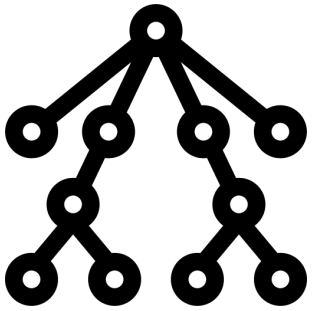


Data Structures Handbook and Commands Directory

Understanding Data Structures with C++

Girish Kumar Goyal

COMPUTER SCIENCE AND ENGINEERING

"Learn, Code, Optimize."

Preface

The purpose of this book is to give you a thorough introduction to **data structures**. It is assumed that you already have a basic understanding of programming, but no previous background in data structures or algorithms is needed.

This book is especially intended for *students* who want to learn how to efficiently organize and process data, and to build a strong foundation in algorithmic problem solving. However, it is equally suitable for professionals looking to strengthen their technical fundamentals or prepare for coding interviews.

Each chapter is designed with conceptual clarity, illustrative C++ code. It includes fundamental structures such as arrays and linked lists, dive into advanced trees and graphs, and discuss both classic and modern algorithms with a focus on time and space complexity.

I believe that theory must meet practice. Therefore, this book contains visual diagrams, sample programs.

I would like to express my gratitude to my mentors, peers, and the open-source community that continues to shape and evolve computer science education. Their contributions, directly or indirectly, have inspired many parts of this book.

This book is more than a technical reference — it is a roadmap for anyone who wishes to unlock the power of structured thinking and logical problem solving.

To the readers — approach each chapter with curiosity, code with confidence, and always keep learning.

India, April 2025
Girish Kumar Goyal

Contents

List of Figures

List of Tables

Part I

Basic Introduction

Chapter 1

Data Structure Introduction

What is Data Structure?

Data structures are ways of **organizing and storing data** so they can be used efficiently. As the name suggests, they involve organizing data in memory.

A data structure is **not a programming language** like C, C++, Java, or Python. Instead, it refers to a **set of algorithms and methods** that can be implemented in any programming language to manage data effectively in memory.

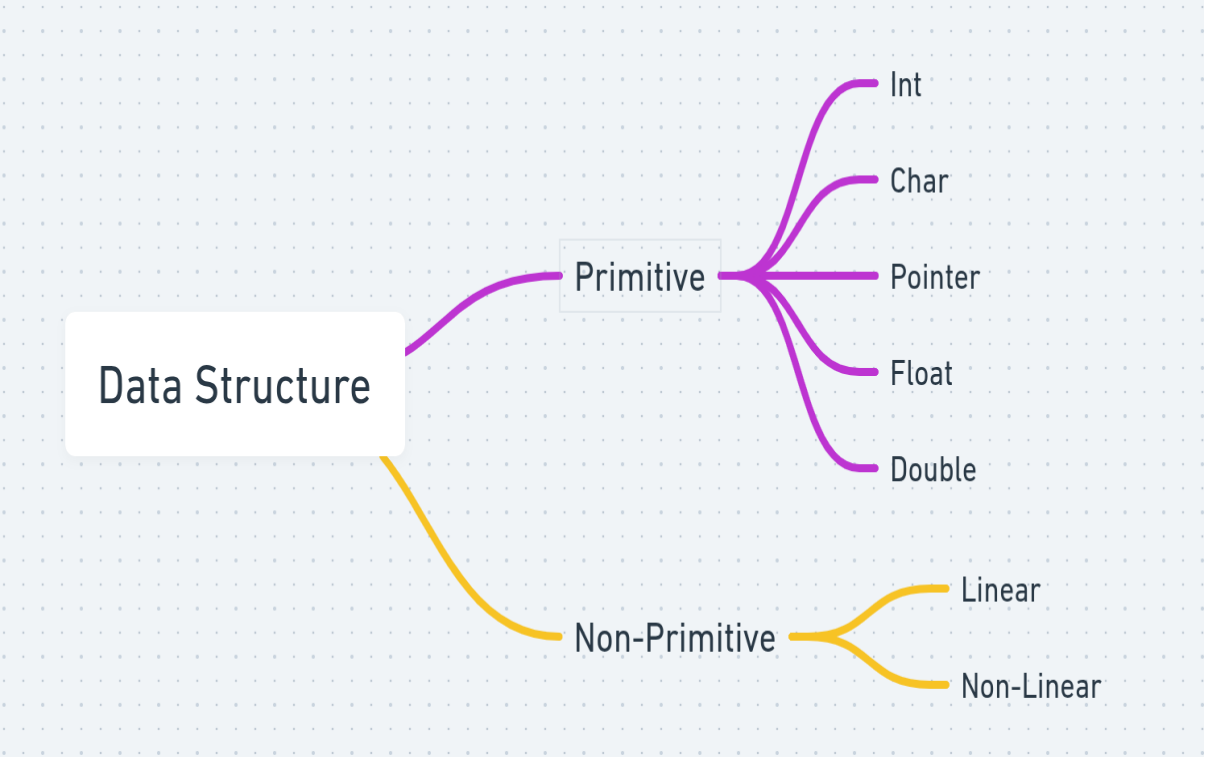


Figure 1.1: Types of Data Structure

Linear Data Structure

Linear data structures organize elements sequentially, where each element is connected to the next in a **single level** order. This structure simplifies tasks such as **traversal, insertion, and deletion**, and includes:

- Arrays
- Linked Lists
- Stacks
- Queues

Non-Linear Data Structure

Non-linear data structures are data structures in which elements are not arranged sequentially or linearly. Instead, each element can be connected to one or more elements in a hierarchical or interconnected manner, forming structures like trees and graphs. These structures are ideal for representing relationships where data cannot be stored linearly, such as hierarchical relationships (e.g., file systems) or complex networks (e.g., social networks, maps). Non-linear data structures organize elements in a **hierarchical or interconnected manner**, not linearly. These are suitable for representing complex relationships, such as:

- **Trees:** Representing hierarchical data like file systems.
- **Graphs:** Representing networks like social media or maps.

Algorithms and Abstract Data Types

Why??..

An **Abstract Data Type (ADT)** is a **conceptual model** that defines **what operations can be performed** on data, but **not how they are implemented**.

In simpler terms:

- **What** the data structure does (its behavior)
- **Not how** it does it (its implementation)

Key Characteristics of ADTs:

1. **Abstraction:** Hides implementation details. Users interact with the **interface**, not internals.
2. **Encapsulation:** Bundles **data and related operations** together.

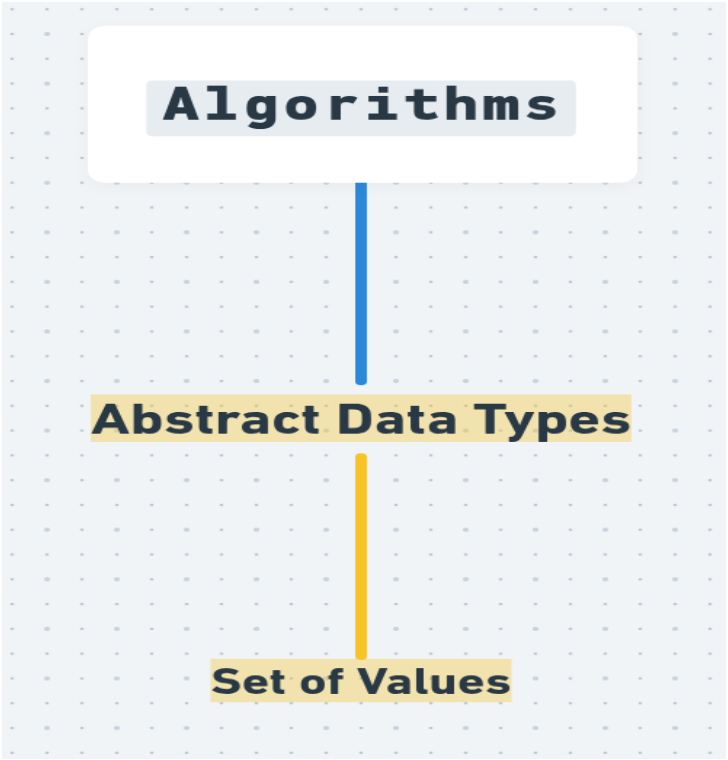


Figure 1.2: Algorithm

3. **Modularity:** Changes in implementation don't affect usage.
4. **Interface-Based Usage:** Operations like `insert`, `delete`, `search`, etc.

Examples of Common ADTs:

ADT	Common Operations	Implemented Using
List	insert, delete, traverse	Arrays, Linked Lists
Stack	push, pop, peek	Arrays, Linked Lists
Queue	enqueue, dequeue, front, rear	Arrays, Linked Lists
Deque	insertFront, insertRear, deleteFront, deleteRear	Arrays, Linked Lists

Table 1.1: Examples of Common Abstract Data Types

Basic Terminology

1. What is Data?

Data refers to raw facts and figures that have no meaning by themselves but can be processed to gain useful information. These could be **numbers, text, images, audio, etc**.

2. What is Record?

A **record** is a collection of **related data items or fields** that describe one entity or object. It often combines multiple **attributes**.

3. What is File?

A **file** is a collection of related records stored together. It is a **unit of data storage** and often represents a complete dataset.

4. What is an Attribute?

An **attribute** is a **characteristic or property** of an entity, describing one aspect of it.

5. What is Entity?

An **entity** is a real-world **object or concept** with data stored about it. It's typically represented as a **record**.

Need for Data Structures

Efficient data handling is **crucial** in computer science. As software grows in complexity, **organization and management of data** becomes essential. That's where **data structures** help.

1. Efficient Data Access and Processing

They enable **fast access, retrieval, and updates** — e.g., `arrays` allow direct indexing, `hash tables` provide constant-time lookup.

2. Code Optimization

Choosing the right structure improves **time and space complexity**, leading to more **efficient code**.

3. Data Organization

Logical arrangement (e.g., `trees`, `graphs`) simplifies representation of **relationships and hierarchy**.

4. Reusability and Modularity

Abstract data structures allow for **modular design**, enhancing reuse across applications.

5. Real-World Problem Solving

Applications like **search engines, databases, and social networks** rely on well-structured data.

6. Memory Management

Structures like `linked lists` or `trees` support **dynamic memory allocation** and efficient usage.

7. Algorithm Design

Most **algorithms** are designed around a **specific data structure** — making DSA foundational.

8. Scalability

Scalable structures ensure performance doesn't degrade with increased data.

Advantages of Data Structures

- **Improved Performance:** Optimized code runs faster.
- **Efficient Memory Usage:** Reduces memory wastage.
- **Reusability:** Use across multiple programs.
- **Maintainability:** Easier debugging and management.
- **Enhanced Productivity:** Simplifies complex logic.
- **Foundation of Algorithm Development:** Core of problem-solving.
- **Support for Complex Computations:** Trees, graphs enable solving advanced problems.

Chapter 2

Classification of Data Structures

Data structures can be broadly classified based on how data is organized and accessed. The classification helps in choosing the appropriate data structure for solving different kinds of problems efficiently.

1. Primitive and Non-Primitive Data Structures

Primitive Data Structures

These are the most basic data structures and serve as the building blocks for more complex data handling. They are directly operated upon by machine instructions.

- **Integer:** Stores whole numbers.
- **Float:** Stores decimal numbers.
- **Character:** Stores single characters.
- **Boolean:** Stores true or false values.

Example:

```
int age = 25;
float temperature = 36.6;
char grade = 'A';
bool isAvailable = true;
```

Non-Primitive Data Structures

These are more complex structures built using primitive data types. They are further divided into:

- **Linear Data Structures**
- **Non-Linear Data Structures**

---

2. Linear Data Structures

Linear data structures arrange data in a sequential manner. Each element is connected to its previous and next element, forming a linear order.

1. Array

An array is a fixed-size collection of elements of the same type stored in contiguous memory locations.

Example:

```
int arr[5] = {10, 20, 30, 40, 50};
for (int i = 0; i < 5; i++) {
    cout << arr[i] << " ";
}
```

2. Linked List

A linked list is a linear structure where each element (node) points to the next, allowing dynamic memory allocation.

Example:

```
struct Node {
    int data;
    Node* next;
};
Node* head = new Node{10, nullptr};
head->next = new Node{20, nullptr};
```

3. Stack

A stack is a collection of elements that follows the Last In, First Out (LIFO) principle.

Example:

```
stack<int> s;
s.push(10);
s.push(20);
s.pop();
cout << s.top(); // Output: 10
```

4. Queue

A queue follows the First In, First Out (FIFO) principle.

Example:

```
queue<int> q;
q.push(10);
q.push(20);
q.pop();
```

```
cout << q.front(); // Output: 20
```

5. Deque (Double Ended Queue)

Deque allows insertion and deletion from both ends (front and rear).

Example:

```
deque<int> dq;
dq.push_front(10);
dq.push_back(20);
cout << dq.front(); // Output: 10
cout << dq.back(); // Output: 20
```

---

3. Non-Linear Data Structures

Non-linear structures store data hierarchically or in a network. Elements are not arranged in sequence.

1. Tree

A tree is a hierarchical structure with a root node and child nodes. Common trees include binary trees, binary search trees, AVL trees, etc.

Example:

```
struct TreeNode {
    int val;
    TreeNode* left;
    TreeNode* right;
};
TreeNode* root = new TreeNode{10, nullptr, nullptr};
```

2. Graph

A graph consists of a set of nodes (vertices) and edges connecting pairs of nodes. Graphs may be directed or undirected, weighted or unweighted.

Example using adjacency list:

```
vector<int> graph[5];
graph[0].push_back(1);
graph[1].push_back(2);
```

---

4. Static and Dynamic Data Structures

Static Data Structures

These structures have fixed size and memory is allocated at compile time (e.g., arrays).

Example:

```
int nums[100]; // Static allocation
```

Dynamic Data Structures

Size can change during runtime and memory is allocated dynamically (e.g., linked lists, trees).

Example:

```
int* ptr = new int[100];
delete[] ptr;
```

---

5. Homogeneous and Non-Homogeneous Data Structures

Homogeneous

All elements are of the same data type (e.g., arrays).

Non-Homogeneous

Elements can be of different data types (e.g., structures in C/C++).

Example:

```
struct Student {
    int id;
    string name;
    float marks;
};
```

Summary Table

Category	Type	Examples
Primitive	-	int, float, char, boolean
Non-Primitive	Linear	Array, Linked List, Stack, Queue
Non-Primitive	Non-Linear	Tree, Graph
Based on Memory	Static	Array
Based on Memory	Dynamic	Linked List, Tree
Based on Type	Homogeneous	Array
Based on Type	Non-Homogeneous	Structure

Table 2.1: Classification of Data Structures

Primitive Data Type Ranges

- **Integer (int):**
  - Minimum Value: **-2,147,483,648** (i.e., -2^{31})
  - Maximum Value: **2,147,483,647** (i.e., $2^{31} - 1$)
  - Size: **4 bytes** (32 bits)
- **Floating Point (float):**
  - Minimum Positive Value: **1.4×10^{-45}** (approx)
  - Maximum Value: **3.4×10^{38}** (approx)
  - Precision: **6 to 7 digits**
  - Size: **4 bytes** (IEEE 754 standard)
- **Character (char):**
  - Minimum Value: **0** (Null character, `'\0'`)
  - Maximum Value: **127** (Standard ASCII) or **255** (Extended ASCII)
  - Size: **1 byte** (8 bits)
- **Boolean:**
  - Possible Values: **true**, **false**
  - Size: **1 bit** (may occupy 1 byte in memory)

Operations on Data Structures

Data structures support several essential operations that allow us to access, modify, and manipulate stored data efficiently. Below are the fundamental operations:

1. Traversing

Traversing refers to the process of visiting each element in the data structure exactly once to perform some operation (e.g., displaying or processing data).

- In linear structures (like arrays or linked lists), traversal is typically done from the first to the last element.
- In tree structures, traversal can be in-order, pre-order, or post-order.
- In graphs, traversal can be depth-first (DFS) or breadth-first (BFS).

Example:

```
// Traversing an array
for (int i = 0; i < n; i++) {
    cout << arr[i] << " ";
}
```

2. Insertion

Insertion involves adding a new element to the data structure at a specific location.

- In arrays, inserting in the middle requires shifting elements.
- In linked lists, it involves changing node pointers.
- In trees or heaps, it must maintain structural properties.

Time Complexity:

- Array (unsorted): $O(n)$
- Linked List (at head): $O(1)$

3. Deletion

Deletion removes a specified element from a data structure.

- In arrays, this involves shifting elements to fill the gap.
- In linked lists, pointers are updated to unlink the node.
- In binary search trees, deletion can be complex if the node has children.

Time Complexity:

- Array (unsorted): $O(n)$
- Linked List (if pointer is given): $O(1)$

4. Searching

Searching locates the position of a given element in the data structure.

- **Linear Search:** Sequentially checks every element. Time Complexity: $O(n)$
- **Binary Search:** Used in sorted arrays. Time Complexity: $O(\log n)$

Example:

```
// Linear Search
for (int i = 0; i < n; i++) {
    if (arr[i] == key) return i;
}
```

5. Sorting

Sorting is the process of arranging data in a particular order (ascending or descending). It improves the efficiency of other operations like searching and merging.

Common Algorithms:

- Bubble Sort — $O(n^2)$
- Insertion Sort — $O(n^2)$
- Merge Sort — $O(n \log n)$
- Quick Sort — Average: $O(n \log n)$, Worst: $O(n^2)$

6. Merging

Merging is the process of combining two sorted data structures into one sorted structure.

- Often used in algorithms like Merge Sort.
- Efficient merging requires both input structures to be sorted.

Example:

```
// Merging two sorted arrays
while (i < n && j < m) {
    if (A[i] < B[j]) C[k++] = A[i++];
    else C[k++] = B[j++];
}
```

—

These operations are fundamental to algorithm development and problem-solving in computer science and form the core of many real-world applications.

Chapter 3

Introduction to Algorithm

Algorithms

What is an Algorithm?

An **algorithm** is a finite sequence of well-defined instructions designed to perform a specific task or solve a particular problem. It serves as a step-by-step guide for solving computational problems and can be implemented in any programming language.

Algorithms are fundamental to computer science and are used in data processing, calculations, artificial intelligence, automation, and numerous other fields. A good algorithm is not only correct but also efficient in terms of time and space.

Characteristics of an Algorithm

An algorithm must satisfy the following essential properties:

1. Input

An algorithm should have zero or more inputs. These are the values or data provided to the algorithm before it starts execution.

Example: An algorithm to calculate the sum of two numbers requires two input values.

2. Output

An algorithm must produce at least one output. The output is the result obtained after executing the algorithm.

Example: The result of the sum operation in the above algorithm.

3. Unambiguity

Every step or instruction in an algorithm should be clear and unambiguous. There should be no room for multiple interpretations.

Example: Instead of saying "Sort the numbers," say "Use Bubble Sort to sort the numbers in ascending order."

4. Finiteness

An algorithm must always terminate after a finite number of steps. It should not enter into an infinite loop.

Example: A loop running from 1 to 10 is finite, but a loop with no exit condition can be infinite.

5. Effectiveness

Each operation in an algorithm must be sufficiently basic and capable of being performed exactly and in a finite amount of time by a person or machine.

Example: Simple arithmetic operations like addition, multiplication, etc., are effective steps.

Approaches in Algorithm

There are various approaches to designing and solving problems using algorithms. Each approach offers a unique way of breaking down the problem:

- **Brute Force:** Tries all possible solutions until the correct one is found. Simple but inefficient for large problems.
- **Divide and Conquer:** Breaks the problem into smaller sub-problems, solves them independently, and combines their results. Example: Merge Sort.
- **Greedy Approach:** Builds up a solution piece by piece, always choosing the option that seems best at the moment. Example: Dijkstra's Algorithm.
- **Dynamic Programming:** Solves problems by combining the solutions of overlapping sub-problems. Example: Fibonacci with memoization.
- **Backtracking:** Tries all possibilities by exploring every path recursively and backtracking when needed. Example: N-Queens Problem.
- **Randomized Algorithms:** Uses randomness as part of its logic. Example: Randomized QuickSort.

Algorithm Analysis

Algorithm analysis is a critical step in the development of efficient software. It helps in evaluating an algorithm's efficiency in terms of the computational resources it consumes—primarily time and memory. The two primary measures for algorithm analysis are:

- **Time Complexity:** How long an algorithm takes to run.
- **Space Complexity:** How much memory an algorithm uses during execution.

A Priori Analysis

A Priori Analysis refers to analyzing the algorithm theoretically before implementing it. It involves estimating time and space complexity by examining the structure of the algorithm and using mathematical formulas.

Example: For a sorting algorithm with nested loops, we may conclude its time complexity is $O(n^2)$ based on loop counts.

A Posteriori Analysis

A Posteriori Analysis is done after implementing the algorithm. It involves running the code with different inputs and measuring actual execution time and memory consumption using tools or profilers.

Example: Measuring runtime of a sorting function in seconds using a stopwatch or system profiler.

Time Complexity

Time Complexity refers to the computational time taken by an algorithm to run as a function of the size of the input.

- Expressed using Big-O notation (e.g., $O(1)$, $O(n)$, $O(n^2)$).
- Helps in comparing algorithms independent of hardware.
- Affects scalability — i.e., how well an algorithm performs as input grows.

Space Complexity

Space Complexity is the amount of memory space required by an algorithm during its execution, including:

- Input storage
- Auxiliary space (temporary variables, stack, etc.)

Efficient algorithms minimize both time and space usage, though often there is a trade-off between them.

Common Algorithms and Their Complexities

Algorithm	**Time Complexity**	**Space Complexity**	**Type**
Linear Search	$O(n)$	$O(1)$	Searching
Binary Search	$O(\log n)$	$O(1)$	Searching (Sorted Array)
Bubble Sort	$O(n^2)$	$O(1)$	Sorting
Merge Sort	$O(n \log n)$	$O(n)$	Sorting (Divide and Conquer)
Quick Sort	$O(n \log n)$ (avg), $O(n^2)$ (worst)	$O(\log n)$	Sorting (Divide and Conquer)
Insertion Sort	$O(n^2)$	$O(1)$	Sorting
DFS (Graph)	$O(V+E)$	$O(V)$	Graph Traversal
BFS (Graph)	$O(V+E)$	$O(V)$	Graph Traversal
Dijkstra's Algorithm	$O((V+E)\log V)$	$O(V)$	Shortest Path
Fibonacci (Recursion)	$O(2^n)$	$O(n)$	Recursion
Fibonacci (DP)	$O(n)$	$O(n)$ or $O(1)$ (optimized)	Dynamic Programming

Note:

- V = number of vertices, E = number of edges in a graph.
- Choosing the right algorithm is essential to building efficient software systems.

Implementation of Basic Algorithms

1. Linear Search

```
int linearSearch(vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target)
            return i;
    }
    return -1;
}
```

Listing 3.1: Linear Search

2. Binary Search

```
int binarySearch(vector<int>& arr, int target) {
    int left = 0, right = arr.size() - 1;
    while (left <= right) {
        int mid = left + (right - left) / 2;
        if (arr[mid] == target) return mid;
        else if (arr[mid] < target) left = mid + 1;
        else right = mid - 1;
    }
    return -1;
}
```

Listing 3.2: Binary Search (Iterative)

3. Bubble Sort

```
void bubbleSort(vector<int>& arr) {
    for (int i = 0; i < arr.size() - 1; i++) {
        for (int j = 0; j < arr.size() - i - 1; j++) {
            if (arr[j] > arr[j + 1])
                swap(arr[j], arr[j + 1]);
        }
    }
}
```

Listing 3.3: Bubble Sort

4. Merge Sort

```
void merge(vector<int> &arr, int low, int mid, int high){
    vector<int> temp;
    int left = low;
```

```
    int right = mid + 1;
    while(left <= mid && right <= high){
        if(arr[left] <= arr[right]){
            temp.push_back(arr[left]);
            left++;
        }
        else{
            temp.push_back(arr[right]);
            right++;
        }
    }
    while(left <= mid){
        temp.push_back(arr[left]);
        left++;
    }
    while(right <= high){
        temp.push_back(arr[right]);
        right++;
    }
    for(int i = low; i <= high; ++i){
        arr[i] = temp[i - low];
    }
}

void merge_sort(vector<int> &arr, int low, int high){
    if(low >= high){
        return;
    }
    int mid = (low + high) / 2;
    merge_sort(arr, low, mid);
    merge_sort(arr, mid + 1, high);
    merge(arr, low, mid, high);
}
```

Listing 3.4: Merge Sort

5. Quick Sort

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

```
void quickSort(vector<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);
    }
}
```

Listing 3.5: Quick Sort

Chapter 4

Asymptotic Analysis

Asymptotic Analysis

Asymptotic analysis is a method used to describe the behavior of functions as the input size n grows large. It provides a framework to compare algorithms by focusing on their dominant terms while ignoring constant factors and lower-order terms.

1. Case Analysis

- **Worst Case:** The maximum number of operations an algorithm performs on any input of size n. This gives an upper bound.
- **Average Case:** The expected number of operations, averaged over all inputs of size n (assuming some probability distribution over inputs).
- **Best Case:** The minimum number of operations required for some input of size n. Though fast, it is less useful for guarantees.

2. Asymptotic Notations

- **O (Big-O):** $f(n) = O(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that

$$0 \leq f(n) \leq c \cdot g(n) \quad \text{for all } n \geq n_0.$$

  This is an upper bound on the growth rate of $f(n)$.

- ■ **(Theta):** $f(n) = \Theta(g(n))$ if there exist constants $c_1, c_2 > 0$ and $n_0 > 0$ such that

$$0 \leq c_1 \cdot g(n) \leq f(n) \leq c_2 \cdot g(n) \quad \text{for all } n \geq n_0.$$

  This provides a tight bound.

- ■ **(Big-Omega):** $f(n) = \Omega(g(n))$ if there exist constants $c > 0$ and $n_0 > 0$ such that

$$0 \leq c \cdot g(n) \leq f(n) \quad \text{for all } n \geq n_0.$$

  This is a lower bound on the growth rate.

3. Graphical Representation

Below is a TikZ graph comparing the asymptotic bounds for the quadratic function $f(n) = n^2$. In this graph:

- The red dashed curve represents an upper bound $O(n^2)$ (e.g., $1.2n^2$).
- The blue solid curve represents a tight bound $\Theta(n^2)$ (i.e., n^2).
- The green dotted curve represents a lower bound $\Omega(n^2)$ (e.g., $0.8n^2$).

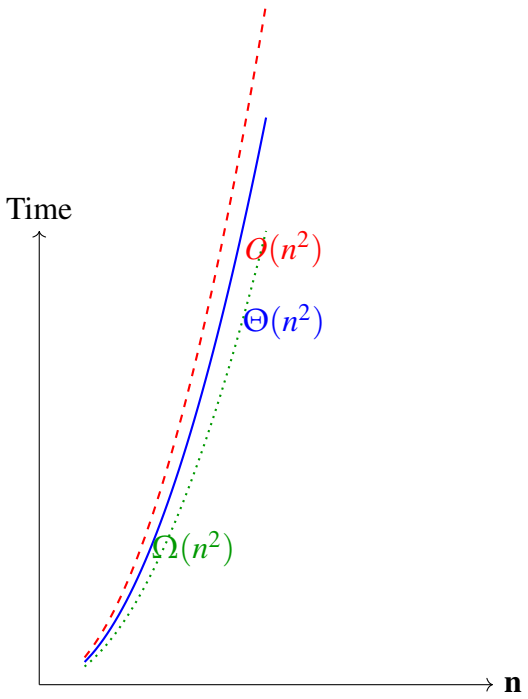


Figure 4.1: Graphical illustration of $O(n^2)$, $\Theta(n^2)$, and $\Omega(n^2)$.

4. Example: Linear Search

Consider the Linear Search algorithm in an unsorted array of size n.

- **Best Case:** The key is found at the first index: $\Theta(1)$.
- **Worst Case:** The key is found at the last index or not present: $O(n)$.
- **Average Case:** On average, about $\frac{n}{2}$ comparisons are made: $\Theta(n)$.

5. Additional C++ Examples

Below are two C++ examples that further illustrate algorithm behavior and their asymptotic analysis.

Example 1: Binary Search

Binary Search on a sorted array runs in $O(\log n)$ time. The following code snippet demonstrates the algorithm:

```
int binarySearch(int arr[], int n, int key) {
    int low = 0, high = n - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == key)
            return mid; // Key found, best case: O(1)
        else if (arr[mid] < key)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Key not found, worst case: O(log n)
}
```

Listing 4.1: Binary Search in C++

Time Complexity Analysis: Binary Search operates by repeatedly dividing the search interval in half. Let's analyze this behavior more formally.

Let the size of the array be n. At each step:

- We compare the key with the middle element.
- Based on the comparison, we discard half of the elements.

Thus, the size of the array becomes:

$$n, \frac{n}{2}, \frac{n}{4}, \ldots, \frac{n}{2^k}$$

We stop when the sub-array has only one element:

$$\frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log_2 n$$

Therefore, the maximum number of steps required is $\log_2 n$.

$$\Rightarrow T(n) = O(\log n)$$

Best Case:

- When the key is found at the middle on the first comparison.
- **Time Complexity:** $\Theta(1)$

Worst Case:

- When the key is not present, or found after completely narrowing the search interval.
- **Time Complexity:** $O(\log n)$

Average Case:

- On average, it also takes approximately $\log n$ steps to find the key.
- **Time Complexity:** $\Theta(\log n)$

Conclusion: Binary Search is extremely efficient for large sorted datasets, with a logarithmic time complexity.

Best Case:	$\Theta(1)$
Average Case:	$\Theta(\log n)$
Worst Case:	$O(\log n)$

Example 2: Quick Sort

Quick Sort has an average-case time complexity of $O(n \log n)$ and a worst-case complexity of $O(n^2)$. Here is a simplified version in C++:

```
void swap(int &a, int &b) {
    int temp = a;
    a = b;
    b = temp;
}
int partition(int arr[], int low, int high) {
    int pivot = arr[high]; // Choosing the last element as pivot
    int i = low - 1;
    for (int j = low; j < high; j++) {
        if (arr[j] < pivot) {
            i++;
            swap(arr[i], arr[j]);
        }
    }
    swap(arr[i+1], arr[high]);
    return i + 1;
}
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);
    }
}
```

Listing 4.2: Quick Sort in C++

Time Complexity Analysis: Quick Sort is a **divide-and-conquer** algorithm. It works by:

- Selecting a **pivot** element.

- Partitioning the array so that all elements less than the pivot are on the left, and those greater are on the right.
- Recursively applying the same process to the left and right subarrays.

Let $T(n)$ be the time complexity to sort an array of size n.

Best and Average Case: If the pivot divides the array into two equal parts (or close to equal), then:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

- $O(n)$ is the cost of partitioning.
- Solving this recurrence gives:

$$T(n) = O(n \log n)$$

Worst Case: If the pivot is the smallest or largest element (highly unbalanced partition), then:

$$T(n) = T(n-1) + O(n)$$

- This leads to the recurrence:

$$T(n) = T(n-1) + n \Rightarrow T(n) = O(n^2)$$

- Occurs when the array is already sorted (ascending or descending) and pivot selection is poor.

Average Case (More Detailed): Average time complexity considers all possible partitioning scenarios and averages them. On average, each partition divides the array into two parts of size i and $n-i-1$. The recurrence is:

$$T(n) = \frac{1}{n} \sum_{i=0}^{n-1} (T(i) + T(n-i-1)) + cn$$

Solving this results in:

$$T(n) = \Theta(n \log n)$$

Conclusion:

Best Case:	$\Theta(n \log n)$
Average Case:	$\Theta(n \log n)$
Worst Case:	$O(n^2)$

Quick Sort is efficient in practice due to low constant factors and good cache performance, especially with randomized or median-of-three pivot strategies.

Space Complexity Analysis: Quick Sort is an **in-place** sorting algorithm, meaning it requires only a small, constant amount of extra space for partitioning.

- In the best and average case, the depth of the recursion tree is $\log n$, leading to:

$$\boxed{\text{Space Complexity (Auxiliary Stack): } O(\log n)}$$

- In the worst case (unbalanced partitions), the recursion depth becomes n:

$$\boxed{\text{Worst Case Space Complexity: } O(n)}$$

Recursion Tree Visualization: Below is a simplified recursion tree for Quick Sort in the best case, where the pivot splits the array into two equal halves each time.

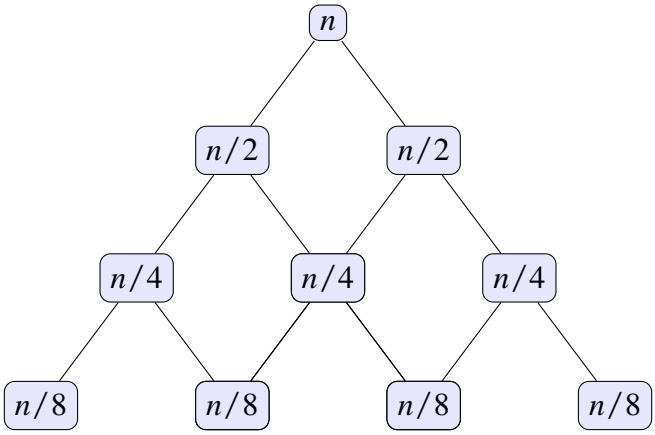


Figure 4.2: Recursion Tree of Quick Sort in Best Case (Perfectly Balanced)

Observation:

- The recursion depth is $\log n$ (base 2).
- Each level of the tree performs total work proportional to n.
- Total time across all levels:

$$n + n + n + \cdots + n = n \log n$$

- Hence, the overall time complexity in the best/average case is:

$$\boxed{T(n) = O(n \log n)}$$

6. Insertion Sort

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

```
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}
```

Listing 4.3: Insertion Sort

6. Insertion Sort (Analysis)

Time Complexity Analysis: Insertion Sort builds the sorted array one element at a time. For each element, it is compared with all previous elements and shifted accordingly.

- **Best Case:** The array is already sorted. Each element requires only one comparison:

$$T(n) = \sum_{i=1}^{n-1} 1 = n-1 \Rightarrow \boxed{O(n)}$$

- **Worst Case:** The array is reverse sorted. Every new element is compared with all previous elements and shifted:

$$T(n) = \sum_{i=1}^{n-1} i = \frac{(n-1)n}{2} \Rightarrow \boxed{O(n^2)}$$

- **Average Case:** On average, each element is compared with half of the sorted part:

$$T(n) = \sum_{i=1}^{n-1} \frac{i}{2} = \frac{1}{2} \cdot \frac{(n-1)n}{2} = \frac{n(n-1)}{4} \Rightarrow \boxed{O(n^2)}$$

Space Complexity: Insertion Sort is an in-place sorting algorithm.

$$\boxed{\text{Space Complexity: } O(1)}$$

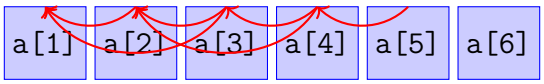


Figure 4.3: Comparison pattern in Insertion Sort (Worst Case)

Comparison Pattern (TikZ Visualization):

Summary:

- **Best Case:** $\boxed{O(n)}$
- **Average Case:** $\boxed{O(n^2)}$
- **Worst Case:** $\boxed{O(n^2)}$
- **Space Complexity:** $\boxed{O(1)}$

7. Selection Sort

```
void selectionSort(vector<int>& arr) {
    for (int i = 0; i < arr.size() - 1; i++) {
        int minIdx = i;
        for (int j = i + 1; j < arr.size(); j++) {
            if (arr[j] < arr[minIdx]) minIdx = j;
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

Listing 4.4: Selection Sort

7. Selection Sort (Analysis)

Time Complexity Analysis: Selection Sort works by repeatedly finding the minimum element from the unsorted part and moving it to the sorted part. It always performs the same number of comparisons regardless of the initial order of the array.

- **Best Case:** Array is already sorted. Still needs to compare all elements to find the minimum.

$$T(n) = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

- **Worst Case:** Array is in reverse order. Comparisons remain the same.

$$T(n) = \sum_{i=0}^{n-2} (n-i-1) = \frac{n(n-1)}{2} \Rightarrow \boxed{O(n^2)}$$

- **Average Case:** Same number of comparisons as worst and best cases.

$$\boxed{O(n^2)}$$

Swap Count: Unlike Insertion Sort, Selection Sort performs fewer swaps:

$$\text{Maximum Swaps: } n-1 \Rightarrow \boxed{O(n)}$$

Space Complexity: Selection Sort is an in-place sorting algorithm and does not require extra space:

$$\boxed{\text{Space Complexity: } O(1)}$$

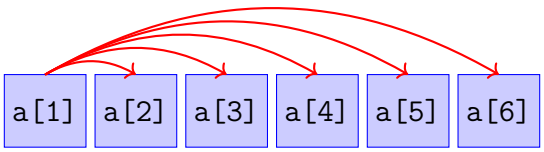


Figure 4.4: Selection of the minimum element in each iteration

Comparison Pattern (TikZ Visualization):

Summary:

- **Best Case:** $\boxed{O(n^2)}$
- **Average Case:** $\boxed{O(n^2)}$
- **Worst Case:** $\boxed{O(n^2)}$
- **Swap Complexity:** $\boxed{O(n)}$
- **Space Complexity:** $\boxed{O(1)}$

8. Heap Sort

```
void heapify(vector<int>& arr, int n, int i) {
    int largest = i;
    int l = 2 * i + 1;
    int r = 2 * i + 2;
    if (l < n && arr[l] > arr[largest]){
        largest = l;
    }
    if (r < n && arr[r] > arr[largest]){
        largest = r;
    }
    if (largest != i) {
        swap(arr[i], arr[largest]);
        heapify(arr, n, largest);
    }
}
void heapSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = n / 2 - 1; i >= 0; i--){
        heapify(arr, n, i);
    }
    for(int i = n - 1; i > 0; i--){
        swap(arr[0], arr[i]);
        heapify(arr, i, 0);
    }
}
```

Listing 4.5: Heap Sort

Time Complexity Analysis (Max-Heap):

- **Heapify operation:** takes $O(\log n)$.
- **Building heap:** $O(n)$ (by applying heapify bottom-up).

- **Extracting max (n times):** $n \cdot O(\log n) = O(n \log n)$.

Overall Time Complexity (Max-Heap): $O(n \log n)$

Space Complexity: Heap Sort is in-place:

$O(1)$

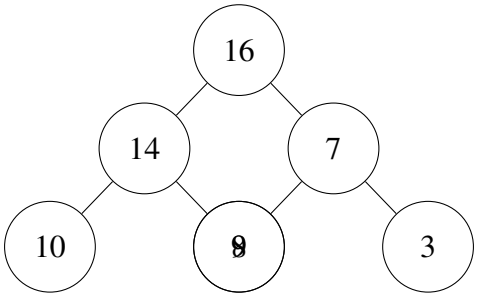


Figure 4.5: Max-Heap Tree Structure

Tree Diagram (Max-Heap Representation):

Array Representation (Max-Heap):

```
[16, 14, 7, 10, 8, 9, 3]
```

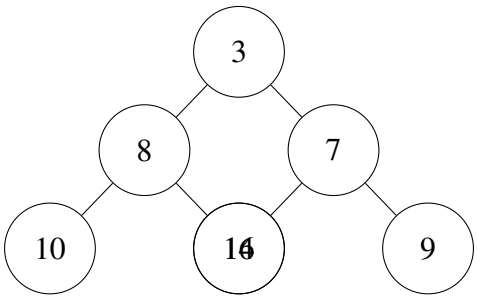


Figure 4.6: Min-Heap Tree Structure

Tree Diagram (Min-Heap Representation):

Array Representation (Min-Heap):

```
[3, 8, 7, 10, 14, 16, 9]
```

Time Complexity Analysis (Min-Heap): For a min-heap, if we sort in descending order, the time complexity is the same:

- **Heapify:** $O(\log n)$
- **Build Heap:** $O(n)$
- **n Deletions:** $O(n \log n)$

$$\boxed{\text{Overall Time Complexity (Min-Heap): } O(n \log n)}$$

Summary:

- **Best Case:** $O(n \log n)$
- **Average Case:** $O(n \log n)$
- **Worst Case:** $O(n \log n)$
- **Space Complexity:** $O(1)$
- **Stable:** No

9. Counting Sort

```
void countingSort(vector<int>& arr) {
    if (arr.empty()){
        return;
    }
    int maxVal = *max_element(arr.begin(), arr.end());
    vector<int> count(maxVal + 1, 0);
    for (int num : arr){
        count[num]++;
    }
    int idx = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i]-- > 0) arr[idx++] = i;
    }
}
```

Listing 4.6: Counting Sort (Non-negative Integers)

Time Complexity Analysis: Counting Sort assumes all input elements are non-negative integers and works by counting occurrences.

Let n be the number of elements and k be the range of input values.

- **Counting frequency:** Takes $O(n)$.
- **Populating sorted array:** Takes up to $O(k)$ iterations.

$$\boxed{\text{Time Complexity: } O(n+k)}$$

Space Complexity: Extra space for the count array of size $k+1$:

$$\boxed{O(k)}$$

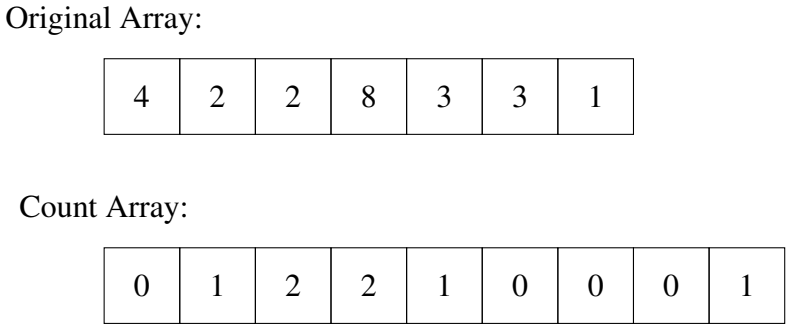


Figure 4.7: Counting Sort Diagram: Frequency Count

Diagram Representation:

Summary:

- **Best Case:** $O(n+k)$
- **Average Case:** $O(n+k)$
- **Worst Case:** $O(n+k)$
- **Space Complexity:** $O(k)$
- **Stable:** Yes

10. Radix Sort

```
int getMax(vector<int>& arr) {
    return *max_element(arr.begin(), arr.end());
}
void countingSort(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<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] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
```

```
    for (int i = 0; i < n; i++) {
        arr[i] = output[i];
    }
}
void radixSort(vector<int>& arr) {
    if (arr.size() <= 1) return;
    int max = getMax(arr);
    for(int exp = 1; max / exp > 0; exp *= 10){
        countingSort(arr, exp);
    }
}
```

Listing 4.7: Radix Sort (LSD, Non-negative Integers)

Time Complexity Analysis: Radix Sort processes each digit individually using Counting Sort as a stable subroutine.

Let:

- n be the number of elements,
- k be the maximum value in the input,
- d be the number of digits in the maximum number (i.e., $d = \log_b k$, where b is the base).

Each counting sort pass takes $O(n+b)$ time and we do it for d digits:

$$\boxed{\text{Time Complexity: } O(d(n+b)) = O(n \log k)}$$

For decimal representation ($b = 10$):

$$\boxed{O(n \cdot \log_{10} k)}$$

Space Complexity: Uses temporary arrays of size n and b:

$$\boxed{O(n+b)}$$

Original Array: `[170, 45, 75, 90, 802, 24, 2, 66]`

Pass 1 (Units Digit):
`[170, 90, 802, 2, 24, 45, 75, 66]`

Pass 2 (Tens Digit):
`[802, 2, 24, 45, 66, 170, 75, 90]`

Pass 3 (Hundreds Digit):
`[2, 24, 45, 66, 75, 90, 170, 802]`

Figure 4.8: Radix Sort Progression by Digit Position

Diagram Representation:

Summary:

- **Best Case:** $O(n \cdot \log_b k)$
- **Average Case:** $O(n \cdot \log_b k)$
- **Worst Case:** $O(n \cdot \log_b k)$
- **Space Complexity:** $O(n+b)$
- **Stable:** Yes

11. Bucket Sort

```
void bucketSort(vector<float>& arr) {
    int n = arr.size();
    vector<vector<float>> buckets(n);
    for (int i = 0; i < n; i++) {
        int idx = n * arr[i];
        buckets[idx].push_back(arr[i]);
    }
    for (int i = 0; i < n; i++){
        sort(buckets[i].begin(), buckets[i].end());
    }
    int idx = 0;
    for (int i = 0; i < n; i++) {
        for (float val : buckets[i]) {
            arr[idx++] = val;
        }
    }
}
```

Listing 4.8: Bucket Sort (Float numbers between 0 and 1)

Time Complexity Analysis: Bucket Sort distributes elements into buckets and sorts them individually.

Let:

- n be the number of elements,
- k be the number of buckets (commonly $k = n$).

Assuming uniform distribution and insertion sort inside each bucket:

- **Distribute into buckets:** $O(n)$
- **Sort each bucket:** Expected $O(n)$ if elements are uniformly distributed

$$\boxed{\text{Expected Time Complexity: } O(n)}$$

Worst-case: When all elements fall into a single bucket, leading to:

$$\boxed{O(n^2)}$$

Space Complexity: Additional space for buckets:

$$\boxed{O(n)}$$

Buckets:

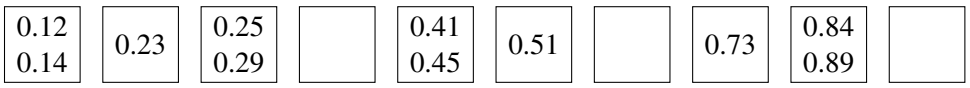


Figure 4.9: Bucket Sort Buckets with Sorted Values

Diagram Representation:

Summary:

- **Best Case:** $O(n)$
- **Average Case:** $O(n)$
- **Worst Case:** $O(n^2)$
- **Space Complexity:** $O(n)$
- **Stable:** Depends on internal sort (e.g., insertion sort = stable)

6. Highlighting Function Notations

We often express the dominating term of a function using color to emphasize its impact. For example, consider:

$$\mathbf{f(n) = n^2 + 3n + 2}$$

For large n, the term n^2 dominates, hence:

$$f(n) = \Theta(n^2)$$

This shows that lower-order terms and constants are ignored in asymptotic analysis.

Chapter 5

Pointers in C++

Pointers:

Pointers are variables that store the memory address of another variable. They are a powerful feature in C/C++ that allow direct memory access and manipulation. This chapter defines pointers, explains pointer-to-pointer, pointer arithmetic, and advanced pointer usage with examples, diagrams, and a comparison table.

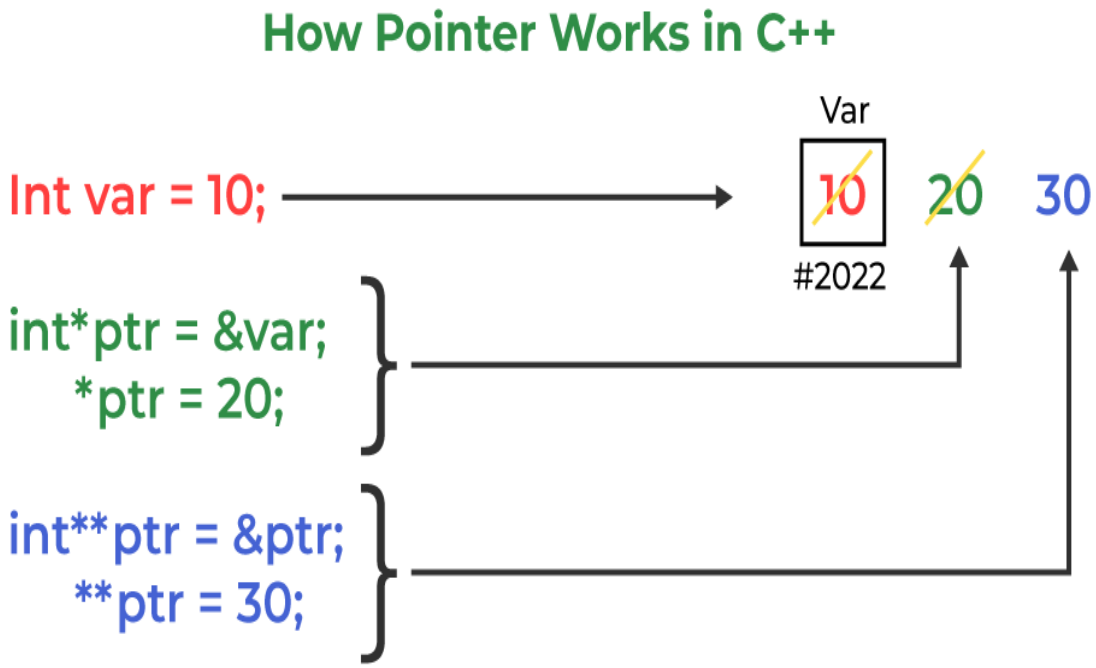


Figure 5.1: Pointer in C++

1. Definition and Explanation of Pointers

A **pointer** is a variable whose value is the address of another variable. For example, if we have an integer variable, its pointer will store the memory location where that integer is held.

Key Points:

- Declaring a pointer: use the asterisk (*) before the pointer variable name.
- Dereferencing: the operator * is used to access the value at the memory address stored in the pointer.
- Address-of operator: the operator & is used to get the memory address of a variable.

2. Basic Pointer Usage in C++

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 42;
    int *ptr = &a;
    cout << "Value of a: " << a << endl;
    cout << "Address of a: " << &a << endl;
    cout << "Value stored in ptr (address of a): " << ptr << endl;
    cout << "Dereferenced ptr (value of a): " << *ptr << endl;
    return 0;
}
```

Listing 5.1: Basic Pointer Example in C++

3. Pointer to Pointer

A **pointer to pointer** stores the address of another pointer.

```cpp
#include <iostream>
using namespace std;

int main() {
    int a = 100;
    int *ptr = &a;
    int **pptr = &ptr;
    cout << "Value of a: " << a << endl;
    cout << "Address of a: " << &a << endl;
    cout << "ptr points to: " << ptr << endl;
    cout << "*ptr: " << *ptr << endl;
    cout << "pptr points to: " << pptr << endl;
    cout << "*pptr: " << *pptr << endl;
    cout << "**pptr: " << **pptr << endl;
    return 0;
}
```

Listing 5.2: Pointer to Pointer in C++

4. Pointer Arithmetic and Arrays

Pointer arithmetic is useful in array navigation. Adding 1 to a pointer makes it point to the next memory location of the pointed data type.

```cpp
#include <iostream>
using namespace std;

int main() {
    int arr[5] = {10, 20, 30, 40, 50};
    int *ptr = arr;
    cout << "Array elements using pointer arithmetic:" << endl;
    for (int i = 0; i < 5; i++) {
        cout << "Element " << i << ": " << *(ptr + i) << endl;
    }
    return 0;
}
```

Listing 5.3: Pointer Arithmetic with Array in C++

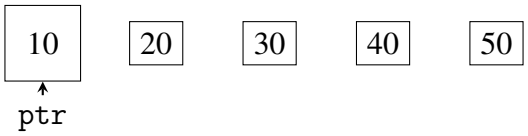


Figure 5.2: Pointer arithmetic: `ptr` points to `arr[0]`, `ptr+1` points to `arr[1]`, and so on.

5. Table: Pointer Types and Use Cases

Pointer Type	Declaration	Usage
Single Pointer	`int *p;`	Holds address of an int
Double Pointer	`int **pp;`	Holds address of pointer to int
Null Pointer	`int *p = NULL;`	Indicates no address assigned
Void Pointer	`void *p;`	Generic pointer (typecast before use)
Function Pointer	`int (*fptr)(int);`	Points to a function

Table 5.1: Common pointer types and their usage

Summary

- Pointers store memory addresses and enable dynamic memory access.
- They support arithmetic, point to arrays, and allow pointer-to-pointer usage.
- Advanced forms include void pointers, function pointers, and pointer arrays.

Chapter 6

Struct, Union and Enum

Introduction

C++ provides **struct**, **union**, and **enum** as user-defined data types. They allow grouping of variables under one name for efficient data management.

6.1 Structure (`struct`)

Definition

A structure is a collection of variables (of different types) grouped together under a single name.

Syntax

```
struct StructName {
    dataType member1;
    dataType member2;
    ...
};
```

Listing 6.1: Syntax of struct

Example

```
#include <iostream>
using namespace std;

struct Student {
    int roll;
    float marks;
    char grade;
};
```

```
int main() {
    Student s1 = {101, 95.5, 'A'};
    cout << "Roll: " << s1.roll << "\n";
    cout << "Marks: " << s1.marks << "\n";
    cout << "Grade: " << s1.grade << "\n";
    return 0;
}
```

Listing 6.2: C++ Code using struct

Accessing Members with Pointer

```
Student s = {102, 88.4, 'B'};
Student *ptr = &s;
cout << ptr->roll << "\n";
cout << ptr->marks << "\n";
cout << ptr->grade << "\n";
```

Listing 6.3: Accessing struct members using pointer

Memory Representation

Member	Type	Size (Bytes)	Offset
roll	int	4	0
marks	float	4	4
grade	char	1	8 (may pad to 12)

Diagram

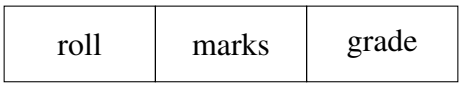


Figure 6.1: Memory layout of struct Student

Advantages

- Group heterogeneous data types.
- Enhances code clarity and reusability.
- Useful in data modeling (e.g., records, databases).

6.2 Union (union)

Definition

A union allows storing different data types in the same memory location. Only one member can hold a value at a time.

Syntax

```
union UnionName {
    dataType member1;
    dataType member2;
    ...
};
```

Listing 6.4: Syntax of union

Example

```
#include <iostream>
using namespace std;

union Data {
    int i;
    float f;
};

int main() {
    Data d;
    d.i = 10;
    cout << "Integer: " << d.i << "\n";

    d.f = 3.14;
    cout << "Float: " << d.f << "\n";

    // Now d.i is likely corrupted
    cout << "Integer after float: " << d.i << "\n";
    return 0;
}
```

Listing 6.5: C++ Code using union

Memory Representation

Member	Type	Size (Bytes)
i	int	4
f	float	4

Total Memory: Max of all members (4 bytes here)

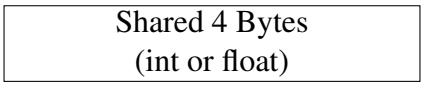


Figure 6.2: Union memory layout (shared memory)

Advantages

- Saves memory when only one member is used at a time.
- Useful in embedded systems and low-level programming.

6.3 Enumeration (`enum`)

Definition

An enum is a user-defined type consisting of a set of named integer constants.

Syntax

```
enum EnumName {
    CONSTANT1,
    CONSTANT2,
    ...
};
```

Listing 6.6: Syntax of enum

Example

```
#include <iostream>
using namespace std;

enum Day {Mon, Tue, Wed, Thu, Fri, Sat, Sun};

int main() {
    Day d = Wed;
    cout << "Value of Wed: " << d << "\n";
    return 0;
}
```

Listing 6.7: C++ Code using enum

Custom Values and Iteration

```
enum Status {Pending=1, Approved=5, Rejected=10};

int main() {
    Status s = Approved;
    cout << "Status code: " << s << "\n";
    return 0;
}
```

Listing 6.8: Enum with custom values and loop

Advantages

- Improves code readability.
- Restricts variable to valid set of values.
- Useful in state machines, switch cases.

Underlying Values

```
By default: Mon = 0, Tue = 1, ..., Sun = 6
Custom: You can assign any value explicitly
```

Enum Constant	Value
Mon	0
Tue	1
Wed	2
Thu	3
Fri	4
Sat	5
Sun	6

Part II

Linear Data Structures

Chapter 7

Arrays

Introduction

An **array** is a collection of elements, all of the same type, stored in contiguous memory locations. It allows for efficient access using indices. Arrays can be *static* (fixed size) or *dynamic* (resized at runtime). In C++, the STL `vector` provides a dynamic array implementation.

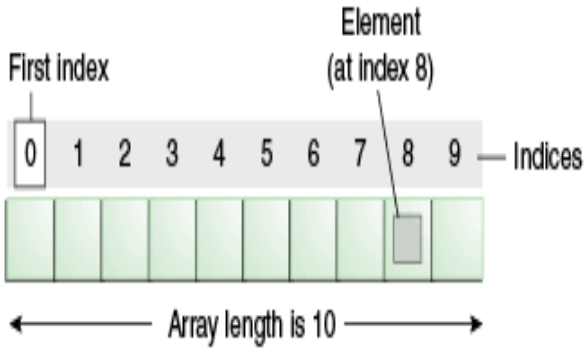


Figure 7.1: Conceptual representation of an array in memory

7.1 Array Declarations and Syntax

1D Array Declaration

Static (C/C++):

```
int arr[5]; // Declares an array of 5 integers
```

Listing 7.1: Static 1D Array Declaration

Dynamic (C++ using pointers):

```
int* arr = new int[5]; // Allocates memory for 5 integers dynamically
```

Listing 7.2: Dynamic 1D Array Declaration using Pointers

Using STL Vector (C++):

```cpp
#include <vector>
vector<int> arr(5); // Vector of size 5
```

Listing 7.3: 1D Array using STL Vector

2D Array Declaration

Static (C/C++):

```cpp
int matrix[3][4]; // 3 rows and 4 columns
```

Listing 7.4: Static 2D Array Declaration

Dynamic (C++ using pointers):

```cpp
int** matrix = new int*[3];
for(int i = 0; i < 3; i++) {
    matrix[i] = new int[4];
}
```

Listing 7.5: Dynamic 2D Array Declaration using Pointers

Using STL Vector (C++):

```cpp
#include <vector>
vector<vector<int>> matrix(3, vector<int>(4)); // 3x4 2D vector
```

Listing 7.6: 2D Array using STL Vector

3D Array Declaration (Using STL Vector)

```cpp
#include <vector>
vector<vector<vector<int>>> arr3D(
    2, vector<vector<int>>(
          3, vector<int>(4)));
// 2x3x4 3D vector
```

Listing 7.7: 3D Array using STL Vector

7.2 Time and Space Complexity

Time Complexity

- **Access:** $O(1)$ - Direct access via index.
- **Search:** $O(n)$ - Linear search in the worst case.
- **Insertion/Deletion at End:** $O(1)$ - Amortized time.
- **Insertion/Deletion at Beginning or Middle:** $O(n)$ - Requires shifting elements.

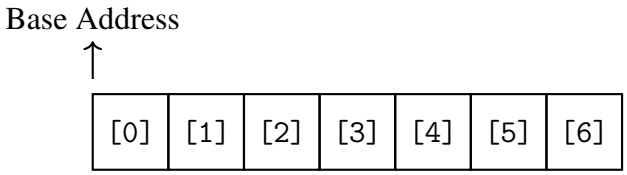
Space Complexity

- **Space:** $O(n)$ - Where n is the number of elements in the array.

7.3 Memory Allocation and Address Calculation

Memory Allocation

Arrays are stored in contiguous memory locations. For an array declared as `int arr[n];`, the memory layout is illustrated below:



Address Calculation

The address of the ith element is computed as:

$$\text{Address of } arr[i] = \text{Base Address} + i \times \text{Size of each element}$$

Indexing and Byte Addressing

0-Based Indexing:

$$\text{Address of } arr[i] = \text{Base Address} + i \times \text{Size}$$

1-Based Indexing:

$$\text{Address of } arr[i] = \text{Base Address} + (i-1) \times \text{Size}$$

7.4 Passing Arrays to Functions

Example: Passing a 1D Array to a Function

```cpp
#include <bits/stdc++.h>
using namespace std;

void printArray(int arr[], int size) {
    for (int i = 0; i < size; ++i) {
        cout << arr[i] << " ";
    }
    cout << "\n";
}

int main() {
    int myArray[] = {10, 20, 30, 40, 50};
    int size = sizeof(myArray) / sizeof(myArray[0]);
```

```
    printArray(myArray, size);
    return 0;
}
```

Listing 7.8: Passing Array to Function

7.5 Two-Dimensional Arrays and Matrix Representation

Row-Major Order

The address of element $A[i][j]$ is given by:

$$\text{Address of } A[i][j] = B + [(i \cdot N) + j] \cdot w$$

where B is the base address, N is the number of columns, and w is the size of each element.

Column-Major Order

The address of element $A[i][j]$ is:

$$\text{Address of } A[i][j] = B + [(j \cdot M) + i] \cdot w$$

where M is the number of rows.

Example: 2D Array in C++ (Row-Major Order)

```
#include <bits/stdc++.h>
using namespace std;

int main() {
    int A[2][3] = {{1, 2, 3}, {4, 5, 6}};

    for (int i = 0; i < 2; i++) {
        for (int j = 0; j < 3; j++) {
            cout << A[i][j] << " ";
        }
        cout << endl;
    }
    return 0;
}
```

Listing 7.9: 2D Array in Row-Major Order

7.6 Multidimensional Arrays

Beyond 2D arrays, multidimensional arrays (such as 3D arrays) are used in applications like 3D graphics, simulations, and complex data representations.

Example: 3D Array (Using STL Vector in C++)

```cpp
#include <bits/stdc++.h>
using namespace std;

int main() {
    vector<vector<vector<int>>> arr3D(
        2, vector<vector<int>>(
              3, vector<int>(4, 0))); // 2x3x4 3D vector initialized
                 with 0

    // Accessing an element:
    arr3D[0][1][2] = 42;

    cout << "Element at [0][1][2]: " << arr3D[0][1][2] << endl;
    return 0;
}
```

Listing 7.10: 3D Array Declaration using STL Vector

7.7 Matrix Representation and Applications

A matrix is typically represented as a 2D array and is used in:

- Image representation (pixels)
- Graph adjacency matrices
- Dynamic programming tables
- Mathematical computations

7.8 Drawbacks of Arrays

Despite their efficiency, arrays have some limitations:

1. **Fixed Size:** The size must be known at compile time for static arrays.
2. **Memory Waste:** Overestimating size leads to unused memory, while underestimating can cause overflow.
3. **Costly Insertions/Deletions:** Shifting elements results in $O(n)$ time operations.
4. **No Built-in Bound Checking:** Accessing out-of-bound indices causes undefined behavior.
5. **Homogeneous Data:** Arrays store elements of the same type only.
6. **Contiguous Memory Requirement:** Large blocks of contiguous memory might not be available in fragmented systems.

Solution: Use STL containers like `vector` for dynamic arrays.

7.9 Summary Table of Array Types and Syntax

Array Type	Declaration Syntax	Memory Allocation	Example / Notes
1D Static Array	`int arr[5];`	Stack, Contiguous	`int arr[5] = {1, 2, 3, 4, 5};`
1D Dynamic Array	`int* arr = new int[5];`	Heap, Contiguous	Allocated at runtime using `new`
2D Static Array	`int matrix[3][4];`	Stack, Row-major	3 rows, 4 columns declared statically
2D Dynamic Array	`int** matrix = new int*[3];`	Heap (row pointers), Contiguous rows	Allocate each row: `matrix[i] = new int[4];`
STL 1D Vector	`vector<int> arr(5);`	Heap, Dynamic resizing	Uses STL; resizable, auto-managed memory
STL 2D Vector	`vector<vector<int»> matrix;`	Heap, Nested Vectors	Allows jagged arrays; dynamic allocation
STL 3D Vector	`vector<vector<..<int>.>`	Heap, Fully Dynamic	Example: `vector<vector<vector<int»> arr3D(2, vector<vector<int»(3, vector<int>(4)));`

Table 7.1: Summary of Array Types, Memory Allocation, and Syntax in C++

7.10 Additional Diagrams

1D Array Memory Layout

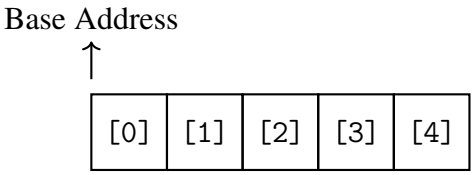


Figure 7.2: Memory layout of a 1D array with 5 elements.

2D Array Memory Layout (Row-Major)

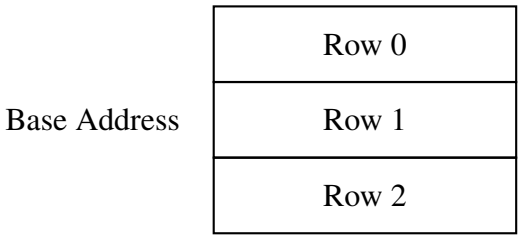


Figure 7.3: Row-major layout of a 2D array (3 rows, 4 columns).

7.11 Index Diagram and Table

Consider a 1D array with 5 elements. The following table and diagram illustrate how array indices correspond to the stored elements.

Index Mapping Table

Index	Element (Label)
0	`arr[0]`
1	`arr[1]`
2	`arr[2]`
3	`arr[3]`
4	`arr[4]`

Table 7.2: Mapping of indices to array elements for a 1D array

1D Array Index Diagram

Figure 7.4: Index diagram for a 1D array of 5 elements

Chapter 8

Linked List

Introduction

Linked lists are dynamic data structures in which nodes are linked together via pointers. They allow efficient insertions and deletions compared to static arrays. In this chapter, we cover three types of linked lists:

1. **Singly Linked List**
2. **Doubly Linked List**
3. **Circular Linked List**

Each type is explained in detail with corresponding C++ implementations, diagrams, and a summary of time and space complexities for common operations.

8.1 Singly Linked List

8.1.1 Node Structure (Singly)

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

Listing 8.1: Singly Linked List Node Structure in C++

Theory: A singly linked list consists of nodes where each node holds data and a pointer to the next node. This design makes the list flexible in size, allowing nodes to be added or removed without reallocating or reorganizing the entire structure. However, because each node only points forward, traversing the list in reverse order is not straightforward without additional modifications.

8.1.2 Operations on Singly Linked List

Insertion

```
void insertAtFront(Node*& head, int val) {
    Node* newNode = new Node(val);
    newNode->next = head;
    head = newNode;
}
```

Listing 8.2: Insert at Front in Singly Linked List

```
void insertAtEnd(Node*& head, int val) {
    Node* newNode = new Node(val);
    if (!head) {
        head = newNode;
        return;
    }
    Node* temp = head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
}
```

Listing 8.3: Insert at End in Singly Linked List

```
void insertAtPosition(Node*& head, int pos, int val) {
    if (pos == 0) {
        insertAtFront(head, val);
        return;
    }
    Node* newNode = new Node(val);
    Node* temp = head;
    for (int i = 0; temp != nullptr && i < pos - 1; i++)
        temp = temp->next;
    if (!temp) return; // Position out of bounds
    newNode->next = temp->next;
    temp->next = newNode;
}
```

Listing 8.4: Insert at Position in Singly Linked List

Insertion Theory: Insertion at the front of a singly linked list is very efficient ($O(1)$) because it simply involves updating the head pointer. Insertion at the end requires traversal of the entire list, resulting in $O(n)$ time complexity. When inserting at a given position, the list must be traversed until that position is reached, making it $O(n)$ as well. These operations illustrate the trade-off between flexibility and direct access in linked lists.

Deletion

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

Listing 8.5: Delete from Front in Singly Linked List

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

Listing 8.6: Delete from End in Singly Linked List

```
void deleteAtPosition(Node*& head, int pos) {
    if (!head) return;
    if (pos == 0) {
        deleteFromFront(head);
        return;
    }
    Node* temp = head;
    for (int i = 0; temp != nullptr && i < pos - 1; i++)
        temp = temp->next;
    if (!temp || !temp->next) return; // Position out of bounds
    Node* delNode = temp->next;
    temp->next = delNode->next;
    delete delNode;
}
```

Listing 8.7: Delete from Position in Singly Linked List

Deletion Theory: Deletion operations in a singly linked list are efficient if the node to be deleted is known ($O(1)$). However, if a search is needed to find the node, the operation becomes $O(n)$. The simplicity of pointer adjustments in deletion is a key advantage over array-based structures.

Search

```
int search(Node* head, int key) {
    int index = 0;
    while (head) {
        if (head->data == key)
            return index;
        head = head->next;
        index++;
    }
    return -1; // Not found
}
```

Listing 8.8: Search in Singly Linked List

Search Theory: Since singly linked lists are sequential, searching for an element requires traversing the list from the head until the element is found, leading to $O(n)$ time complexity. This is less efficient compared to direct indexing in arrays but is acceptable given the dynamic nature of linked lists.

8.1.3 Diagram: Singly Linked List

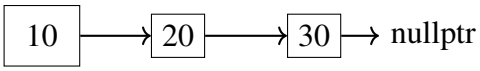


Figure 8.1: Singly Linked List Diagram

8.2 Doubly Linked List

8.2.1 Node Structure (Doubly)

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

Listing 8.9: Doubly Linked List Node Structure in C++

8.2.2 Operations on Doubly Linked List

Insertion

```
void insertAtFront(DNode*& head, int val) {
    DNode* newNode = new DNode(val);
    newNode->next = head;
    if (head)
        head->prev = newNode;
    head = newNode;
}
```

Listing 8.10: Insert at Front in Doubly Linked List

```
void insertAtEnd(DNode*& head, int val) {
    DNode* newNode = new DNode(val);
    if (!head) {
        head = newNode;
        return;
    }
    DNode* temp = head;
    while (temp->next)
        temp = temp->next;
    temp->next = newNode;
    newNode->prev = temp;
}
```

Listing 8.11: Insert at End in Doubly Linked List

Theory: Doubly linked lists maintain two pointers in each node, which allows traversal in both directions. This feature simplifies certain operations like deletion and backward traversal but requires additional memory for the extra pointer.

Deletion

```
void deleteFromFront(DNode*& head) {
    if (!head) return;
    DNode* temp = head;
    head = head->next;
    if (head)
        head->prev = nullptr;
    delete temp;
}
```

Listing 8.12: Delete from Front in Doubly Linked List

```
void deleteFromEnd(DNode*& head) {
    if (!head) return;
    if (!head->next) {
        delete head;
        head = nullptr;
        return;
    }
    DNode* temp = head;
    while (temp->next)
        temp = temp->next;
    temp->prev->next = nullptr;
    delete temp;
}
```

Listing 8.13: Delete from End in Doubly Linked List

```
int search(DNode* head, int key) {
    int index = 0;
    while (head) {
        if (head->data == key)
            return index;
        head = head->next;
        index++;
    }
    return -1;
}
```

Listing 8.14: Search in Doubly Linked List

8.2.3 Diagram: Doubly Linked List

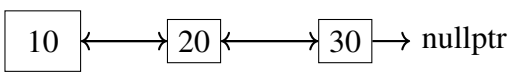


Figure 8.2: Doubly Linked List Diagram

8.3 Circular Linked List

8.3.1 Node Structure (Circular)

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

Listing 8.15: Circular Linked List Node Structure in C++

8.3.2 Operations on Circular Linked List

Insertion

```
void insertCircular(CNode*& head, int val) {
    CNode* newNode = new CNode(val);
    if (!head) {
        head = newNode;
        newNode->next = head;
        return;
    }
    CNode* temp = head;
    while (temp->next != head)
        temp = temp->next;
    temp->next = newNode;
    newNode->next = head;
}
```

Listing 8.16: Insert in Circular Linked List

Deletion

```
void deleteCircular(CNode*& head, int key) {
    if (!head) return;
    // If head is to be deleted
    if (head->data == key) {
        if (head->next == head) { // Only one node exists
            delete head;
            head = nullptr;
            return;
        }
        // Find last node to update its next pointer
        CNode* temp = head;
        while (temp->next != head)
            temp = temp->next;
        CNode* del = head;
```

```
        head = head->next;
        temp->next = head;
        delete del;
        return;
    }
    // Delete node other than head
    CNode* curr = head;
    while (curr->next != head && curr->next->data != key)
        curr = curr->next;
    if (curr->next->data == key) {
        CNode* del = curr->next;
        curr->next = del->next;
        delete del;
    }
}
```

Listing 8.17: Delete Node in Circular Linked List

Search

```
int searchCircular(CNode* head, int key) {
    if (!head) return -1;
    int index = 0;
    CNode* temp = head;
    do {
        if (temp->data == key)
            return index;
        temp = temp->next;
        index++;
    } while (temp != head);
    return -1;
}
```

Listing 8.18: Search in Circular Linked List

8.3.3 Diagram: Circular Linked List

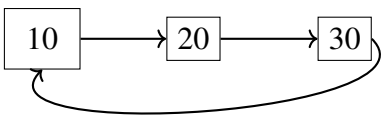


Figure 8.3: Circular Linked List Diagram

8.4 Complexity Analysis for Linked List Operations

Operation	Time Complexity	Space Complexity
Access	$O(n)$	$O(n)$ (node overhead)
Search	$O(n)$	$O(1)$
Insertion (given pointer)	$O(1)$	$O(1)$
Insertion (search required)	$O(n)$	$O(1)$
Deletion (given pointer)	$O(1)$	$O(1)$
Deletion (search required)	$O(n)$	$O(1)$

Table 8.1: Time and Space Complexity of Linked List Operations

8.5 Syntax Summary for Linked List Node Structures

Type	Node Structure Syntax
Singly Linked List	`struct Node { int data; Node* next; }`
Doubly Linked List	`struct DNode { int data; DNode* next; DNode* prev; }`
Circular Linked List	`struct CNode { int data; CNode* next; }`

Table 8.2: Node Structure Syntax for Different Linked Lists

8.6 Additional Theoretical Concepts

8.6.1 Singly Linked List Theory

Singly linked lists are one of the simplest forms of linked lists. They are ideal for implementations where only forward traversal is needed. Their simplicity, however, comes at a cost: operations like finding the previous node require traversing the list from the head, which can be inefficient. They are best used in scenarios where memory reallocation is frequent and operations mostly involve adding or removing nodes at the head.

8.6.2 Doubly Linked List Theory

Doubly linked lists enhance singly linked lists by providing backward pointers. This allows efficient bidirectional traversal, which simplifies operations like deletion from the end or inserting before a given node. However, this comes at the cost of additional memory for the extra pointer, and the operations are slightly more complex because both pointers must be updated during insertions and deletions.

8.6.3 Circular Linked List Theory

Circular linked lists differ from the other types in that the last node points back to the first node, forming a continuous loop. This structure is useful for applications where the list needs to be traversed cyclically (e.g., round-robin scheduling). They eliminate the need for null checks at the end of the list, but careful handling is required during insertion and deletion to avoid infinite loops.

8.6.4 Memory Management and Pointer Usage

Linked lists dynamically allocate memory for each node, which means memory is used only as needed. However, the overhead of storing pointers in each node can be significant compared to array elements. In systems with limited memory, the trade-off between flexibility and memory overhead must be considered. Garbage collection (in languages that support it) or manual memory management in C++ is essential to avoid memory leaks.

8.7 Conclusion

Linked lists are versatile and powerful dynamic data structures that provide efficient insertion and deletion operations. Each type—singly, doubly, and circular—has its own advantages and trade-offs. Singly linked lists are simple and efficient for forward traversal; doubly linked lists allow for bidirectional movement; and circular linked lists facilitate continuous looping without explicit null termination. Understanding the underlying theory, along with the syntax and complexity of various operations, is crucial for designing efficient data structures and algorithms.

This chapter provides both practical C++ implementations and theoretical insights to help you master linked list concepts.

Chapter 9

Stack

9.1 Introduction

A **stack** is a fundamental abstract data type that operates on the Last In First Out (LIFO) principle. In other words, the last element added to the stack is the first one to be removed. Stacks are widely used in various computing applications, including expression evaluation, recursion management, and backtracking algorithms.

9.2 Definition and Theoretical Background

A stack supports the following primary operations:

- **Push**: Add an element to the top of the stack.
- **Pop**: Remove the element from the top of the stack.
- **Top/Peek**: Retrieve the top element without removing it.
- **IsEmpty**: Check whether the stack is empty.

Formally, a stack S can be defined as a tuple (D, P), where:

- D is a finite set of data elements.
- P is a pointer or index indicating the top element in the stack.

The operations are defined as:

- **Push**: For a stack $S = (D, P)$, pushing an element x transforms it into $S' = (D \cup \{x\}, x)$. The new element becomes the top element.
- **Pop**: Removing the top element x from S results in a new stack $S' = (D \setminus \{x\}, y)$, where y is now the top element.
- **Top/Peek**: This operation retrieves x (the current top element) without modifying the stack.
- **IsEmpty**: A boolean function that returns true if no elements are in D (i.e., P is undefined or negative).

9.2.1 Additional Theoretical Considerations

There are two common ways to implement a stack:

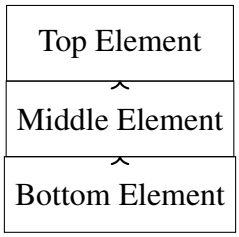
- **Array-based Implementation**: Uses a fixed-size array to store elements. It is simple and fast but has a size limitation.
- **Linked List Implementation**: Uses nodes where each node points to the next element. It can grow dynamically, although it may incur extra memory overhead.

Understanding these two implementations is essential because each has its own performance characteristics and trade-offs in terms of time and space complexity.

9.3 Diagrams of Stack Operations

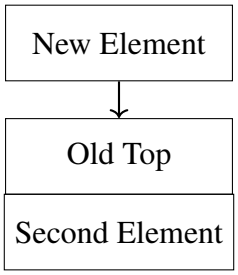
9.3.1 Overall Stack Structure

The following diagram shows a simple stack with three elements. The top element is at the uppermost position:



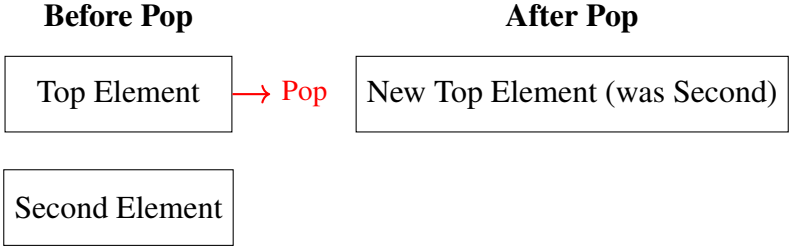
9.3.2 Push Operation Diagram

The diagram below illustrates a push operation. A new element is inserted and becomes the new top:



9.3.3 Pop Operation Diagram

The following diagram shows the pop operation, where the top element is removed:



9.4 C++ Implementation of a Stack

Below is an extended implementation of a stack using an array in C++ with additional inline comments for clarity.

```cpp
#include <bits/stdc++.h>
#define MAX 100

class Stack {
private:
    int arr[MAX]; // Array to store stack elements
    int top;     // Index of the top element
public:
    // Constructor initializes top to -1 indicating an empty stack
    Stack() : top(-1) {}

    // Check if the stack is empty
    bool isEmpty() {
        return (top < 0);
    }
    // Push an element onto the stack
    bool push(int x) {
        if(top >= (MAX - 1)) {
            cout << "Stack Overflow" << "\n";
            return false;
        } else {
            arr[++top] = x;
            return true;
        }
    }
    // Pop an element from the stack
    int pop() {
        if(isEmpty()){
            cout << "Stack Underflow" << "\n";
            return -1;
        } else {
            int x = arr[top--];
            return x;
        }
    }
    // Peek at the top element without removing it
    int peek() {
        if(isEmpty()){
            cout << "Stack is Empty" << "\n";
            return -1;
        } else {
            return arr[top];
        }
    }
```

```
};
int main() {
    Stack s;
    // Example operations on the stack
    s.push(10);  // Stack: [10]
    s.push(20);  // Stack: [10, 20]
    s.push(30);  // Stack: [10, 20, 30]
    cout << "Top element is " << s.peek() << "\n"; // Should print 30
    cout << "Popped element is " << s.pop() << "\n"; // Removes 30
    cout << "New top element is " << s.peek() << "\n"; // Should print
       20

    return 0;
}
```

Listing 9.1: C++ implementation of a Stack using an array

9.5 Example Scenario in Detail

Consider the following sequence of operations on an initially empty stack:

1. **Push** 5 onto the stack.
   Stack state: [5] (5 becomes the top element)

2. **Push** 10 onto the stack.
   Stack state: [5, 10] (10 is now at the top)

3. **Push** 15 onto the stack.
   Stack state: [5, 10, 15] (15 becomes the new top)

4. **Pop** the top element.
   Stack state: [5, 10] (15 is removed; 10 becomes the top)

The above scenario is illustrated in the diagrams provided earlier. When a new element is pushed, it becomes the new top, and when an element is popped, the previous element becomes the top.

9.6 Infix, Postfix, and Prefix Conversion

Expressions can be written in three common forms:

- **Infix:** The operator is written between the operands (e.g., `A + B`).
- **Postfix (Reverse Polish Notation):** The operator follows the operands (e.g., `A B +`).
- **Prefix (Polish Notation):** The operator precedes the operands (e.g., `+ A B`).

9.6.1 Theory and Conversion Algorithms

To convert an infix expression to postfix, follow these steps:

1. Scan the infix expression from left to right.
2. Use a stack to store operators and an output string for the result.
3. When an operand (letter or number) is encountered, append it to the output.
4. When an operator is encountered, pop from the stack to the output until either the stack is empty or the operator at the top of the stack has lower precedence than the current operator. Then, push the current operator onto the stack.
5. When a left parenthesis (is encountered, push it onto the stack.
6. When a right parenthesis) is encountered, pop from the stack to the output until a left parenthesis is encountered; then discard the pair of parentheses.
7. After processing the entire expression, pop any remaining operators from the stack to the output.

For **infix to prefix conversion**, a common method is:

- Reverse the infix expression.
- Swap every (with) and vice-versa.
- Convert the modified expression to postfix.
- Reverse the resulting postfix expression to obtain the prefix expression.

9.6.2 Extended C++ Implementation

Below is an extended C++ code that implements both infix-to-postfix and infix-to-prefix conversion. Notice the use of helper functions to reverse strings and swap parentheses.

```
#include <bits/stdc++.h>

using namespace std;

// Function to return precedence of operators
int precedence(char op) {
    if(op == '+' || op == '-') return 1;
    if(op == '*' || op == '/') return 2;
    return 0;
}
// Infix to Postfix conversion
string infixToPostfix(const string &infix) {
    string postfix;
    stack<char> s;
    for (char c : infix) {
        // If the character is an operand, append it to output
```

```
        if(isalnum(c))
            postfix += c;
        // If '(' encountered, push to stack
        else if(c == '(')
            s.push(c);
        // If ')' encountered, pop until '('
        else if(c == ')') {
            while(!s.empty() && s.top() != '(') {
                postfix += s.top();
                s.pop();
            }
            if(!s.empty())
                s.pop(); // pop '('
        }
        // If operator encountered
        else {
            while(!s.empty() && precedence(c) <= precedence(s.top())) {
                postfix += s.top();
                s.pop();
            }
            s.push(c);
        }
    }
    // Pop remaining operators
    while(!s.empty()) {
        postfix += s.top();
        s.pop();
    }
    return postfix;
}
// Utility function to reverse a string
string reverseString(string s) {
    reverse(s.begin(), s.end());
    return s;
}
// Swap parentheses in a string
string swapParentheses(string s) {
    for (char &c : s) {
        if(c == '(') c = ')';
        else if(c == ')') c = '(';
    }
    return s;
}

// Infix to Prefix conversion using reverse-process method
string infixToPrefix(string infix) {
    // Reverse the infix expression and swap parentheses
    string rev = reverseString(infix);
```

```
    rev = swapParentheses(rev);
    // Convert reversed expression to postfix
    string revPostfix = infixToPostfix(rev);
    // Reverse the postfix expression to get prefix
    string prefix = reverseString(revPostfix);
    return prefix;
}

int main() {
    // Example 1
    string infix1 = "((A+B)*C)-(D/E)";
    cout << "Infix 1: " << infix1 << endl;
    cout << "Postfix 1: " << infixToPostfix(infix1) << endl;
    cout << "Prefix 1: " << infixToPrefix(infix1) << endl << endl;

    // Example 2
    string infix2 = "A*(B+C)/D";
    cout << "Infix 2: " << infix2 << endl;
    cout << "Postfix 2: " << infixToPostfix(infix2) << endl;
    cout << "Prefix 2: " << infixToPrefix(infix2) << endl;

    return 0;
}
```

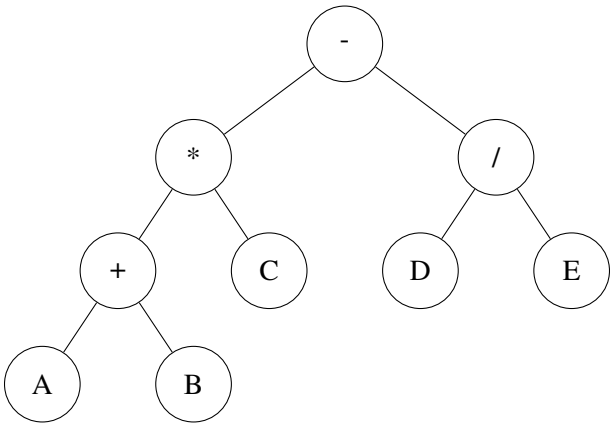
Listing 9.2: Extended C++ Code for Infix, Postfix, and Prefix Conversion

9.6.3 Expression Tree Diagrams

The following diagrams represent the expression trees for the given expressions. These trees help visualize how the expressions are structured and how different traversals yield the infix, postfix, and prefix notations.

Example 1: $((A+B)*C)-(D/E)$

Expression Tree:

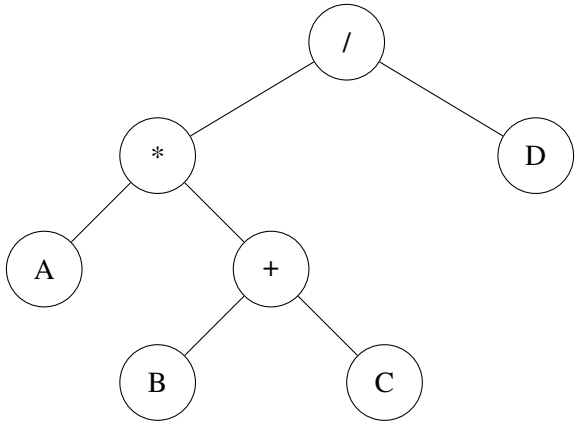


Traversals:

- **Inorder (Infix):** $((A+B)*C)-(D/E)$
- **Postorder (Postfix):** $AB+C*DE/-$
- **Preorder (Prefix):** $-*+ABC/DE$

Example 2: $A*(B+C)/D$

Expression Tree:



Traversals:

- **Inorder (Infix):** $A*(B+C)/D$
- **Postorder (Postfix):** $ABC+*D/$
- **Preorder (Prefix):** $/*A+BCD$

This section has provided detailed theory, extended C++ implementations, additional examples, and visual diagrams. With these tools, you should be able to understand and implement conversions between infix, postfix, and prefix notations effectively.

9.7 C++ Implementation of a Stack using Linked List

Unlike the array-based stack, a linked list implementation does not have a fixed size. It can grow and shrink dynamically as needed. Each node in the linked list stores a value and a pointer to the next node.

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

class StackLL {
```

```
private:
    Node* top;

public:
    // Constructor
    StackLL() {
        top = nullptr;
    }
    // Push operation
    void push(int x) {
        Node* temp = new Node();
        if (!temp) {
            cout << "Heap Overflow\n";
            return;
        }
        temp->data = x;
        temp->next = top;
        top = temp;
    }
    // Check if the stack is empty
    bool isEmpty() {
        return top == nullptr;
    }
    // Pop operation
    void pop() {
        if (isEmpty()) {
            cout << "Stack Underflow\n";
            return;
        }
        Node* temp = top;
        top = top->next;
        delete temp;
    }

    // Peek operation
    int peek() {
        if (!isEmpty())
            return top->data;
        else {
            cout << "Stack is Empty\n";
            return -1;
        }
    }
    // Display contents of stack
    void display() {
        if (isEmpty()) {
            cout << "Stack is empty\n";
            return;
```

```
        }
        Node* temp = top;
        while (temp != nullptr) {
            cout << temp->data << " -> ";
            temp = temp->next;
        }
        cout << "NULL\n";
    }
};
int main() {
    StackLL s;
    s.push(5);
    s.push(10);
    s.push(15);
    cout << "Top element is: " << s.peek() << "\n";
    s.pop();
    s.display();

    return 0;
}
```

Listing 9.3: C++ implementation of a Stack using a Linked List

This linked list-based stack grows dynamically, avoiding stack overflow unless system memory is exhausted. The ‘push‘, ‘pop‘, and ‘peek‘ operations each run in $O(1)$ time, offering efficient performance and flexibility.

9.8 Conclusion

Stacks are a fundamental data structure that implements the Last In First Out (LIFO) principle. This chapter has detailed the stack's theoretical basis, explained each operation in depth, and presented multiple diagrams to visually represent the stack and its operations. Additionally, a practical C++ implementation is provided to illustrate how stacks can be implemented and used in real-world applications, such as managing function calls and evaluating expressions.

Chapter 10

Queue and Priority Queue

10.1 Introduction

A **queue** is a linear data structure that follows the First In First Out (FIFO) principle. Elements are inserted at the rear and removed from the front, making it ideal for scheduling tasks, managing requests, and simulating real-world queues. A **priority queue** is an extension where each element is associated with a priority, and elements are dequeued based on their priority rather than solely on their order of insertion.

10.2 Definition and Theoretical Background

A standard queue supports the following operations:

- **Enqueue**: Add an element to the rear of the queue.
- **Dequeue**: Remove the element from the front of the queue.
- **Front**: Retrieve the front element without removing it.
- **IsEmpty**: Check whether the queue is empty.
- **IsFull**: (In array implementations) Check whether the queue has reached its capacity.

For a queue Q, the state can be defined by two indices (or pointers):

- front: Index of the first element.
- rear: Index of the last element.

10.2.1 Time Complexity for Queue Operations

- **Enqueue**: $O(1)$ for both array (if implemented circularly) and linked list.
- **Dequeue**: $O(1)$ for both implementations.
- **Front**: $O(1)$ access.

A **priority queue** maintains elements along with a priority value. The key operations are:

- **Enqueue (Insert)**: Insert an element with its priority.
- **Dequeue (Delete)**: Remove the element with the highest priority (the definition of highest priority may vary; here we assume a lower numerical value indicates higher priority).
- **Peek**: Look at the element with the highest priority.

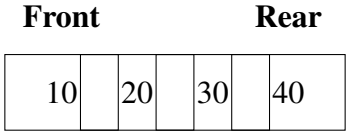
10.2.2 Time Complexity for Priority Queue Operations

- **Array-based (Unsorted)**:
  - Enqueue: $O(1)$
  - Dequeue: $O(n)$ (since finding the highest priority element takes linear time)
- **Linked List (Sorted)**:
  - Enqueue: $O(n)$ (to insert at the correct position)
  - Dequeue: $O(1)$ (removing from the front)

10.3 Diagrams of Queue Operations

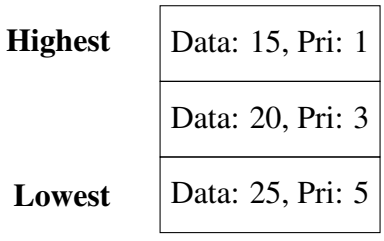
10.3.1 Standard Queue Diagram

The following diagram illustrates a standard queue with a `front` and a `rear` pointer.



10.3.2 Priority Queue Diagram

Below is a diagram for a priority queue. Note that elements are arranged according to priority rather than insertion order.



10.4 C++ Implementations

10.4.1 Queue Using Array

This implementation uses a fixed-size array. Note that a circular array implementation could handle wrap-around more elegantly, but here we use a simple version.

```
#include <iostream>
using namespace std;
#define MAX 100

class Queue {
private:
    int arr[MAX];
    int front, rear;
public:
    Queue() {
        front = 0;
        rear = -1;
    }
    bool isEmpty() {
        return (front > rear);
    }
    bool isFull() {
        return (rear == MAX - 1);
    }
    void enqueue(int x) {
        if(isFull()) {
            cout << "Queue Overflow\n";
            return;
        }
        arr[++rear] = x;
    }
    int dequeue() {
        if(isEmpty()) {
            cout << "Queue Underflow\n";
            return -1;
        }
        return arr[front++];
    }
    int getFront() {
        if(isEmpty()) {
            cout << "Queue is empty\n";
            return -1;
        }
        return arr[front];
    }
};

int main() {
    Queue q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
```

```
    cout << "Front element is: " << q.getFront() << "\n"; // Should
        print 10
    cout << "Dequeued element is: " << q.dequeue() << "\n"; // Removes
        10
    cout << "New front element is: " << q.getFront() << "\n"; // Should
         print 20
    return 0;
}
```

Listing 10.1: C++ implementation of a Queue using an Array

10.4.2 Queue Using Linked List

This implementation uses nodes to allow the queue to grow dynamically.

```
#include <iostream>
using namespace std;

class Node {
public:
    int data;
    Node* next;
};

class QueueLL {
private:
    Node* front;
    Node* rear;
public:
    QueueLL() {
        front = rear = nullptr;
    }
    bool isEmpty() {
        return (front == nullptr);
    }
    void enqueue(int x) {
        Node* temp = new Node();
        temp->data = x;
        temp->next = nullptr;
        if(rear == nullptr) {
            front = rear = temp;
            return;
        }
        rear->next = temp;
        rear = temp;
    }
    int dequeue() {
        if(isEmpty()) {
            cout << "Queue Underflow\n";
```

```
            return -1;
        }
        int x = front->data;
        Node* temp = front;
        front = front->next;
        if(front == nullptr)
            rear = nullptr;
        delete temp;
        return x;
    }
    int getFront() {
        if(isEmpty()) {
            cout << "Queue is empty\n";
            return -1;
        }
        return front->data;
    }
};

int main() {
    QueueLL q;
    q.enqueue(10);
    q.enqueue(20);
    q.enqueue(30);
    cout << "Front element is: " << q.getFront() << "\n"; // Should
       print 10
    cout << "Dequeued element is: " << q.dequeue() << "\n"; // Removes
       10
    cout << "New front element is: " << q.getFront() << "\n"; // Should
        print 20
    return 0;
}
```

Listing 10.2: C++ implementation of a Queue using a Linked List

10.4.3 Priority Queue Using Array (Unsorted)

In this version, insertion is $O(1)$ while deletion (removing the highest priority element) is $O(n)$.

```
#include <iostream>
using namespace std;
#define MAX 100

struct Element {
    int data;
    int priority;
};

class PriorityQueue {
```

```
private:
    Element arr[MAX];
    int size;
public:
    PriorityQueue() {
        size = 0;
    }
    bool isEmpty() {
        return (size == 0);
    }
    bool isFull() {
        return (size == MAX);
    }
    void enqueue(int data, int priority) {
        if(isFull()) {
            cout << "Priority Queue Overflow\n";
            return;
        }
        arr[size].data = data;
        arr[size].priority = priority;
        size++;
    }
    int dequeue() {
        if(isEmpty()) {
            cout << "Priority Queue Underflow\n";
            return -1;
        }
        // Find element with highest priority (lower number indicates
           higher priority)
        int idx = 0;
        for (int i = 1; i < size; i++) {
            if(arr[i].priority < arr[idx].priority) {
                idx = i;
            }
        }
        int data = arr[idx].data;
        // Remove the element by shifting subsequent elements
        for (int i = idx; i < size - 1; i++) {
            arr[i] = arr[i + 1];
        }
        size--;
        return data;
    }
};

int main() {
    PriorityQueue pq;
    pq.enqueue(100, 3);
```

```
    pq.enqueue(200, 1);
    pq.enqueue(300, 2);
    cout << "Dequeued element is: " << pq.dequeue() << "\n"; // Should
        remove element 200 (highest priority)
    return 0;
}
```

Listing 10.3: C++ implementation of a Priority Queue using an Array

10.4.4 Priority Queue Using Linked List (Sorted)

This implementation keeps the list sorted according to priority so that dequeue is $O(1)$ while insertion takes $O(n)$.

```
#include <iostream>
using namespace std;

class PNode {
public:
    int data;
    int priority;
    PNode* next;
};

class PriorityQueueLL {
private:
    PNode* head;
public:
    PriorityQueueLL() {
        head = nullptr;
    }
    bool isEmpty() {
        return (head == nullptr);
    }
    void enqueue(int data, int priority) {
        PNode* temp = new PNode();
        temp->data = data;
        temp->priority = priority;
        // If list is empty or new node has higher priority than the
            head
        if(head == nullptr || priority < head->priority) {
            temp->next = head;
            head = temp;
        } else {
            PNode* current = head;
            while(current->next != nullptr && current->next->priority <=
                priority) {
                current = current->next;
            }
```

```
            temp->next = current->next;
            current->next = temp;
        }
    }
    int dequeue() {
        if(isEmpty()) {
            cout << "Priority Queue Underflow\n";
            return -1;
        }
        PNode* temp = head;
        int data = temp->data;
        head = head->next;
        delete temp;
        return data;
    }
    int peek() {
        if(isEmpty()) {
            cout << "Priority Queue is empty\n";
            return -1;
        }
        return head->data;
    }
};

int main() {
    PriorityQueueLL pq;
    pq.enqueue(100, 3);
    pq.enqueue(200, 1);
    pq.enqueue(300, 2);
    cout << "Dequeued element is: " << pq.dequeue() << "\n"; // Should
        remove element 200 (highest priority)
    return 0;
}
```

Listing 10.4: C++ implementation of a Priority Queue using a Linked List

10.5 Example Scenario in Detail

Consider a standard queue scenario:

1. **Enqueue** 10 → Queue: [10]
2. **Enqueue** 20 → Queue: [10, 20]
3. **Enqueue** 30 → Queue: [10, 20, 30]
4. **Dequeue** → 10 is removed, Queue: [20, 30]

For a priority queue (assuming lower numbers indicate higher priority):

1. Enqueue (data: 100, priority: 3)
2. Enqueue (data: 200, priority: 1)
3. Enqueue (data: 300, priority: 2)

After insertion, the elements are ordered by priority as: [200 (priority 1), 300 (priority 2), 100 (priority 3)]. Dequeuing returns 200 first.

10.6 Conclusion

Queues are an essential data structure used in a variety of applications from scheduling to buffering. This chapter presented a detailed theoretical background, discussed time complexities, and illustrated the operations with diagrams. We also provided full C++ implementations for queues and priority queues using both array-based and linked list-based approaches. Understanding these implementations not only helps in grasping the abstract concept of FIFO and priority ordering but also shows practical trade-offs between different implementations.

Chapter 11

Skip List

11.1 Introduction

A **skip list** is a probabilistic data structure that allows fast search, insertion, and deletion operations within an ordered sequence of elements. It achieves performance comparable to balanced trees (average $O(\log n)$ time for operations) by maintaining multiple levels of linked lists. Higher levels "skip" many nodes, allowing rapid traversal of the list.

11.2 Definition and Theoretical Background

A skip list consists of multiple levels:

- The bottom level is an ordinary sorted linked list containing all elements.
- Each higher level acts as an "express lane" that skips several elements. Nodes appear in higher levels with a fixed probability p (commonly $p = 0.5$).

11.2.1 Key Operations

- **Search**: Start from the topmost level and traverse right until the next node's key exceeds the search key. Then drop down one level and continue. Average search time is $O(\log n)$.
- **Insertion**: Similar to search, find the correct position at each level and randomly decide how many levels the new node should appear in. Insert the node by updating pointers. Average time is $O(\log n)$.
- **Deletion**: Search for the node, then update pointers at all levels where the node appears. Average time is $O(\log n)$.

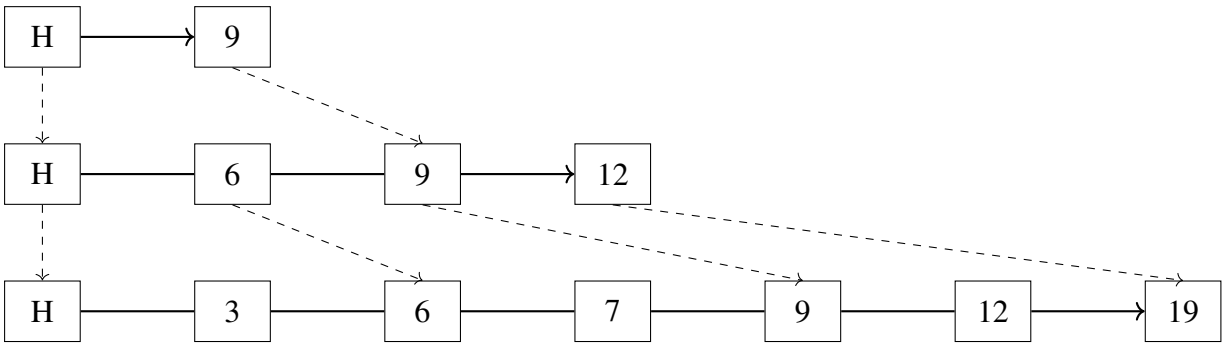
11.2.2 Time Complexity

- **Search**: $O(\log n)$ average
- **Insertion**: $O(\log n)$ average
- **Deletion**: $O(\log n)$ average

11.3 Diagrams of Skip List Operations

11.3.1 Skip List Structure Diagram

The diagram below illustrates a skip list with three levels. The bottom level (Level 0) contains all elements. Higher levels (Levels 1 and 2) contain a subset of nodes to enable faster traversal.



In the diagram:

- **H** stands for the header node.
- Dashed arrows indicate pointers connecting nodes between levels.

11.4 C++ Implementation of a Skip List

Below is a sample C++ implementation of a skip list that supports search, insertion, and deletion operations.

```
#include <iostream>
#include <cstdlib>
#include <climits>
#include <ctime>
using namespace std;

class Node {
public:
    int key;
    Node **forward;
    // Constructor: creates a node with given key and level
    Node(int key, int level) : key(key) {
        forward = new Node*[level + 1];
        for (int i = 0; i <= level; i++) {
            forward[i] = nullptr;
        }
    }
    ~Node() {
        delete [] forward;
    }
```

```
};

class SkipList {
    int maxLevel;
    float p;
    int level;
    Node *header;
public:
    SkipList(int maxLevel, float p) : maxLevel(maxLevel), p(p), level
        (0) {
        // Create header node with key -1 (a sentinel) and maximum
            level
        header = new Node(-1, maxLevel);
    }
    ~SkipList() {
        // Free nodes (not fully implemented for brevity)
        delete header;
    }

    // Randomly generate a level for node
    int randomLevel() {
        int lvl = 0;
        while (((float)rand() / RAND_MAX) < p && lvl < maxLevel)
            lvl++;
        return lvl;
    }

    // Insert a key into skip list
    void insertElement(int key) {
        Node *current = header;
        Node *update[maxLevel + 1];
        for (int i = 0; i <= maxLevel; i++) {
            update[i] = nullptr;
        }
        // Start from highest level and move downwards
        for (int i = level; i >= 0; i--) {
            while (current->forward[i] != nullptr && current->forward[i
                ]->key < key)
                current = current->forward[i];
            update[i] = current;
        }
        current = current->forward[0];
        // If key not found, insert new node
        if (current == nullptr || current->key != key) {
            int rlevel = randomLevel();
            if (rlevel > level) {
                for (int i = level + 1; i <= rlevel; i++) {
                    update[i] = header;
```

```
            }
            level = rlevel;
        }
        Node* n = new Node(key, rlevel);
        for (int i = 0; i <= rlevel; i++) {
            n->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = n;
        }
        cout << "Successfully inserted key " << key << "\n";
    }
}

// Delete an element from skip list
void deleteElement(int key) {
    Node *current = header;
    Node *update[maxLevel + 1];
    for (int i = 0; i <= maxLevel; i++) {
        update[i] = nullptr;
    }
    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i
            ]->key < key)
            current = current->forward[i];
        update[i] = current;
    }
    current = current->forward[0];
    if (current != nullptr && current->key == key) {
        for (int i = 0; i <= level; i++) {
            if (update[i]->forward[i] != current)
                break;
            update[i]->forward[i] = current->forward[i];
        }
        delete current;
        while (level > 0 && header->forward[level] == nullptr)
            level--;
        cout << "Successfully deleted key " << key << "\n";
    }
}

// Search for an element in the skip list
Node* searchElement(int key) {
    Node *current = header;
    for (int i = level; i >= 0; i--) {
        while (current->forward[i] != nullptr && current->forward[i
            ]->key < key)
            current = current->forward[i];
    }
    current = current->forward[0];
```

```
        if (current != nullptr && current->key == key)
            return current;
        return nullptr;
    }

    // Display the skip list levels
    void displayList() {
        cout << "\n***** Skip List *****" << "\n";
        for (int i = 0; i <= level; i++) {
            Node *node = header->forward[i];
            cout << "Level " << i << ": ";
            while (node != nullptr) {
                cout << node->key << " ";
                node = node->forward[i];
            }
            cout << "\n";
        }
    }
};

int main() {
    srand((unsigned)time(0));
    SkipList lst(3, 0.5);
    lst.insertElement(3);
    lst.insertElement(6);
    lst.insertElement(7);
    lst.insertElement(9);
    lst.insertElement(12);
    lst.insertElement(19);
    lst.displayList();

    int searchKey = 9;
    Node *res = lst.searchElement(searchKey);
    if(res != nullptr)
        cout << "Found key: " << searchKey << "\n";
    else
        cout << "Key " << searchKey << " not found\n";

    lst.deleteElement(3);
    lst.displayList();

    return 0;
}
```

Listing 11.1: C++ implementation of a Skip List

11.5 Example Scenario in Detail

Consider the following operations on an initially empty skip list:

1. **Insert** 3, 6, 7, 9, 12, 19 into the skip list.
2. The skip list will organize these keys into multiple levels where higher levels contain fewer nodes.
3. **Search** for key 9: Starting from the top level, the algorithm quickly narrows down to the correct node.
4. **Delete** key 3: After deletion, pointers at all levels are updated accordingly.

11.6 Conclusion

Skip lists offer an elegant probabilistic alternative to balanced trees, providing $O(\log n)$ average time for search, insertion, and deletion. This chapter presented the theoretical background, detailed operations, and diagrams illustrating the multi-level structure. The included C++ implementation demonstrates how a skip list can be built and manipulated, making it a powerful tool for ordered data storage and retrieval.

Part III

Non-linear Data Structures

Chapter 12

Tree Data Structures

12.1 Introduction

A **tree** is a hierarchical data structure that consists of nodes connected by edges. Trees are widely used in computer science to represent hierarchical relationships such as file systems, organizational structures, and decision processes. A **binary tree** is a tree in which each node has at most two children, commonly referred to as the left and right child. A **binary search tree (BST)** is a special kind of binary tree that maintains an order: for any node, all elements in its left subtree are less than or equal to the node's key and all elements in its right subtree are greater.

12.2 Definition, Terminology, and Formulas

12.2.1 Basic Definitions and Terminology

A tree T is defined as a pair (V, E) where:

- V is a set of nodes.
- E is a set of edges connecting the nodes.

Key terms include:

Root: The topmost node in a tree.

Parent: A node that has one or more children.

Child: A node that is a descendant of another node.

Leaf: A node with no children.

Subtree: A tree consisting of a node and its descendants.

Height: The length of the longest path from the root to a leaf. Formally, for a node v,

$$\text{height}(v) = \begin{cases} 0, & \text{if } v \text{ is a leaf,} \\ 1 + \max(\text{height}(v.\text{left}), \text{height}(v.\text{right})), & \text{otherwise.} \end{cases}$$

Depth: The length of the path from the root to the node.

12.2.2 Performance Formulas

For a balanced binary tree containing n nodes:

- **Search, Insertion, Deletion:** Average time complexity is $O(\log n)$.

In a degenerate (unbalanced) tree, these operations degrade to $O(n)$.

12.2.3 Summary Table of Tree Properties

Tree Type	Average Height	Search / Insertion / Deletion	Space Complexity
Binary Search Tree (Balanced)	$O(\log n)$	$O(\log n)$	$O(n)$
Binary Search Tree (Unbalanced)	$O(n)$	$O(n)$	$O(n)$
AVL Tree / Red-Black Tree	$O(\log n)$	$O(\log n)$	$O(n)$
Complete Binary Tree	$O(\log n)$	$O(\log n)$	$O(n)$

Table 12.1: Comparison of Tree Data Structures

12.3 Types of Tree Data Structures

Trees come in many forms, each with its own characteristics:

- **Binary Tree**: Each node has at most two children.
- **Binary Search Tree (BST)**: A binary tree that maintains sorted order.
- **Balanced Trees**: Such as AVL trees and Red-Black trees, which maintain $O(\log n)$ height.
- **Complete Binary Tree**: A binary tree in which all levels, except possibly the last, are completely filled.
- **Heap**: A specialized tree-based structure that satisfies the heap property (used for priority queues).
- **B-tree and B+ tree**: Used in databases and filesystems, designed to work well on disks.
- **Trie**: A tree used for storing associative arrays where the keys are usually strings.

12.4 Converting Arrays into Trees

A common example is converting an array into a **complete binary tree**. If we have an array A with n elements, we can represent it as a complete binary tree where:

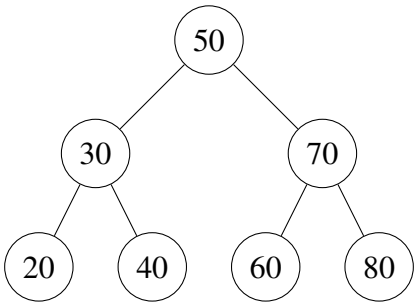
- The element at index i is the parent.
- The left child is at index $2i + 1$.
- The right child is at index $2i + 2$.

This representation is particularly efficient because it uses the array indices to simulate pointers, which is common in heap implementations.

12.5 Diagrams of Tree Structures and Traversals

12.5.1 Binary Tree Diagram

The diagram below shows a simple binary tree:



12.5.2 Traversal Order Example

For the tree above:

- **Preorder**: 50, 30, 20, 40, 70, 60, 80
- **Inorder**: 20, 30, 40, 50, 60, 70, 80
- **Postorder**: 20, 40, 30, 60, 80, 70, 50
- **Level Order**: 50, 30, 70, 20, 40, 60, 80

12.6 C++ Implementation of a Binary Search Tree

Below is a complete C++ implementation of a binary search tree (BST) including insertion, search, deletion, and all traversal methods.

```
#include <iostream>
#include <queue>
using namespace std;
```

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

class BST {
private:
    Node* root;

    Node* insert(Node* node, int key) {
        if (node == nullptr)
            return new Node(key);
        if (key < node->key)
            node->left = insert(node->left, key);
        else if (key > node->key)
            node->right = insert(node->right, key);
        return node;
    }

    Node* search(Node* node, int key) {
        if (node == nullptr || node->key == key)
            return node;
        if (key < node->key)
            return search(node->left, key);
        else
            return search(node->right, key);
    }

    // Find the minimum node in a subtree
    Node* minValueNode(Node* node) {
        Node* current = node;
        while (current && current->left != nullptr)
            current = current->left;
        return current;
    }

    Node* remove(Node* node, int key) {
        if (node == nullptr) return node;
        if (key < node->key)
            node->left = remove(node->left, key);
        else if (key > node->key)
            node->right = remove(node->right, key);
        else {
            // Node with only one child or no child
            if (node->left == nullptr) {
                Node* temp = node->right;
```

```
            delete node;
            return temp;
        } else if (node->right == nullptr) {
            Node* temp = node->left;
            delete node;
            return temp;
        }
        // Node with two children: Get the inorder successor (
            smallest in the right subtree)
        Node* temp = minValueNode(node->right);
        node->key = temp->key;
        node->right = remove(node->right, temp->key);
    }
    return node;
}

// Preorder Traversal (Root, Left, Right)
void preorder(Node* node) {
    if (node != nullptr) {
        cout << node->key << " ";
        preorder(node->left);
        preorder(node->right);
    }
}

// Inorder Traversal (Left, Root, Right)
void inorder(Node* node) {
    if (node != nullptr) {
        inorder(node->left);
        cout << node->key << " ";
        inorder(node->right);
    }
}

// Postorder Traversal (Left, Right, Root)
void postorder(Node* node) {
    if (node != nullptr) {
        postorder(node->left);
        postorder(node->right);
        cout << node->key << " ";
    }
}

public:
    BST() : root(nullptr) {}

    void insert(int key) {
        root = insert(root, key);
```

```
    }

    Node* search(int key) {
        return search(root, key);
    }

    void remove(int key) {
        root = remove(root, key);
    }

    void preorder() {
        preorder(root);
        cout << "\n";
    }

    void inorder() {
        inorder(root);
        cout << "\n";
    }

    void postorder() {
        postorder(root);
        cout << "\n";
    }

    // Level Order Traversal (Breadth-first)
    void levelOrder() {
        if (root == nullptr) return;
        queue<Node*> q;
        q.push(root);
        while (!q.empty()) {
            Node* current = q.front();
            q.pop();
            cout << current->key << " ";
            if (current->left != nullptr)
                q.push(current->left);
            if (current->right != nullptr)
                q.push(current->right);
        }
        cout << "\n";
    }
};

int main() {
    BST tree;
    // Insertion
    tree.insert(50);
    tree.insert(30);
```

```
    tree.insert(20);
    tree.insert(40);
    tree.insert(70);
    tree.insert(60);
    tree.insert(80);

    cout << "Inorder Traversal: ";
    tree.inorder();    // 20 30 40 50 60 70 80

    cout << "Preorder Traversal: ";
    tree.preorder();   // 50 30 20 40 70 60 80

    cout << "Postorder Traversal: ";
    tree.postorder();  // 20 40 30 60 80 70 50

    cout << "Level Order Traversal: ";
    tree.levelOrder(); // 50 30 70 20 40 60 80

    // Search for a key
    int searchKey = 40;
    if (tree.search(searchKey))
        cout << "Key " << searchKey << " found in the tree.\n";
    else
        cout << "Key " << searchKey << " not found in the tree.\n";

    // Deletion
    tree.remove(30);
    cout << "Inorder Traversal after deleting 30: ";
    tree.inorder();

    return 0;
}
```

Listing 12.1: C++ implementation of a Binary Search Tree

12.7 AVL Trees: Implementation and Rotations

AVL trees are a type of self-balancing binary search tree that ensures the difference in heights (balance factor) between the left and right subtrees for any node is at most one. This guarantees $O(\log n)$ time complexity for search, insertion, and deletion.

12.7.1 Node Structure and Balance Factor

For an AVL tree, each node is augmented with a **height** attribute. The **balance factor** is defined as:

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

A node is balanced if its balance factor is -1, 0, or 1.

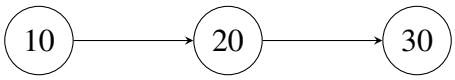
12.7.2 AVL Rotations

To restore balance after an insertion or deletion, AVL trees perform rotations:

- **Right Rotation**: Used for a left-left imbalance.
- **Left Rotation**: Used for a right-right imbalance.
- **Left-Right Rotation**: A left rotation followed by a right rotation, used for a left-right imbalance.
- **Right-Left Rotation**: A right rotation followed by a left rotation, used for a right-left imbalance.

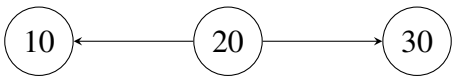
12.7.3 Diagram: Left Rotation Example

Before rotation (inserting keys causing a right-right imbalance):



Before rotation

After performing a left rotation at node 10, the tree becomes:



After left rotation

12.7.4 C++ Implementation of an AVL Tree

Below is a complete C++ implementation for an AVL tree that includes node structure, insertion with rebalancing (rotations), and a preorder traversal function for demonstration.

```
#include <iostream>
#include <algorithm>
using namespace std;

struct AVLNode {
    int key;
    int height;
    AVLNode *left, *right;
    AVLNode(int key) : key(key), height(1), left(nullptr), right(
        nullptr) {}
};

int height(AVLNode *node) {
```

```
    return node ? node->height : 0;
}

int getBalance(AVLNode *node) {
    return node ? height(node->left) - height(node->right) : 0;
}

AVLNode* rightRotate(AVLNode* y) {
    AVLNode *x = y->left;
    AVLNode *T2 = x->right;
    // Perform rotation
    x->right = y;
    y->left = T2;
    // Update heights
    y->height = max(height(y->left), height(y->right)) + 1;
    x->height = max(height(x->left), height(x->right)) + 1;
    return x;
}

AVLNode* leftRotate(AVLNode* x) {
    AVLNode *y = x->right;
    AVLNode *T2 = y->left;
    // Perform rotation
    y->left = x;
    x->right = T2;
    // Update heights
    x->height = max(height(x->left), height(x->right)) + 1;
    y->height = max(height(y->left), height(y->right)) + 1;
    return y;
}

AVLNode* insert(AVLNode* node, int key) {
    // Standard BST insertion
    if(!node)
       return new AVLNode(key);
    if(key < node->key)
       node->left = insert(node->left, key);
    else if(key > node->key)
       node->right = insert(node->right, key);
    else
       return node; // no duplicates allowed

    // Update the height of this ancestor node
    node->height = max(height(node->left), height(node->right)) + 1;
    int balance = getBalance(node);

    // Left Left Case
    if(balance > 1 && key < node->left->key)
```

```
        return rightRotate(node);
    // Right Right Case
    if(balance < -1 && key > node->right->key)
        return leftRotate(node);
    // Left Right Case
    if(balance > 1 && key > node->left->key) {
        node->left = leftRotate(node->left);
        return rightRotate(node);
    }
    // Right Left Case
    if(balance < -1 && key < node->right->key) {
        node->right = rightRotate(node->right);
        return leftRotate(node);
    }
    return node;
}

void preOrder(AVLNode *root) {
    if(root != nullptr) {
        cout << root->key << " ";
        preOrder(root->left);
        preOrder(root->right);
    }
}

int main() {
    AVLNode *root = nullptr;
    int arr[] = {10, 20, 30, 40, 50, 25};
    for (int i = 0; i < 6; i++)
        root = insert(root, arr[i]);
    cout << "Preorder traversal of the constructed AVL tree is: \n";
    preOrder(root);
    return 0;
}
```

Listing 12.2: C++ implementation of an AVL Tree

12.8 AVL Trees: Rotations

AVL trees are a type of self-balancing binary search tree that ensures the difference in heights (balance factor) between the left and right subtrees for any node is at most one. This guarantees $O(\log n)$ time complexity for search, insertion, and deletion.

12.8.1 Node Structure and Balance Factor

For an AVL tree, each node is augmented with a **height** attribute. The **balance factor** of a node is defined as:

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

A node is balanced if its balance factor is -1, 0, or 1.

12.8.2 AVL Rotations: Terminology and Diagrams

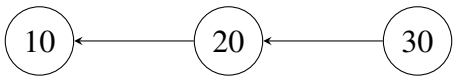
When an insertion or deletion causes a node's balance factor to fall outside of $[-1, 1]$, the AVL tree must be rebalanced using rotations. The key rotation types are:

- **Right Rotation**: Corrects a left-left imbalance.
- **Left Rotation**: Corrects a right-right imbalance.
- **Left-Right Rotation**: A combination; perform a left rotation on the left child, then a right rotation on the node.
- **Right-Left Rotation**: A combination; perform a right rotation on the right child, then a left rotation on the node.

Right Rotation Diagram

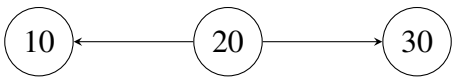
Right Rotation is used when a node's left subtree is heavy (i.e., a left-left imbalance). Consider the following subtree:

Before right rotation:



Before right rotation

After right rotation at node y:

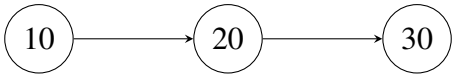


After right rotation

Left Rotation Diagram

Left Rotation is used when a node's right subtree is heavy (i.e., a right-right imbalance). Consider the following subtree:

Before left rotation:



Before left rotation

After left rotation at node x:

After left rotation

12.9 Conclusion

This chapter has provided an in-depth exploration of tree data structures. We began with the theoretical foundations and key terminology, introduced essential formulas, and presented a summary table of tree properties. We then discussed various types of trees—including binary trees, binary search trees, balanced trees, heaps, and tries—and explained how to convert an array into a complete binary tree.

Chapter 13

Graph Data Structures

13.1 Introduction

A **graph** is a data structure used to represent a set of objects (called **vertices** or **nodes**) and the relationships between them (called **edges**). Graphs are widely used to model many real-world problems such as social networks, transportation systems, and web page link structures. Depending on the application, graphs can be:

- **Directed** (where edges have a direction).
- **Undirected** (where edges are bidirectional).
- **Weighted** (if edges carry a cost, distance, or other metric).

13.2 Definition, Terminology, and Formulas

A graph G is defined as a pair (V, E), where:

- V is a non-empty set of vertices.
- E is a set of edges, where each edge is a pair (or ordered pair in directed graphs) of vertices.

13.2.1 Key Terminology

Vertex (Node): An individual object in the graph.

Edge: A connection between two vertices.

Adjacent: Two vertices that are directly connected by an edge.

Degree: For an undirected graph, the degree of a vertex is the number of edges incident on it. In directed graphs, we differentiate between **in-degree** and **out-degree**.

Path: A sequence of vertices connected by edges.

Cycle: A path that starts and ends at the same vertex without repeating an edge.

Connectivity: A graph is connected if there is a path between any two vertices.

13.2.2 Useful Formulas

- **Handshaking Lemma (Undirected):**

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

- **Directed Graph Degree Relationship:**

$$\sum_{v \in V} \text{in-degree}(v) = \sum_{v \in V} \text{out-degree}(v) = |E|$$

- **Path Length:** The length of a path is defined as the number of edges in the path.

13.3 Graph Representations

Graphs can be represented in several ways. The two primary methods are:

13.3.1 Adjacency Matrix

An adjacency matrix is a 2D array A of size $|V| \times |V|$ where:

$$A[i][j] = \begin{cases} 1 & \text{if there is an edge from vertex } i \text{ to vertex } j, \\ 0 & \text{otherwise.} \end{cases}$$

For weighted graphs, the entry may store the edge weight instead of 1 (with a special value such as ∞ or 0 indicating no edge).

13.3.2 Adjacency List

An adjacency list represents the graph as an array (or vector) of lists. The list at index i contains all vertices adjacent to vertex i. This representation is especially space-efficient for sparse graphs.

13.3.3 Edge List

An edge list is simply a list of all edges. Each edge is represented by a pair (or triple for weighted graphs) of vertices:

$$\text{Edge List } = \{(u,v) \mid u,v \in V\}$$

This representation is simple and is often used as an intermediate step in algorithms like Kruskal's for finding minimum spanning trees.

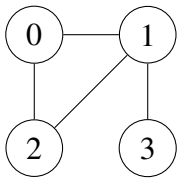
13.4 Types of Graphs and Detailed Diagrams

Graphs can be categorized based on the nature of their edges and vertices:

13.4.1 Undirected Graph

In an undirected graph, the edges do not have a direction. An edge between vertices u and v is represented as $\{u, v\}$.

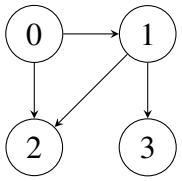
Diagram:



13.4.2 Directed Graph (Digraph)

In a directed graph, each edge has a direction. An edge from vertex u to vertex v is represented as (u, v).

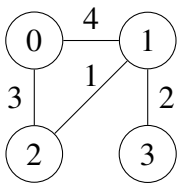
Diagram:



13.4.3 Weighted Graph

A weighted graph has weights (or costs) associated with each edge. These weights can represent distances, costs, or other metrics.

Diagram (Undirected Weighted Graph):



13.5 C++ Implementations of Graph Representations

Below are sample C++ implementations for the different graph representations.

13.5.1 Adjacency Matrix Implementation

```
#include <iostream>
#include <vector>
using namespace std;

class GraphMatrix {
private:
    int V; // number of vertices
```

```
    vector<vector<int>> adjMatrix;
public:
    GraphMatrix(int V) : V(V) {
        adjMatrix.resize(V, vector<int>(V, 0));
    }
    void addEdge(int u, int v, bool undirected = true, int weight = 1)
        {
        adjMatrix[u][v] = weight;
        if(undirected)
            adjMatrix[v][u] = weight;
    }
    void printMatrix() {
        for(int i = 0; i < V; i++) {
            for(int j = 0; j < V; j++)
                cout << adjMatrix[i][j] << " ";
            cout << "\n";
        }
    }
};

int main() {
    int V = 4;
    GraphMatrix graph(V);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    cout << "Adjacency Matrix:\n";
    graph.printMatrix();
    return 0;
}
```

Listing 13.1: C++ implementation using an Adjacency Matrix

13.5.2 Adjacency List Implementation

```
#include <iostream>
#include <vector>
using namespace std;

class GraphList {
private:
    int V;
    vector<vector<int>> adjList;
public:
    GraphList(int V) : V(V) {
        adjList.resize(V);
    }
```

```
    void addEdge(int u, int v, bool undirected = true) {
        adjList[u].push_back(v);
        if(undirected)
            adjList[v].push_back(u);
    }
    void printList() {
        for(int i = 0; i < V; i++) {
            cout << "Vertex " << i << ": ";
            for(auto v : adjList[i])
                cout << v << " ";
            cout << "\n";
        }
    }
};

int main() {
    int V = 4;
    GraphList graph(V);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    cout << "Adjacency List:\n";
    graph.printList();
    return 0;
}
```

Listing 13.2: C++ implementation using an Adjacency List

13.5.3 Edge List Representation

```
#include <iostream>
#include <vector>
using namespace std;

class GraphEdgeList {
private:
    int V;
    vector<pair<int, int>> edgeList;
public:
    GraphEdgeList(int V) : V(V) { }
    void addEdge(int u, int v) {
        edgeList.push_back({u, v});
    }
    void printEdges() {
        for(auto edge : edgeList)
            cout << edge.first << " - " << edge.second << "\n";
    }
```

```
};

int main() {
    int V = 4;
    GraphEdgeList graph(V);
    graph.addEdge(0, 1);
    graph.addEdge(0, 2);
    graph.addEdge(1, 2);
    graph.addEdge(1, 3);
    cout << "Edge List:\n";
    graph.printEdges();
    return 0;
}
```

Listing 13.3: C++ implementation using an Edge List

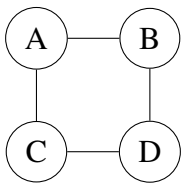
13.6 Graph Algorithms

In this section we discuss two fundamental graph traversal algorithms: Breadth-First Search (BFS) and Depth-First Search (DFS). We also include a discussion on spanning trees, which are subgraphs connecting all vertices without cycles.

13.6.1 Breadth-First Search (BFS)

BFS is a level-order traversal that visits nodes layer by layer. It uses a queue to keep track of the next vertex to visit.

BFS Diagram:



BFS Pseudocode:

```
BFS(Graph, start):
    create a queue Q
    mark start as visited and enqueue start
    while Q is not empty:
        v = Q.dequeue()
        for all neighbors w of v:
            if w is not visited:
                mark w as visited
                enqueue w
```

BFS C++ Implementation

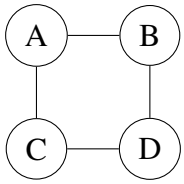
```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void BFS(const vector<vector<int>>& adjList, int start) {
    int V = adjList.size();
    vector<bool> visited(V, false);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
        int v = q.front();
        q.pop();
        cout << v << " ";

        for (int neighbor : adjList[v]) {
            if (!visited[neighbor]) {
                visited[neighbor] = true;
                q.push(neighbor);
            }
        }
    }
}

int main() {
    vector<vector<int>> adjList = {
        {1, 2},    // Neighbors of 0
        {0, 2, 3}, // Neighbors of 1
        {0, 1, 3}, // Neighbors of 2
        {1, 2}     // Neighbors of 3
    };
    cout << "BFS Traversal starting from vertex 0:\n";
    BFS(adjList, 0);
    return 0;
}
```

Listing 13.4: BFS in C++ using an Adjacency List

13.6.2 Depth-First Search (DFS)

DFS is a traversal technique that explores as far as possible along each branch before backtracking. It is implemented using recursion or a stack.

DFS Diagram:



DFS Pseudocode:

```
DFS(Graph, v):
    mark v as visited
    for all neighbors w of v:
        if w is not visited:
            DFS(Graph, w)
```

DFS C++ Implementation

```cpp
#include <iostream>
#include <vector>
using namespace std;

void DFSUtil(const vector<vector<int>>& adjList, int v, vector<bool>&
    visited) {
    visited[v] = true;
    cout << v << " ";

    for (int neighbor : adjList[v]) {
        if (!visited[neighbor]) {
            DFSUtil(adjList, neighbor, visited);
        }
    }
}

void DFS(const vector<vector<int>>& adjList, int start) {
    int V = adjList.size();
    vector<bool> visited(V, false);
    DFSUtil(adjList, start, visited);
}

int main() {
    vector<vector<int>> adjList = {
        {1, 2},
        {0, 2, 3},
        {0, 1, 3},
        {1, 2}
    };
    cout << "DFS Traversal starting from vertex 0:\n";
    DFS(adjList, 0);
    return 0;
}
```

Listing 13.5: DFS in C++ using an Adjacency List

13.7 Spanning Trees

13.7.1 Definition

A **spanning tree** of a connected, undirected graph is a subgraph that is a tree and connects all the vertices together. A graph can have multiple spanning trees.

13.7.2 Properties of Spanning Trees

- A spanning tree of a graph with n vertices has exactly $n-1$ edges.
- It is a minimal connected subgraph of the graph.
- There is no cycle in a spanning tree.
- Removing any edge from a spanning tree disconnects the graph.
- A connected graph with n vertices and $n-1$ edges is a tree.

13.7.3 Minimum Spanning Tree (MST)

A **minimum spanning tree** is a spanning tree whose sum of edge weights is the smallest among all possible spanning trees of the graph.

Applications:

- Network design (e.g., electrical grid, computer networks)
- Approximation algorithms
- Clustering in machine learning

Common Algorithms:

- **Kruskal's Algorithm**: Adds edges in increasing order of weight, skipping cycles.
- **Prim's Algorithm**: Grows the MST one vertex at a time from a starting node.

13.7.4 Maximum Spanning Tree

A **maximum spanning tree** is similar to an MST but maximizes the total edge weight. It is useful in certain optimization problems where higher weights are preferred (e.g., maximizing reliability or capacity).

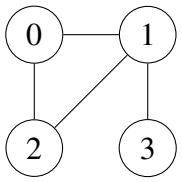
13.7.5 Comparison

Aspect	MST	MaxST
Objective	Minimize total weight	Maximize total weight
Used In	Cost-saving applications	Capacity or reliability maximization
Algorithm Change	Sort edges by increasing weight	Sort edges by decreasing weight

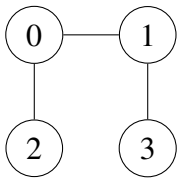
A **spanning tree** of a graph is a subgraph that includes all the vertices of the original graph, is connected, and has no cycles. In an undirected graph, any spanning tree will have exactly $V - 1$ edges.

Example and Diagram of a Spanning Tree

Given an undirected graph:



A possible spanning tree for the graph is:



Spanning Tree via BFS (C++ Implementation)

Below is an example of using BFS to generate a spanning tree from a connected undirected graph.

```cpp
#include <iostream>
#include <vector>
#include <queue>
using namespace std;

void spanningTreeBFS(const vector<vector<int>>& adjList, int start) {
    int V = adjList.size();
    vector<bool> visited(V, false);
    vector<int> parent(V, -1);
    queue<int> q;

    visited[start] = true;
    q.push(start);

    while (!q.empty()) {
```

```
        int u = q.front();
        q.pop();

        for (int v : adjList[u]) {
            if (!visited[v]) {
                visited[v] = true;
                parent[v] = u;
                q.push(v);
            }
        }
    }

    cout << "Spanning Tree (parent representation):\n";
    for (int i = 0; i < V; i++) {
        if (parent[i] != -1)
            cout << parent[i] << " - " << i << "\n";
    }
}

int main() {
    vector<vector<int>> adjList = {
        {1, 2},
        {0, 2, 3},
        {0, 1, 3},
        {1, 2}
    };
    spanningTreeBFS(adjList, 0);
    return 0;
}
```

Listing 13.6: Spanning Tree using BFS in C++

13.8 Diagrams for Graph Representations

In this section, we provide visual diagrams for the two primary representations of graphs: the adjacency matrix and the adjacency list. We will consider a sample graph with vertices $0, 1, 2, 3$ and edges $\{0,1\}$, $\{0,2\}$, $\{1,2\}$, and $\{1,3\}$.

13.8.1 Adjacency Matrix Diagram

The adjacency matrix for the sample graph is a 4×4 matrix where the rows and columns correspond to the vertices. A value of 1 indicates the presence of an edge between the corresponding vertices, and a value of 0 indicates no edge.

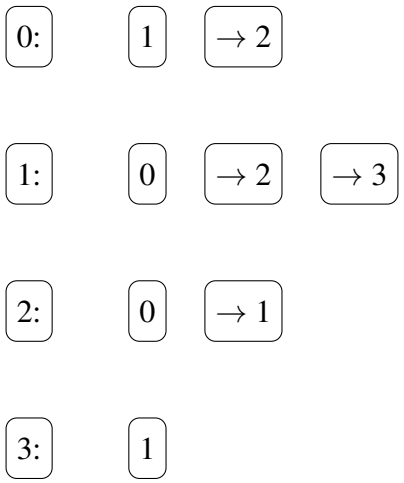
	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	0
3	0	1	0	0

13.8.2 Adjacency List Diagram

The adjacency list represents the graph by listing each vertex followed by the vertices adjacent to it. For our sample graph, the adjacency list is as follows:

$$
\begin{aligned}
0: &\quad 1 \to 2 \\
1: &\quad 0 \to 2 \to 3 \\
2: &\quad 0 \to 1 \\
3: &\quad 1
\end{aligned}
$$

The diagram below visually represents this structure:



These diagrams help in visualizing the structural differences between the two common graph representations. The matrix provides a compact, fixed-size representation (especially useful for dense graphs), while the list provides a more flexible and space-efficient representation (especially useful for sparse graphs).

13.9 Conclusion

This chapter has provided an in-depth exploration of graph data structures and algorithms. We began with the theoretical foundations, key terminology, and useful formulas.

Part IV

Searching and Sorting

Chapter 14

Searching Algorithms

This chapter explains fundamental searching algorithms with step-by-step examples, complete C++ implementations using the `listings` package, and diagrams illustrating the search process.

14.1 Linear Search

Linear Search scans each element of the array sequentially until the target value is found or the end of the array is reached.

14.1.1 Explanation

Given an array *arr* of size *n* and a target value, the algorithm starts at the first element and compares each element with the target. If a match is found, the index is returned; otherwise, it continues until all elements are checked.

14.1.2 C++ Code Example

```
#include <bits/stdc++.h>
using namespace std;

int linearSearch(const vector<int>& arr, int target) {
    for (int i = 0; i < arr.size(); i++) {
        if (arr[i] == target)
            return i; // Target found at index i
    }
    return -1; // Target not found
}
int main() {
    vector<int> arr = {3, 8, 2, 5, 9, 1};
    int target = 5;
    int index = linearSearch(arr, target);
    if (index != -1)
        cout << "Target " << target << " at index " << index << "\n";
    else
```

```
        cout << "Target " << target << " not found." << "\n";

    return 0;
}
```

Listing 14.1: Linear Search Implementation

14.1.3 Diagram: Linear Search Process

The following diagram shows how Linear Search traverses the array sequentially.

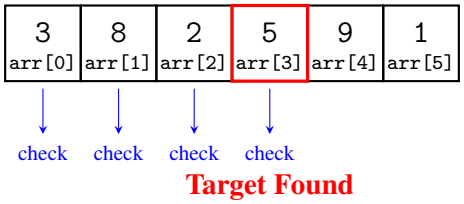


Figure 14.1: Linear Search: Sequentially checking each element

14.2 Binary Search

Binary Search is a highly efficient algorithm for searching in a sorted array. It repeatedly divides the search interval in half.

14.2.1 Explanation

Given a sorted array *arr* of size n and a target value, Binary Search works as follows:

1. Set two pointers, low $= 0$ and high $= n-1$.
2. Find the middle index: mid $= \lfloor (\text{low} + \text{high})/2 \rfloor$.
3. If $arr[\text{mid}]$ equals the target, return mid.
4. If the target is less than $arr[\text{mid}]$, repeat the search in the left subarray.
5. Otherwise, repeat the search in the right subarray.

14.2.2 C++ Code Example

```
#include <bits/stdc++.h>
using namespace std;

int binarySearch(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid; // Target found
```

```
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1; // Target not found
}
int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11};
    int target = 7;
    int index = binarySearch(arr, target);
    if (index != -1)
        cout << "Target " << target << " at index " << index << "\n";
    else
        cout << "Target " << target << " not found." << "\n";
    return 0;
}
```

Listing 14.2: Binary Search Implementation

14.2.3 Diagram: Binary Search Process

The following diagram illustrates how Binary Search halves the search interval at each step.

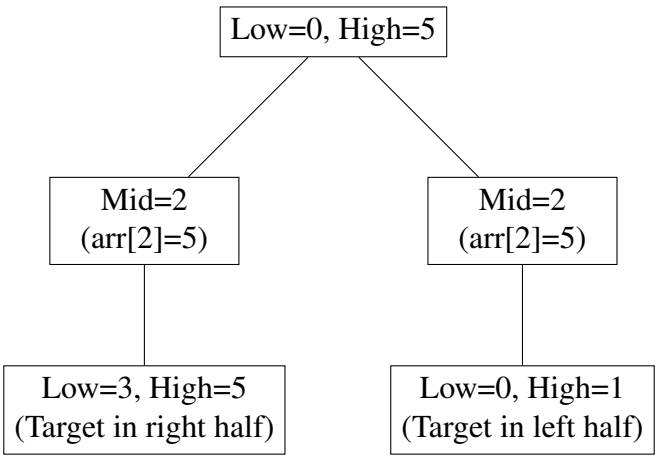


Figure 14.2: Binary Search: Halving the search interval

14.2.4 Step-by-Step Example of Binary Search

Consider a sorted array: $[1,3,5,7,9,11]$ and a target value of 7.

1. Initialize: low $= 0$, high $= 5$. Compute mid $= 2$. $arr[2] = 5$. Since $5 < 7$, move to the right half.
2. Update: low $= 3$, high $= 5$. Compute mid $= 4$. $arr[4] = 9$. Since $9 > 7$, move to the left half.
3. Update: low $= 3$, high $= 3$. Compute mid $= 3$. $arr[3] = 7$. Target found at index 3.

14.3 Exponential Search

Exponential Search is useful for unbounded or large sorted arrays. It first finds a range where the target may reside by repeatedly doubling the index, then performs a Binary Search in that range.

14.3.1 Explanation

Given a sorted array arr of size n and a target value:

1. If $arr[0]$ is the target, return index 0.
2. Otherwise, initialize index $i = 1$ and double i until $arr[i] >$ target or $i \geq n$.
3. Perform Binary Search on the subarray from $i/2$ to $\min(i, n-1)$.

14.3.2 C++ Code Example

```cpp
#include <bits/stdc++.h>
using namespace std;

int binarySearch(const vector<int>& arr, int low, int high, int target
    ) {
    while (low <= high) {
        int mid = low + (high - low) / 2;
        if (arr[mid] == target)
            return mid;
        else if (arr[mid] < target)
            low = mid + 1;
        else
            high = mid - 1;
    }
    return -1;
}

int exponentialSearch(const vector<int>& arr, int target) {
    int n = arr.size();
    if (n == 0) return -1;
    if (arr[0] == target) return 0;
    int i = 1;
    while (i < n && arr[i] <= target)
        i *= 2;
    return binarySearch(arr, i / 2, min(i, n - 1), target);
}
int main() {
    vector<int> arr = {1, 3, 5, 7, 9, 11, 13, 15, 17, 19};
    int target = 15;
    int index = exponentialSearch(arr, target);
    if (index != -1)
```

```
        cout << "Target " << target << " at index " << index << "\n";
    else
        cout << "Target " << target << " not found." << "\n";
    return 0;
}
```

Listing 14.3: Exponential Search Implementation

14.3.3 Diagram: Exponential Search Process

The diagram below shows how the algorithm doubles the index until the range is found, then performs Binary Search.

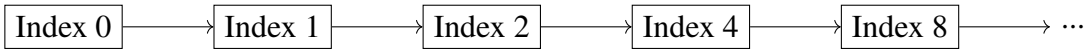


Figure 14.3: Exponential Search: Doubling indices to find the search range

14.4 Interpolation Search

Interpolation Search improves upon Binary Search for uniformly distributed arrays by estimating the position of the target using the values at the endpoints.

14.4.1 Explanation

For a sorted array *arr* of size n and target value:

$$\text{pos} = \text{low} + \frac{(target - arr[low]) \times (high - low)}{arr[high] - arr[low]}$$

This position is used to probe the array. If the target is found at pos, the index is returned; otherwise, the search continues in the appropriate subarray.

14.4.2 C++ Code Example

```
#include <bits/stdc++.h>
using namespace std;

int interpolationSearch(const vector<int>& arr, int target) {
    int low = 0, high = arr.size() - 1;
    while (low <= high && target >= arr[low] && target <= arr[high]) {
        if (low == high) {
            if (arr[low] == target) return low;
            return -1;
        }
        int pos = low + ((double)(high - low) / (arr[high] - arr[low]))
            * (target - arr[low]);

        if (arr[pos] == target)
```

```
            return pos;
        if (arr[pos] < target)
            low = pos + 1;
        else
            high = pos - 1;
    }
    return -1;
}

int main() {
    vector<int> arr = {10, 20, 30, 40, 50, 60, 70, 80, 90};
    int target = 70;
    int index = interpolationSearch(arr, target);

    if (index != -1)
        cout << "Target " << target << " at index " << index << "\n";
    else
        cout << "Target " << target << " not found." << "\n";

    return 0;
}
```

Listing 14.4: Interpolation Search Implementation

Time Complexity Analysis:

- **Average Case:** $O(\log \log n)$ for uniformly distributed data.
- **Worst Case:** $O(n)$ when the data is not uniformly distributed.

14.5 Conclusion

In this chapter, we explored several searching algorithms:

- **Linear Search**: Simple and works on unsorted data ($O(n)$).
- **Binary Search**: Efficient on sorted arrays ($O(\log n)$).
- **Exponential Search**: Finds the range for Binary Search in unbounded arrays ($O(\log n)$).
- **Interpolation Search**: Optimized for uniformly distributed data ($O(\log \log n)$ on average).

Each algorithm has its strengths and is applicable to different scenarios. Understanding these algorithms and their performance characteristics is crucial for selecting the most appropriate method for your needs.

Chapter 15

Sorting Algorithms

This chapter provides detailed C++ implementations of several important sorting algorithms. Each algorithm is explained with code examples using the `listings` package.

15.1 Bubble Sort

Bubble Sort repeatedly compares adjacent elements and swaps them if they are in the wrong order. It is simple but inefficient for large datasets.

```
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 0; i < n - 1; i++) {
        bool swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j+1]) {
                swap(arr[j], arr[j+1]);
                swapped = true;
            }
        }
        if (!swapped)
            break; // Array is sorted
    }
}
```

Listing 15.1: Bubble Sort Implementation

15.2 Insertion Sort

Insertion Sort builds the sorted array one element at a time by comparing each new element with the already sorted elements.

```
void insertionSort(vector<int>& arr) {
    int n = arr.size();
    for (int i = 1; i < n; i++) {
        int key = arr[i];
        int j = i - 1;
```

```
        while (j >= 0 && arr[j] > key) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = key;
    }
}
```

Listing 15.2: Insertion Sort Implementation

15.3 Selection Sort

Selection Sort repeatedly finds the minimum element from the unsorted portion and swaps it with the first unsorted element.

```
void selectionSort(vector<int>& arr) {
    int n = arr.size();
    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;
        }
        swap(arr[i], arr[minIdx]);
    }
}
```

Listing 15.3: Selection Sort Implementation

15.4 Counting Sort

Counting Sort works efficiently when the range of input data (k) is not significantly greater than the number of elements (n). It counts the occurrences of each value and then reconstructs the sorted array.

```
void countingSort(vector<int>& arr) {
    int n = arr.size();
    int maxVal = *max_element(arr.begin(), arr.end());
    vector<int> count(maxVal + 1, 0);
    for (int i = 0; i < n; i++) {
        count[arr[i]]++;
    }
    int index = 0;
    for (int i = 0; i <= maxVal; i++) {
        while (count[i]-- > 0) {
            arr[index++] = i;
        }
    }
```

```
}
```

Listing 15.4: Counting Sort Implementation

Time Complexity Analysis:

- Counting frequency: $O(n)$
- Populating the sorted array: $O(k)$

Overall, the expected time complexity is $O(n+k)$, with space complexity $O(k)$.

15.5 Radix Sort

Radix Sort is a non-comparative sorting algorithm that sorts numbers digit by digit using Counting Sort as a subroutine.

```
int getMax(const vector<int>& arr) {
    return *max_element(arr.begin(), arr.end());
}
void countingSortByDigit(vector<int>& arr, int exp) {
    int n = arr.size();
    vector<int> output(n);
    vector<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] = arr[i];
        count[(arr[i] / exp) % 10]--;
    }
    for (int i = 0; i < n; i++)
        arr[i] = output[i];
}
void radixSort(vector<int>& arr) {
    int maxVal = getMax(arr);
    for (int exp = 1; maxVal / exp > 0; exp *= 10)
        countingSortByDigit(arr, exp);
}
```

Listing 15.5: Radix Sort Implementation

Time Complexity Analysis:

$O(d(n+b))$ where d is the number of digits and b is the base (typically 10).

For decimal numbers, this is often written as $O(n \log_{10} k)$.

15.6 Timsort

Timsort is a hybrid sorting algorithm derived from merge sort and insertion sort. It is used in Python's sort and Java's Arrays.sort() for objects. Below is a simplified version in C++.

```
const int RUN = 32;
void insertionSort(vector<int>& arr, int left, int right) {
    for (int i = left + 1; i <= right; i++) {
        int temp = arr[i];
        int j = i - 1;
        while (j >= left && arr[j] > temp) {
            arr[j+1] = arr[j];
            j--;
        }
        arr[j+1] = temp;
    }
}
void merge(vector<int>& arr, int l, int m, int r) {
    int len1 = m - l + 1, len2 = r - m;
    vector<int> leftArr(len1), rightArr(len2);
    for (int i = 0; i < len1; i++) leftArr[i] = arr[l + i];
    for (int i = 0; i < len2; i++) rightArr[i] = arr[m + 1 + i];
    int i = 0, j = 0, k = l;
    while (i < len1 && j < len2) {
        if (leftArr[i] <= rightArr[j]) arr[k++] = leftArr[i++];
        else arr[k++] = rightArr[j++];
    }
    while (i < len1) arr[k++] = leftArr[i++];
    while (j < len2) arr[k++] = rightArr[j++];
}
void timSort(vector<int>& arr, int n) {
    for (int i = 0; i < n; i += RUN) {
        insertionSort(arr, i, min(i + RUN - 1, n - 1));
    }
    for (int size = RUN; size < n; size *= 2) {
        for (int left = 0; left < n; left += 2 * size) {
            int mid = left + size - 1;
            int right = min(left + 2 * size - 1, n - 1);
            if (mid < right) merge(arr, left, mid, right);
        }
    }
}
```

Listing 15.6: Timsort Implementation (Simplified)

Time Complexity Analysis:

- Best Case (nearly sorted): $O(n)$
- Worst Case: $O(n \log n)$

- Timsort adapts to the natural order in the data.

15.7 Merge Sort

Merge Sort is a classic divide-and-conquer algorithm. It divides an array into two halves, recursively sorts each half, and then merges the two sorted halves.

15.7.1 Merge Sort Implementation in C++

```
#include <bits/stdc++.h>
using namespace std;

void merge(vector<int>& arr, int left, int mid, int right) {
    int n1 = mid - left + 1;
    int n2 = right - mid;
    vector<int> L(n1), R(n2);
    for (int i = 0; i < n1; i++)
        L[i] = arr[left + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[mid + 1 + j];
    int i = 0, j = 0, k = left;
    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++];
}
void mergeSort(vector<int>& arr, int left, int right) {
    if (left < right) {
        int mid = left + (right - left) / 2;
        mergeSort(arr, left, mid);
        mergeSort(arr, mid + 1, right);
        merge(arr, left, mid, right);
    }
}
```

Listing 15.7: Merge Sort Implementation

15.7.2 Example Usage

```
int main() {
    vector<int> arr = {38, 27, 43, 3, 9, 82, 10};
```

```
    int n = arr.size();

    mergeSort(arr, 0, n - 1);

    cout << "Sorted array: ";
    for (int x : arr)
        cout << x << " ";
    cout << endl;

    return 0;
}
```

Listing 15.8: Using Merge Sort

15.7.3 Time Complexity Analysis

Merge Sort divides the array into halves and then merges them. Its recurrence relation is:

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n)$$

which solves to:

$$T(n) = O(n \log n)$$

Thus, Merge Sort has a time complexity of $O(n \log n)$ in all cases.

Space Complexity: Merge Sort requires additional space for temporary arrays during merging:

$$O(n)$$

Summary

Each sorting algorithm has its trade-offs in terms of time and space complexity:

- **Bubble Sort**: $O(n^2)$ worst-case, very simple but inefficient.
- **Insertion Sort**: $O(n^2)$ worst-case, $O(n)$ best-case, efficient for nearly sorted data.
- **Selection Sort**: $O(n^2)$ consistently, few swaps.
- **Counting Sort**: $O(n+k)$ time, where k is the range of the input.
- **Radix Sort**: $O(n \log_{10} k)$ for decimal numbers.
- **Timsort**: Hybrid algorithm with $O(n)$ best-case and $O(n \log n)$ worst-case.

Chapter 16

DSA Coding Questions

Top Interview Topics by Company

- **Google**: Trees, Graphs, DP, Backtracking, Tries, System Design
- **Amazon**: Arrays, Strings, Hashing, Sliding Window, Greedy
- **Microsoft**: DP, Graphs, Trees, Bit Manipulation
- **Adobe**: Heap, Stack, Recursion, Trees
- **Flipkart**: Greedy, Hashing, Priority Queue, Segment Tree
- **Meta (Facebook)**: BFS/DFS, Backtracking, Sliding Window, Graphs
- **Netflix**: Dynamic Programming, Interval Problems, System Design
- **Uber**: Priority Queue, Shortest Path, Greedy
- **Bloomberg**: Hash Maps, Stack, Graphs
- **TCS/Wipro/Infosys**: Basics of Arrays, Searching, Sorting, OOP, Recursion

Core DSA Patterns

- Sliding Window, Two Pointers
- Fast and Slow Pointers, Binary Search on Answer
- BFS / DFS / Backtracking
- Recursion + Memoization
- Prefix Sum, Hashing, Greedy
- Stack (Span/NGE/Histograms)

Arrays and Strings

- **[Easy]** Reverse an array
- **[Easy]** Find max and min in array
- **[Easy]** Move all zeros to the end
- **[Medium]** Union and Intersection of two arrays *(Amazon)*
- **[Medium]** Kadane's Algorithm – Max Subarray Sum *(Google)*
- **[Medium]** Longest substring without repeating characters *(Meta)*
- **[Medium]** Anagram check using hashing *(Bloomberg)*
- **[Hard]** Z-algorithm for pattern matching *(Adobe)*

Searching and Sorting

- Linear Search / Binary Search
- Search in Rotated Sorted Array *(Microsoft)*
- Floor and Ceil of a number *(Flipkart)*
- Merge Sort, Quick Sort
- Count Inversions in array *(Amazon)*

Two Pointers / Sliding Window

- Maximum sum subarray of size K
- Longest substring with at most K distinct characters
- Minimum window substring *(Amazon)*
- Trapping Rain Water *(Google)*

Linked Lists

- Reverse a linked list
- Detect loop using Floyd's cycle
- Merge 2 sorted linked lists
- Check if LL is palindrome *(Microsoft)*
- Clone LL with random pointer *(Amazon)*

Stack and Queue

- Valid Parentheses Checker
- Next Greater Element *(Adobe)*
- Min Stack Design *(Google)*
- Largest Rectangle in Histogram *(Facebook)*
- LRU Cache Implementation

Trees and BST

- Traversals: Inorder, Preorder, Postorder
- Diameter of Binary Tree *(Google)*
- Construct Tree from Inorder and Preorder
- Lowest Common Ancestor (LCA)
- BST Insertion, Deletion, Search
- Kth Smallest/Largest in BST

Graphs

- BFS and DFS Traversal
- Cycle Detection in Directed / Undirected Graph
- Topological Sort
- Dijkstra's Algorithm, Bellman-Ford
- Number of Islands *(Amazon)*
- Clone a Graph

Recursion and Backtracking

- Subsets / Permutations / Combinations
- N-Queens Problem *(Facebook)*
- Sudoku Solver
- Word Search
- Generate Balanced Parentheses

Greedy Algorithms

- Activity Selection
- Fractional Knapsack
- Job Scheduling with Deadlines *(Flipkart)*
- Minimum Coins for Change
- Gas Station Circle

Heap / Priority Queue

- Heapify, Min/Max Heap
- Kth Largest Element *(Amazon)*
- Merge K Sorted Lists
- Median of a Stream
- Reorganize String using Heap

Dynamic Programming (DP)

- Fibonacci (Memoization)
- 0/1 Knapsack
- Longest Common Subsequence (LCS)
- Longest Increasing Subsequence (LIS)
- Coin Change
- Edit Distance
- Palindromic Substrings Count *(Amazon)*

Chapter 17

C++ Template

17.1 Basic Template

This is the basic template that anyone can use to write the code in c++.

```
#include <bits/stdc++.h>

using namespace std;

void solve(){
    //Code Here....
}
void init_code(){
    ios_base :: sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
}
int main(){
    init_code();
    solve();
    return 0;
}
```

Listing 17.1: Main C++ Template

17.2 Debugging Template

```
template < class c > struct rge {
    c b, e;
};
template < class c > rge<c> range(c i, c j){
    return rge<c>{i, j};
}
template < class c > auto dud(c* x) -> decltype(cerr << *x, 0);
template < class c > char dud(...);

struct debug {
    ~debug() { cerr << endl; }
    template < class c > typename enable_if<sizeof dud<c>(0) != 1,
       debug&>::type operator<<(c i) {
        cerr << boolalpha << i;
        return * this;
    }
    template < class c > typename enable_if<sizeof dud<c>(0) == 1,
       debug&>::type operator<<(c i) {
        return * this << range(begin(i), end(i));
    }
    template < class c, class b > debug & operator << (pair < b, c > d)
        {
        return * this << "(" << d.first << ", " << d.second << ")";
    }
    template < class c > debug & operator <<(rge<c> d) {
        *this << "[";
        for (auto it = d.b; it != d.e; ++it)
            *this << ", " + 2 * (it == d.b) << *it;
        return * this << "]";
    }
};
#define imie(...) " [" << #__VA_ARGS__ " : " << (__VA_ARGS__) << "]"
```

Listing 17.2: Debugging Template

17.3 Combined Template

Here is a combined template for development that includes both the main structure and debugging utilities.

```
/*
*
*    Author : Girish Kumar Goyal.
*
*/

#include <bits/stdc++.h>

using namespace std;

template < class c > struct rge {
    c b, e;
};
template < class c > rge<c> range(c i, c j){
    return rge<c>{i, j};
}
template < class c > auto dud(c* x) -> decltype(cerr << *x, 0);
template < class c > char dud(...);
struct debug {
    ~debug() { cerr << endl; }
    template < class c > typename enable_if<sizeof dud<c>(0) != 1,
       debug&>::type operator<<(c i) {
        cerr << boolalpha << i;
        return * this;
    }
    template < class c > typename enable_if<sizeof dud<c>(0) == 1,
       debug&>::type operator<<(c i) {
        return * this << range(begin(i), end(i));
    }
    template < class c, class b > debug & operator << (pair < b, c > d)
         {
        return * this << "(" << d.first << ", " << d.second << ")";
    }
    template < class c > debug & operator <<(rge<c> d) {
        *this << "[";
        for (auto it = d.b; it != d.e; ++it)
            *this << ", " + 2 * (it == d.b) << *it;
        return * this << "]";
    }
};
#define imie(...) " [" << #__VA_ARGS__ " : " << (__VA_ARGS__) << "]"

void solve(){
    //Code Here....
```

```
}
void init_code(){
    ios_base :: sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
}
int main(){
    init_code();
    solve();
    return 0;
}
```

Listing 17.3: Combined Template

17.4 Using the Debugging Template

To use the debugging template, you can print variables or expressions during development by using the ‘debug()‘ object in combination with the ‘imie(...)‘ macro. This will output both the name of the variable and its value, which is extremely useful for tracing bugs.

Syntax:

```
debug() << imie(variable1) << imie(variable2) << imie(expression);
```

Example:

Here’s a complete example demonstrating the use of ‘debug()‘ and ‘imie(...)‘:

```
void solve() {
    int a = 42;
    string s = "hello";
    vector<int> v = {1, 2, 3};

    debug() << imie(a) << imie(s) << imie(v);
}

void init_code(){
    ios_base :: sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
}

int main(){
    init_code();
    solve();
    return 0;
}
```

Listing 17.4: Debugging Example

17.5 C++ program time calculation

Example:

```cpp
#include <bits/stdc++.h>

using namespace std;

void solve() {
    // Code here....
}

void init_code(){
    ios_base :: sync_with_stdio(false);
    cin.tie(nullptr);
    cout.tie(nullptr);
}

int main(){
    init_code();
    clock_t start, end;
    double cpu_time;
    start = clock();
    solve();
    cpu_time = ((double)(end - start)) / CLOCKS_PER_SEC;
    cout << cpu_time << "\n";
    return 0;
}
```

Listing 17.5: CPP Code time Example

17.6 Test case generator program in C++

Example:

Test case generator in c++ basically the code to generate random testcases like this.

```
2
5
1 2 3 4 5
4
2 3 1 43
```

```cpp
#include<bits/stdc++.h>

using namespace std;

int naxTest = 20;
int naxArraySize = 50;
int naxValue = 5000;

vector<int> generateTestCase(int size, int maxValue) {
    vector<int> testCase(size);
    for (int i = 0; i < size; i++) {
        testCase[i] = rand() % maxValue + 1;
    }
    return testCase;
}

int main() {
    srand(time(0));
    int numTestCases = rand() % naxTest + 1;
    int maxArraySize = rand() % naxArraySize + 1;
    int maxValue = rand() % naxValue + 1;
    cout << numTestCases << "\n";
    for (int i = 0; i < numTestCases; i++) {
        int arraySize = rand() % maxArraySize + 1;
        cout << arraySize << "\n";
        vector<int> testCase = generateTestCase(arraySize, maxValue);
        // sort(testCase.begin(), testCase.end());
        for (int num : testCase) {
            cout << num << " ";
        }
        cout << "\n";
    }
    return 0;
}
```

Listing 17.6: Test case generator

17.7 Script for stress testing of c++ code

Syntax:

```
./stress.sh
```

Example:

The following script automates compilation and stress testing of a C++ solution against generated test cases:

```
#!/bin/bash

black=$(tput setaf 0)
red=$(tput setaf 1)
green=$(tput setaf 2)
yellow=$(tput setaf 3)
blue=$(tput setaf 4)
magenta=$(tput setaf 5)
cyan=$(tput setaf 6)
white=$(tput setaf 7)
bold=$(tput bold)
reset=$(tput sgr0)
CPP_VERSION="c++17"
COMPILE_FLAGS="-Wall -Wextra -O2"
TEST_GEN_FILE="./stress_tester/test_gen.cpp"
MAIN_FILE="sol.cpp"
INPUT_FILE="in"
MAX_TESTS=5
TIME_LIMIT=2 # seconds per test case

usage() {
    echo -e "${bold}${cyan}Usage:${reset} $(basename "$0") [-t <num_tests>]"
    exit 1
}
while getopts "t:h" opt; do
    case $opt in
        t)
            MAX_TESTS="$OPTARG"
            ;;
        h)
            usage
            ;;
        *)
            usage
            ;;
    esac
done

log_info() {
    echo -e "${bold}${blue}[INFO]${reset} $1"
}
log_success() {
```

```
    echo -e "${bold}${green}[SUCCESS]${reset} $1"
}
log_error() {
    echo -e "${bold}${red}[ERROR]${reset} $1"
}
log_warning() {
    echo -e "${bold}${yellow}[WARNING]${reset} $1"
}
check_files() {
    echo ""
    echo "------------------------------------------------------------"
    local missing=0
    if [ ! -f "$MAIN_FILE" ]; then
        log_error "Main solution file ${yellow}$MAIN_FILE${reset} not found!"
        missing=1
    fi
    if [ ! -f "$TEST_GEN_FILE" ]; then
        log_error "Test case generator file ${yellow}$TEST_GEN_FILE${reset}
            not found!"
        missing=1
    fi
    if [ $missing -eq 1 ]; then
        exit 1
    fi
}
compile_file() {
    local src_file="$1"
    local exe_file="$2"
    local extra_flags="$3"
    log_info "Compiling ${yellow}$src_file${reset}..."
    local start_ns=$(date +%s%N)
    compile_output=$(g++ -std="$CPP_VERSION" $COMPILE_FLAGS $extra_flags "
        $src_file" -o "$exe_file" 2>&1)
    if [ $? -ne 0 ]; then
        log_error "Compilation failed for ${yellow}$src_file${reset}."
        echo -e "${red}Compiler output:${reset}"
        echo "$compile_output"
        exit 1
    fi
    local end_ns=$(date +%s%N)
    local compile_time_ns=$((end_ns - start_ns))
    local compile_time_ms=$(echo "scale=3; $compile_time_ns/1000000" | bc)
    log_success "Compiled ${yellow}$src_file${reset} to ${magenta}$exe_file${
        reset} in ${cyan}${compile_time_ms} ms${reset}."
}
stress_test() {
    local accepted=0
    local failed=0
    local total_test_time_ns=0
    echo -e "\n${bold}${red}Starting stress testing with ${yellow}$MAX_TESTS$
        {red} test cases...${reset}"
```

```
    for (( i=1; i<=MAX_TESTS; i++ )); do
        echo -e "\n${bold}${blue}======= Test case #$i =======${reset}"
        ./generator > "$INPUT_FILE"
        echo -e "${bold}${magenta}[Input]:${reset}"
        cat "$INPUT_FILE"
        echo ""
        local start_ns=$(date +%s%N)
        output=$(timeout $TIME_LIMIT ./sol < "$INPUT_FILE" 2>&1)
        local ret_code=$?
        local end_ns=$(date +%s%N)
        local elapsed_ns=$((end_ns - start_ns))
        total_test_time_ns=$((total_test_time_ns + elapsed_ns))
        local elapsed_ms=$(echo "scale=3; $elapsed_ns/1000000" | bc)
        echo -e "${bold}${cyan}[Output]:${reset}"
        echo -e "$output"
        if [ $ret_code -eq 0 ]; then
            echo -e "${bold}${green}Test case #$i: Accepted in ${blue}${
                elapsed_ms} ms${reset}"
            ((accepted++))
        elif [ $ret_code -eq 124 ]; then
            echo -e "${bold}${red}Test case #$i: Time Limit Exceeded (TLE)
                after ${yellow}${elapsed_ms} ms${reset}"
            ((failed++))
        elif [ $ret_code -eq 137 ]; then
            echo -e "${bold}${red}Test case #$i: Memory Limit Exceeded (MLE)
                in ${yellow}${elapsed_ms} ms${reset}"
            ((failed++))
        elif [ $ret_code -eq 139 ]; then
            echo -e "${bold}${red}Test case #$i: Runtime Error (Segmentation
                Fault) in ${yellow}${elapsed_ms} ms${reset}"
            ((failed++))
        else
            echo -e "${bold}${red}Test case #$i: Runtime Error (exit code
                $ret_code) in ${yellow}${elapsed_ms} ms${reset}"
            ((failed++))
        fi
        echo -e "${bold}${white}=====================================${reset}
            "
    done
    total_test_time_ms=$(echo "scale=3; $total_test_time_ns/1000000" | bc)
    avg_time=$(echo "scale=3; $total_test_time_ms/$MAX_TESTS" | bc)
    echo -e "\n${bold}${red}Stress testing completed.${reset}"
    echo -e "${bold}${green}[Accepted]: $accepted${reset} ${bold}${red}[
        Failed]: $failed${reset}"
    echo -e "${bold}${cyan}[Total testing time]: ${total_test_time_ms} ms${
        reset}"
    echo -e "${bold}${magenta}[Average time per test]: ${avg_time} ms${reset}
        "
    echo ""
}
check_files
```

```
compile_file "$TEST_GEN_FILE" "generator" ""
compile_file "$MAIN_FILE" "sol" ""
stress_test
```

Part V

Commands

Chapter 18

Linux Commands

18.1 Basic Linux Commands

`ls` List directory contents.

`cd` Change the current working directory.

`pwd` Print the absolute path of the working directory.

`cp` Copy files or directories.

`mv` Move or rename files or directories.

`rm` Remove files or directories.

`mkdir` Create one or more directories.

`rmdir` Remove empty directories.

`touch` Create an empty file or update file timestamps.

`cat` Concatenate and display file contents.

`less` View file contents page by page.

`grep` Search file(s) for lines matching a pattern.

`rm -rf` Remove folder.

18.2 Intermediate Commands

`find` Search for files in a directory hierarchy.

`chmod` Change file or directory permissions.

`chown` Change file or directory ownership.

`tar` Create or extract archive files.

`zip`/`unzip` Compress and decompress files in ZIP format.

`ssh` Securely connect to a remote host.

`scp` Securely copy files between hosts over SSH.

`rsync` Efficiently synchronize files and directories.

`ps` Display current active processes.

`kill`/`killall` Terminate processes by PID or name.

`top` Display dynamic real-time view of running processes.

`df` Report filesystem disk space usage.

`du` Estimate file and directory space usage.

`free` Display memory usage statistics.

`head`/`tail` Output the first or last part of files.

`awk` Pattern scanning and processing language.

`sed` Stream editor for filtering and transforming text.

18.3 Advanced Commands

`htop` Interactive process viewer with tree view and color coding.

`nmap` Network exploration and security auditing tool.

`tcpdump` Command-line packet analyzer.

`lsof` List open files and the processes that opened them.

`strace` Trace system calls and signals.

`iptables`/`nft` Configure Linux kernel packet filtering rules.

`journalctl` Query and display logs from the systemd journal.

`systemctl` Control and inspect systemd services and units.

`curl`/`wget` Transfer data to or from a server via various protocols.

`xargs` Build and execute command lines from standard input.

`dd` Convert and copy a file, often used for disk cloning.

`traceroute` Print the route packets take to network host.

`dig` DNS lookup utility.

`fuser` Identify processes using files or sockets.

`screen`/`tmux` Terminal multiplexers for session management.

`sudo apt-get autoremove -purge` It helps clean up your system and free up disk space.

`sudo apt-get update` It updates the local package index.

`sudo apt-get upgrade` It upgrades all installed packages on your system to their latest versions.

18.4 GIT Commands

Git is a distributed version control system used to track changes in source code during software development. It allows multiple developers to work together efficiently and supports branching, merging, and collaboration through remote repositories (like GitHub or GitLab).

Basic Configuration

Before using Git, configure your identity and preferred settings:

```
git config --global user.name "Your Name"
git config --global user.email "you@example.com"
git config --global init.defaultBranch main
git config --list   % View all configurations
```

Common Git Commands

`git init` Initializes a new Git repository in the current directory.

`git clone <url>` Downloads a repository from a remote URL to your local machine.

`git add <file>` Stages file(s) to be committed.

`git commit -m "message"` Commits staged changes with a message.

`git status` Displays the state of the working directory and staging area.

`git log` Shows the commit history.

`git diff` Shows the differences between files in the working directory and the index.

`git branch` Lists all local branches.

`git branch -M main` Renames the current branch to `main`.

`git checkout <branch>` Switches to the specified branch.

`git checkout -b <branch>` Creates a new branch and switches to it.

`git merge <branch>` Merges changes from another branch into the current one.

`git remote add origin <url>` Adds a remote repository (commonly GitHub).

`git remote set-url origin https://<token>@github.com/<user>/<repo>` Changes the URL of the existing remote.

`git push` Pushes local commits to the remote repository.

`git push origin main` Pushes the `main` branch to the remote named `origin`.

`git push -set-upstream origin main` Sets the upstream tracking for the current branch and pushes it.

`git pull` Fetches from the remote repository and merges changes into the current branch.

`git pull origin main` Pulls changes from the `main` branch of the remote `origin`.

`git reset -hard HEAD~1` Resets the current branch to the previous commit, discarding all changes.

`git stash` Temporarily saves uncommitted changes for later.

`git stash pop` Restores the most recent stashed changes.

`git rm <file>` Removes a file from the working directory and staging area.

`git tag <name>` Tags a commit with a specified name.

`git fetch` Retrieves changes from the remote without merging.

`git revert <commit>` Reverts the changes of a specific commit.

`git show <commit>` Displays information about a specific commit.

`git blame <file>` Shows who changed what and when in a file.

Best Practices

- Commit often with clear messages.
- Use branches to manage different features or bug fixes.
- Use `git pull -rebase` to maintain a linear history when collaborating.
- Always check `git status` before committing or pushing.