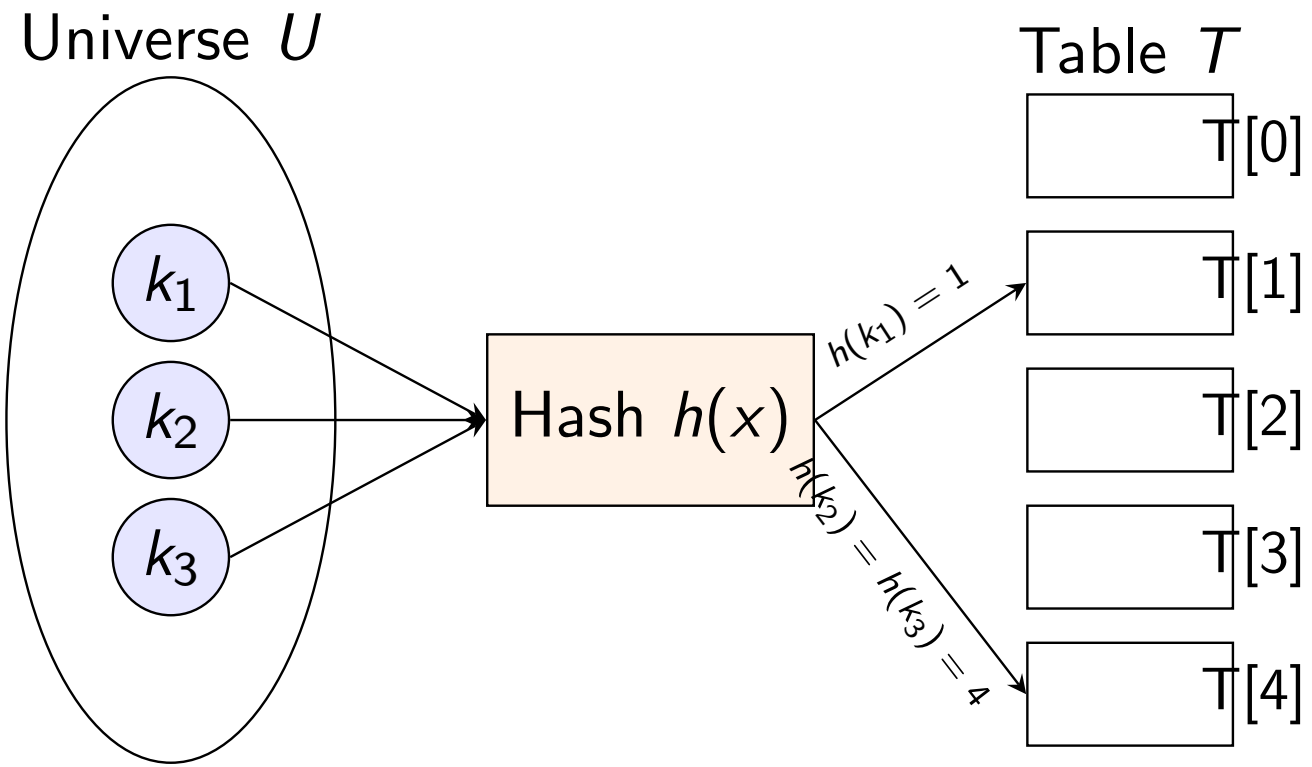
- Direct addressing requires $O(|U|)$ memory space, which is prohibitive for large key universes (e.g., 64-bit keys require 2^{64} slots).
- Hash tables solve this by using a hash function $h : U \to \{0, 1, \ldots, m-1\}$ to map keys into a table of size m, where $m \ll |U|$.
- The primary challenge of hash tables is handling collisions, where two distinct keys map to the exact same slot: $h(k_i) = h(k_j)$ for $k_i \neq k_j$.

Designing Effective Hash Functions

Simple Uniform Hashing Assumption: Each key is equally likely to hash to any of the m slots, independently of where other keys hash.

- **Division Method:** $h(k) = k \pmod m$. Fast, but performance relies heavily on choosing m as a prime not close to a power of 2 to avoid pattern-based collisions.
- **Multiplication Method:** $h(k) = \lfloor m(kA \pmod 1) \rfloor$, where $0 < A < 1$. Knuth suggests $A \approx (\sqrt{5} - 1)/2 \approx 0.6180339887$. Less sensitive to the choice of m.
- **Polynomial Rolling Hash (for Strings):** $h(s) = \left(\sum_{i=0}^{n-1} s[i] \cdot p^i \right) \pmod m$, where p is a small prime (e.g., 31 or 53).

Visualizing Hashing and Collisions



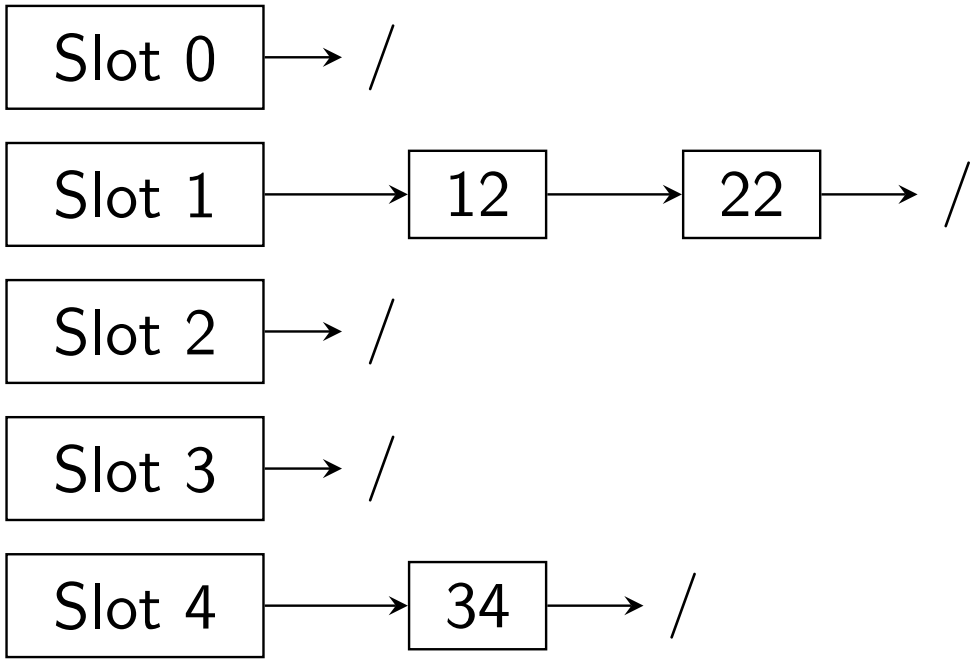
- The universe of keys U is mapped to a smaller table of size $m = 5$.
- Keys k_2 and k_3 map to the same slot index 4, demonstrating a collision.
- Collision resolution strategies must be employed to handle index conflicts dynamically.

Collision Resolution by Chaining

Chaining (Closed Addressing): Place all elements that hash to the same slot into a linked list. Each hash table slot contains a pointer to the head of its corresponding list.

- Insertion takes $O(1)$ time by prepending the new element to the linked list.
- Worst-case search time is $O(n)$ if all n keys hash to the same slot, reducing the table to a single linked list.
- Average-case search time is $O(1 + \alpha)$, where $\alpha = n/m$ is the load factor of the hash table.
- If $n = O(m)$, then $\alpha = O(1)$, yielding $O(1)$ average-case operations.

Visualizing Collision Resolution by Chaining



- Slots 0, 2, and 3 are empty and point to NIL (/ symbol).
- Slot 1 has a linked list containing keys 12 and 22, representing collision resolution where $h(12) = h(22) = 1$.
- Deletion in chaining is performed by updating pointer references in $O(1)$ time if the linked list is doubly linked.

Open Addressing: Linear Probing

Open Addressing: All elements occupy the hash table itself without external linked lists. When a collision occurs, the table systematically probes slots until an empty one is found.

Linear Probing Recurrence: $h(k, i) = (h'(k) + i) \pmod{m}$, where $h'(k)$ is an auxiliary hash function and $i = 0, 1, \ldots, m-1$.

- Linear probing searches sequential slots: $h'(k), h'(k)+1, h'(k)+2, \ldots$
- Suffers from **primary clustering**: long runs of occupied slots build up, increasing search and insertion times.
- Deletion requires placing a special 'DELETED' marker instead of leaving the slot empty to prevent breaking search probe

Open Addressing: Quadratic Probing and Double Hashing

- **Quadratic Probing:** $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \pmod{m}$, where $c_1, c_2 \neq 0$ are constants. Eliminates primary clustering but exhibits mild secondary clustering.
- **Double Hashing:** $h(k,i) = (h_1(k) + i \cdot h_2(k)) \pmod{m}$. One of the best open addressing methods as it generates $\Theta(m^2)$ probe sequences.
- In double hashing, $h_2(k)$ must be relatively prime to m so the probe sequence can cover the whole table. A typical choice is prime m with $h_2(k) = 1 + (k \pmod{m-1})$.
- Double hashing eliminates both primary and secondary clustering, performing very close to uniform hashing.
- Open addressing strictly requires load factor $\alpha = n/m \leq 1$ at all times.

Mathematical Analysis of Hash Tables

Load Factor (α): $\alpha = n/m$, representing the average number of elements stored per slot.

- **Chaining Performance:**
  - Unsuccessful search average time: $\Theta(1 + \alpha)$
  - Successful search average time: $\Theta(1 + \alpha/2)$
- **Open Addressing Performance (Uniform Hashing):**
  - Unsuccessful search average probes: $\leq \frac{1}{1-\alpha}$
  - Successful search average probes: $\leq \frac{1}{\alpha} \ln \frac{1}{1-\alpha}$
- As $\alpha \to 1$ in open addressing, probe counts head to infinity (e.g., $\alpha = 0.9 \implies$ 10 probes).
- Dynamic resizing (rehashing) doubles table size taking $O(n)$ time overall, but amortizing to $O(1)$ per insertion.

Perfect Hashing and Practical Applications

Perfect Hashing: Designed for static key sets fixed in advance to guarantee $O(1)$ worst-case search time using a two-level hashing scheme.

- Secondary level tables use $m_i = n_i^2$ slot allocations to keep collision probability strictly below $1/2$.
- Total expected space remains $O(n)$ despite quadratic sizing of secondary tables.
- **Practical Applications:** Databases (hash joins, indexing), compilers (symbol tables), memory caching, and cryptographic hashing.

Data Structures (DI03000021)

GTU Diploma Engineering

Introduction to Hashing & Mapping

Hashing

A hash function maps keys to indices in a hash table.

- Ideal hash tables achieve $O(1)$ average time complexity for search, insert, and delete operations.
- The universe of potential keys U is typically much larger than the number of table slots m.
- A collision occurs when two distinct keys hash to the identical index, meaning $h(k_1) = h(k_2)$ where $k_1 \neq k_2$.
- A uniform distribution reduces collision probability and prevents degradation of performance.

The Division Method Formula

Division Method

The Division Method maps a key k into one of m slots by taking the remainder of k divided by m.

- Mathematical formula: $h(k) = k \bmod m$.
- Extremely fast computation: requires only a single modulo operation, translating to minimal CPU instructions.
- Maps arbitrary integer keys directly to table indices in the range $[0, m-1]$.
- Non-integer keys must be preprocessed into integers (e.g., base-256 string conversion).

Choosing Table Size m: Powers of 2

Powers of Two Pitfall

Selecting $m = 2^p$ makes division hashing highly vulnerable to bias.

- If $m = 2^p$, then $h(k)$ is simply the p lowest-order bits of the key k.
- This ignores all higher bits, making the hash function completely blind to their variation.
- If keys exhibit patterns in their lower bits, massive clustering and collisions will occur.
- Example: With $m = 8$ (2^3), keys 12 (1100_2) and 20 (10100_2) both map to slot 4.

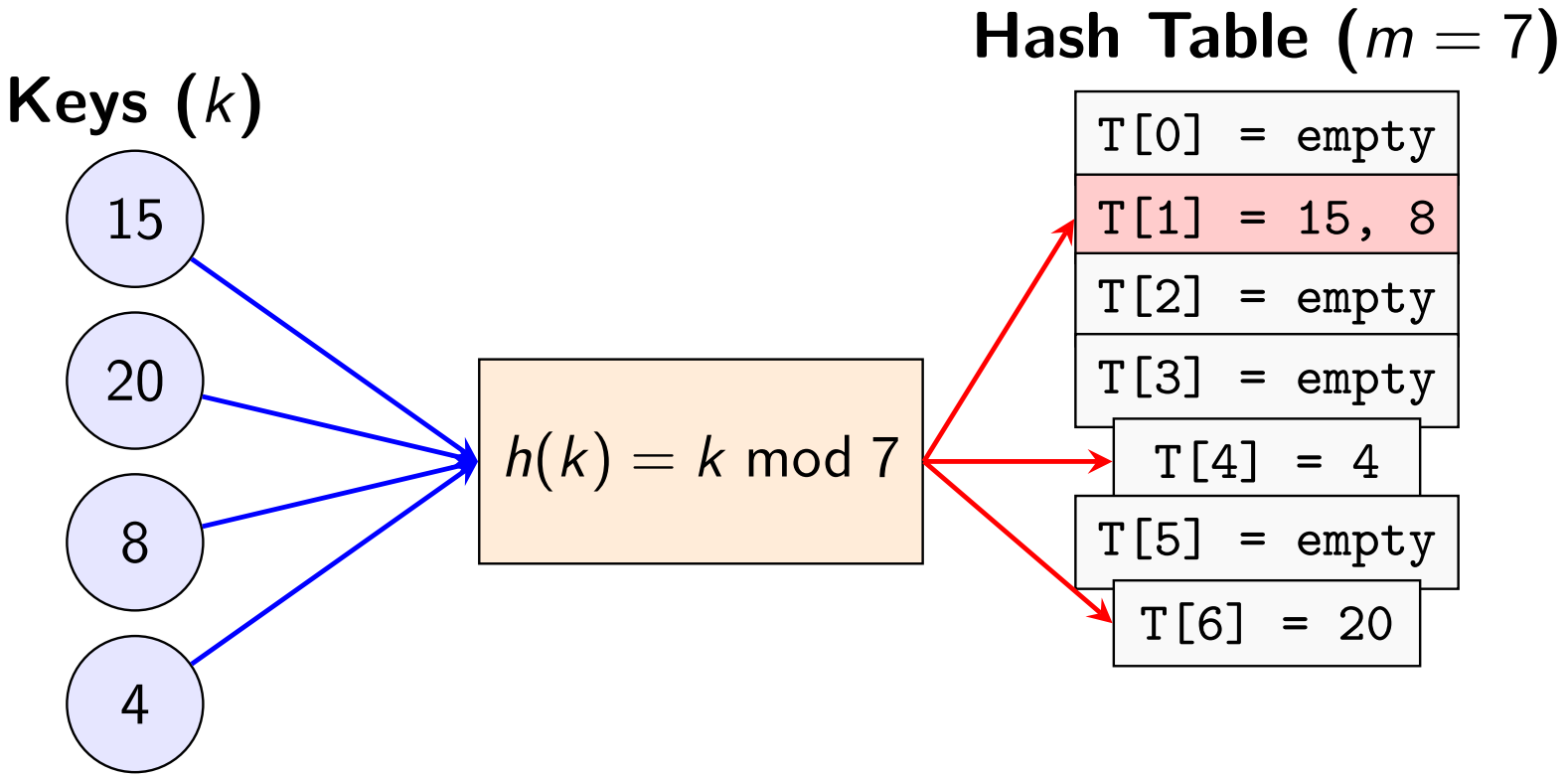
Choosing Table Size m: Prime Numbers

Optimal Table Sizing

A prime number m not close to a power of 2 is the optimal choice for the Division Method.

- A prime table size ensures that the hash value depends on all bits of the key.
- It mitigates patterns in keys that share common factors with table dimensions.
- Avoid primes close to powers of two (e.g., $m = 2^p - 1$ or $m = 2^p + 1$).
- Example: 701 is an excellent prime size for a table designed to hold around 500–600 elements.

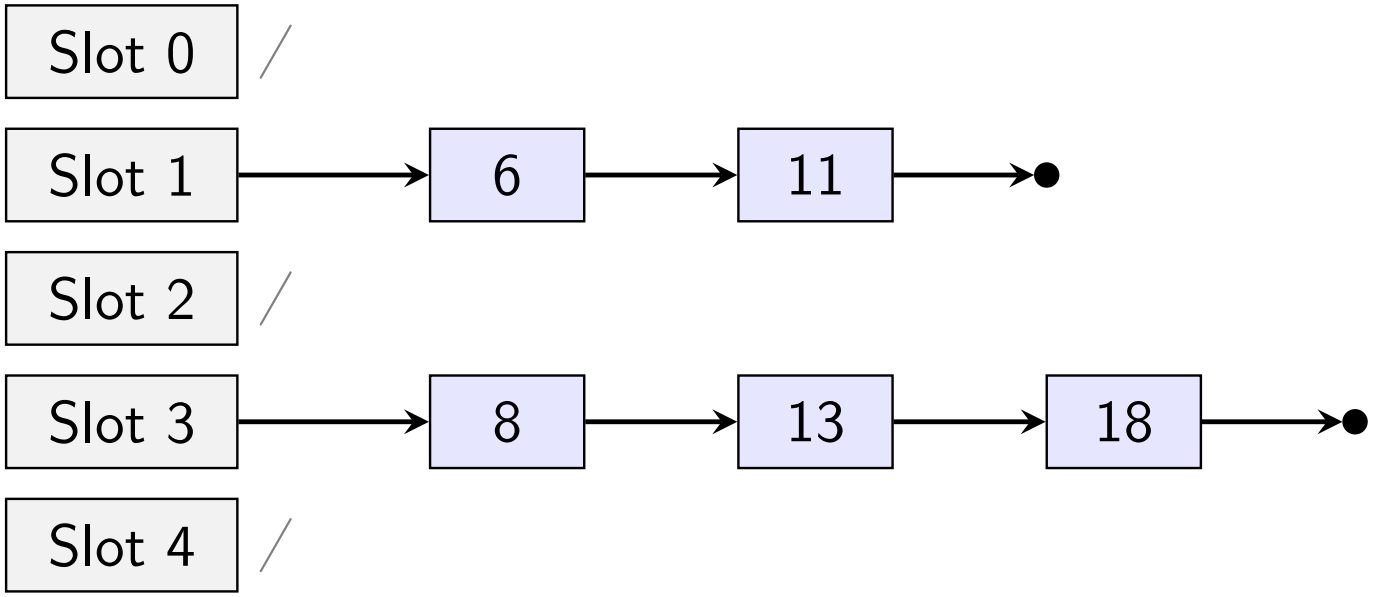
Visualizing the Remainder Mapping



- Visual representation of mapping keys $\{15, 20, 8, 4\}$ into table size $m = 7$.
- Computations: $15 \bmod 7 = 1$, $20 \bmod 7 = 6$, $8 \bmod 7 = 1$, $4 \bmod 7 = 4$.
- A collision occurs at index 1 because both 15 and 8 map to $T[1]$.
- Collisions must be resolved using techniques like chaining or

Collision Resolution: Chaining Example

Table Index ($m = 5$)



- Chaining stores all colliding keys in a linked list associated with each slot.
- In this example with $m = 5$, slot 1 holds $\{6, 11\}$ and slot 3 holds $\{8, 13, 18\}$.
- New keys are inserted at the head or tail of the list in $O(1)$ time.
- Worst-case search time becomes $O(n)$ if all elements hash to the same slot.

Pros and Cons of Division Hashing

Tradeoffs

Evaluating performance tradeoffs and implementation realities.

- **Pro: Speed.** Modulo operation is hardware-accelerated on modern CPUs.
- **Pro: Minimal code.** Implemented easily as `key % m` in standard utilities.
- **Con: Table Size Sensitivity.** Highly dependent on selecting a good table size m to avoid performance degradation.
- **Con: Vulnerable to Patterns.** Susceptible to clustering if key distribution matches patterns in table sizes.

C++ Implementation & Practices

```
class HashTable {
    int m;
    vector<list<int>> table;
public:
    HashTable(int size) : m(size), table(size) {}
    int hashFunction(int key) {
        return (key % m + m) % m;
    }
    void insert(int key) {
        int index = hashFunction(key);
        table[index].push_back(key);
    }
};
```

- Uses `std::vector` of `std::list` for simple chaining implementation.
- Safe modulo implementation: `(key % m + m) % m` handles

Summary & Advanced Alternatives

Summary

Summary of the Division Method and lookahead to alternative hash functions.

- Division method is simple and fast, but requires careful selection of prime table sizes.
- **Alternative:** Multiplication method, where table size m is not critical.
- **Alternative:** Universal hashing, selecting a hash function at random from a family.
- In practice, modern engines use MurmurHash, CityHash, or FNV-1a for general purposes.

Data Structures (DI03000021)

GTU Diploma Engineering

The Hashing Concept & Uniformity

Hashing Concept

A hash function maps keys to array indices. Ideal properties include fast computation $O(1)$ and uniform distribution to minimize collisions.

- Given key K and table size M, $h(K)$ maps to $[0, M-1]$.
- Collisions occur when $h(K_1) = h(K_2)$ for $K_1 \neq K_2$.
- Simple division method $h(K) = K \pmod{M}$ is highly sensitive to the choice of M (e.g., poor distribution if M shares factors with radix).
- Mid-Square and Folding methods help reduce patterns in key distribution.

Mid-Square Method: Definition and Steps

Mid-Square Method

The Mid-Square method squares the key value, and then extracts a middle portion of the digits as the hash address.

- Step 1: Square the key K to get K^2. This spreads the influence of all digits across the resulting value.
- Step 2: Determine the number of digits r required for the address space 10^r (or 2^r in binary).
- Step 3: Extract the middle r digits (or bits) of K^2 as the hash value $h(K)$.
- Since the middle digits of a square depend on all digits of the original key, keys with similar prefixes or suffixes map to different locations.

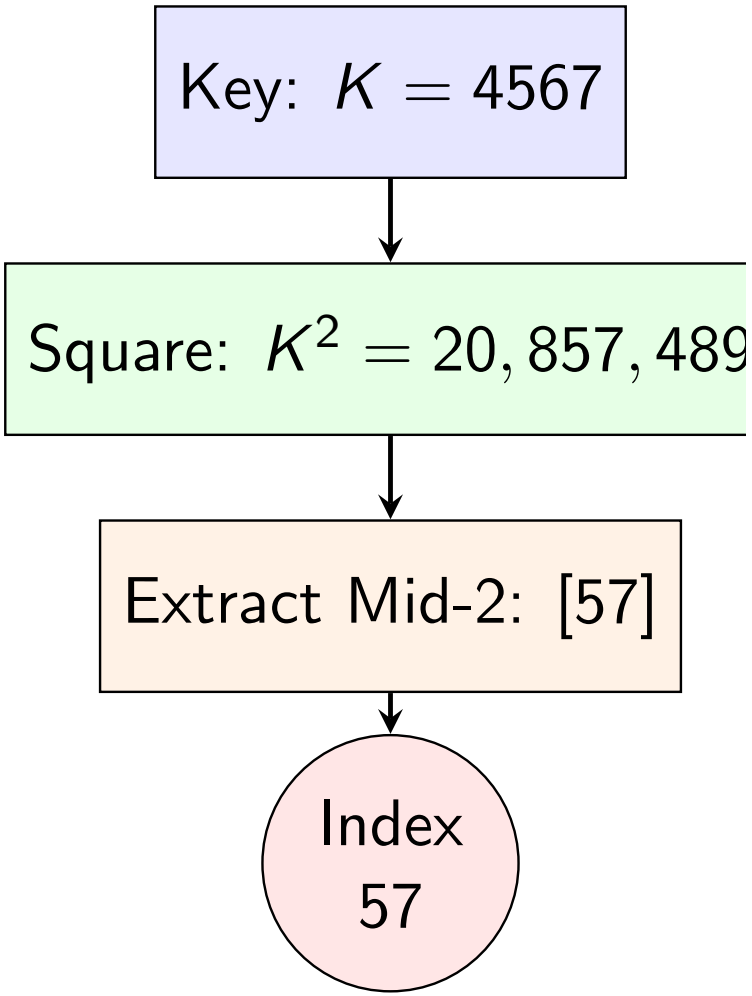
Step-by-Step Example of Mid-Square

Example Parameters

Key $K = 4567$, Table Size $M = 100$ (requires 2 digits, index 00–99).

- Calculate K^2: $4567 \times 4567 = 20,857,489$.
- The result $20,857,489$ has 8 digits.
- To get 2 digits, we extract the middle digits. The middle positions of an 8-digit number are digits at positions 4 and 5 (indexing from right/left).
- Let's extract the middle 2 digits from $20,[85],7,489$ or $20,8[57],489$. Using center digits, we get 57 (or 85, depending on convention).
- Thus, $h(4567) = 57$. Notice how changing a single digit in K (e.g., $4568 \rightarrow 20,866,624$) yields a completely different middle segment 66.

Mid-Square Hashing Process



Mid-Square spreads influences of all digits.

- Visual representation of key mapping using Mid-Square.
- The squaring step is critical because every digit of the original key contributes to the middle digits of the squared value.
- Extraction acts as a pseudo-random distribution mechanism.

Implementing Mid-Square Method in C

```c
int midSquareHash(int key, int tableSize) {
    long long square = (long long)key * key;
    // Extract middle digits
    // For a table of size 100, we need 2 digits
    // If square is 20857489, (20857489 / 1000) %
    long long temp = square / 1000;
    int hashValue = temp % 100;
    return hashValue;
}
```

- C code uses `long long` to prevent integer overflow during squaring of large key values.
- The division operations isolate the target middle segment mathematically.
- Time complexity of this function is $O(1)$ arithmetic operations, making it extremely fast.

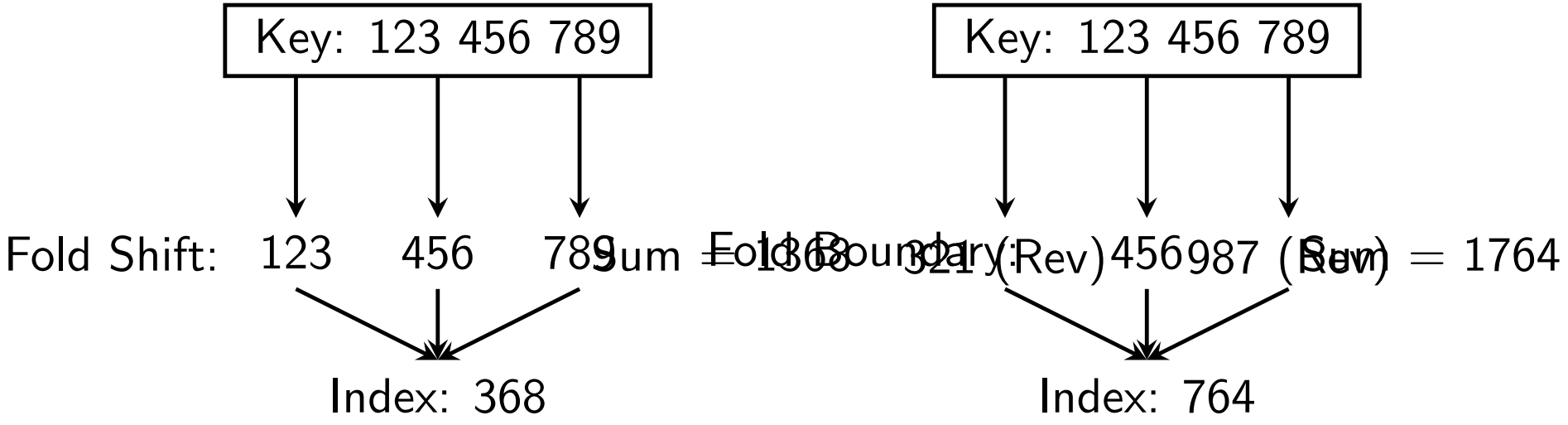
The Folding Method: Definition and Types

Folding Method

The folding method partitions the key into several parts, adds these parts together, and uses the last few digits as the hash address.

- Step 1: Partition the key K into sub-keys of equal size (matching the address size of the table).
- Step 2: Apply a folding strategy: Fold Shift (direct addition) or Fold Boundary (reverse alternate parts before addition).
- Step 3: Sum the parts together.
- Step 4: Truncate or modulo-divide the sum to fit the table size.
- This method is useful when keys are longer than the natural integer word size of the machine (e.g., ISBNs, Social Security Numbers).

Fold Shift vs. Fold Boundary Diagram



- Fold Shift simply adds the segmented digit blocks together.
- Fold Boundary reverses alternating blocks (like folding a strip of paper accordion-style) before adding.
- Reversing boundaries helps preserve randomness when patterns exist in key prefixes/suffixes.

Implementing Folding Method in C

```
int foldShiftHash(char* keyStr, int groupSize, in
    int sum = 0;
    int len = strlen(keyStr);
    for (int i = 0; i < len; i += groupSize) {
        int part = 0;
        for (int j = 0; j < groupSize; j++) {
            if (i + j < len) {
                part = part * 10 + (keyStr[i + j]
            }
        }
        sum += part;
    }
    return sum % tableSize;
}
```

- The folding implementation processes keys as string representation to handle arbitrary lengths.

Comparative Analysis of Hash Methods

Overview

Both methods aim to minimize collisions but are suited for different situations.

- Mid-Square has excellent performance when keys are numerical and digits are somewhat correlated, as it utilizes all digits of the key.
- Folding is ideal for very long keys (e.g., 16-digit credit card numbers) where arithmetic squaring might cause overflow.
- Both methods run in $O(1)$ time complexity for fixed-length keys, but folding takes $O(L)$ where L is key length.
- In practice, these methods are combined with collision resolution strategies like open addressing or chaining.