## Data Structures

- **Data Structure**: A way in which data is stored for efficient search
- Examples: **Stacks**, **Trees**, **Hash Maps**, **Arrays**.
- **Array**: A collection of elements of the same type, stored
- Size is fixed at creation.
- Random access: O(1) element access.

## Algorithms

- **Algorithm**: A specific procedure for solving a well-defined
- Example: **Merge Sort** – A divide-and-conquer sorting method that

## Pointers

- A variable that holds the **address** of a piece of memory.
- Dereferencing (`*p`) accesses the value stored at that address.

## Memory Management

- **malloc()** (C): Allocates raw memory; must use `free()` to release.
- **new** (C++): Allocates memory for an object/array; use
- **Memory Leak**: Allocated memory without deallocation.
- **Dangling Pointer**: Pointer referencing deallocated memory.

## Arrays

- **Static Array**: Size fixed at creation.
- **Dynamic Array**: Allocated with `new`, can be released with

## C-Strings & `std::string`

- **C-String**: Null-terminated array of characters.
- **`std::string`**: Class providing string manipulation in C++ STL.

## Scope

- **Local Scope**: Variables accessible only within their block.
- **Global Scope**: Variables accessible throughout the program.

## Namespaces

- Logical grouping of names to prevent naming conflicts.
- Example: `std::cout` from namespace `std`.

## Control Flow

- **If / Else If / Else**
- **Switch**
- **While Loop**
- **Do-While Loop**
- **For Loop**
- **Break** & **Continue**
- **If Statement**: Executes code based on a condition.
- **Switch Statement**: Selects execution path based on variable value.
- **While Loop**: Repeats while condition is true.
- **Do-While Loop**: Executes at least once, then repeats while
- **For Loop**: Loops with initialization, condition, and iteration
- **Break**: Exits loop/switch early.
- **Continue**: Skips to next loop iteration.

## Functions

- **Declaration**: States function name, parameters, and return type.
- **Definition**: Provides the body/implementation.
- **Pass by Reference**: Function arguments passed as references (&),
- **Function Overloading**: Multiple functions with the same name but
- **Function**: A block of code that performs a specific task.
- **Declaration**: Specifies name, return type, and parameters.
- **Definition**: Provides the implementation.
- **Pass by Reference**: Function receives the variable's reference
- **Function Overloading**: Multiple functions with same name but

## Structures & Classes

- **C-Style Struct**: Aggregates related variables into one type.
- **Class**: Encapsulation of data (member variables) and methods
- **Public**: Accessible from outside the class.
- **Private**: Accessible only from within the class.

## C-Style Structures

- **Struct**: Groups related variables into one type.
- Members accessed using `.` (dot) for objects, `->` for pointers.

## Classes

- **Class**: Blueprint defining data (**member variables**) and behavior
- **Public**: Accessible outside the class.
- **Private**: Accessible only within the class.

## Object-Oriented Principles

- **Abstraction**: Focus on essential features; hide implementation
- **Encapsulation**: Keep data and methods together; hide internal
- **Modularity**: Components have distinct purposes and can be reused
- **Hierarchical Organization**: "Is-a" relationships; specialized types

## Inheritance

- **Inheritance**: A derived class acquires members from a base class,
- **Base Class**: General type.
- **Derived Class**: Specialized type extending the base class.

## Polymorphism

- **Polymorphism**: Ability for different classes to be treated as
- **Overriding**: Derived class provides its own implementation of a
- **Overloading**: Same method name, different parameter list.
- **Virtual Function**: Enables runtime method resolution (dynamic

## Design Patterns

- Reusable solutions to common design problems.
- Examples: Recursion, Divide and Conquer, Adapter, Iterator, Template

## Abstract Classes

- Cannot be instantiated.
- Have at least one **pure virtual function** (`= 0`).
- Define an interface for derived classes to implement.

## Templates

- Allow classes and functions to operate with generic types, avoiding

## Exceptions

- **Exception**: Runtime error condition that can be handled with
- **Throw**: Signal an exception (`throw runtime_error("msg")`).
- **Catch**: Handle an exception (`catch (const runtime_error& e)`).
- **Error vs Exception**:
- *Error*: Crash-causing, not catchable (e.g., segmentation fault).
- *Exception*: Catchable runtime issue.

## Arrays and Vectors

- **Built-in Array**: Fixed-size block of memory; no methods,
- **`std::array`**: Fixed-size container with methods (C++ standard

- **`std::vector`**: Resizable array-like container with dynamic size

## Arrays

- **Array**: An Abstract Data Type (ADT) storing a fixed-size, indexed
- **Static Array**: Size fixed at compile time.
- **Dynamic Array**: Allocated with `new`; size fixed at allocation
- **Characteristics**:
- Random access (O(1) element access).
- Size cannot change without creating a new array and copying data.
- Not objects in C++ (no methods, no bounds checking).
- **Insertion into Array**: Requires shifting elements; O(n).
- **Removal from Array**: Requires shifting elements; O(n).
- **Insertion Sort**: Simple sorting algorithm that builds a sorted

## Standard Library Array (`std::array`)

- **Template**: `std::array<T, N>` — fixed-size, object wrapper around
- **Member Function Categories**:
- **Iterators**: `begin()`, `end()`, `rbegin()`, `rend()`.
- **Capacity**: `size()`, `max_size()`, `empty()`.
- **Access**: `operator[]`, `at()`, `front()`, `back()`, `data()`.
- **Modifiers**: `fill()`, `swap()`.

## Vectors (`std::vector`)

- **Vector**: Resizable array-like container supporting dynamic
- **Member Function Categories**:
- **Big Three**: Constructor, Destructor, Assignment Operator.
- **Iterators**: `begin()`, `end()`, `rbegin()`, `rend()` and constant
- **Capacity**: `size()`, `max_size()`, `resize()`, `capacity()`,
- **Access**: `operator[]`, `at()`, `front()`, `back()`, `data()`.
- **Modifiers**: `assign()`, `push_back()`, `pop_back()`, `insert()`,

## 2D Arrays

- **Static 2D Array**: `int arr[rows][cols];`
- **Dynamic 2D Array**: Array of pointers to arrays.
- **`std::vector<std::vector<T>>`**: Dynamic and resizable alternative

## Lists

- **List**: A sequence of nodes where each node stores:
- An element (data).
- One or more links to other nodes.
- **Advantages over Arrays**:

- No fixed size.
- Insertion/removal does not require shifting elements.
- **Disadvantages**:
- No constant-time random access.
- Must track length and position manually.
- **Terminology**:
- **Node**: Individual list element container.
- **Head**: First node in the list.
- **Tail**: Last node in the list (null link).
- **Element**: Stored data in a node.
- **Link**: Pointer to another node.

## Singly Linked List

- **Structure**: Each node has one link to the next node.
- **Key Operations**:
- `addFront()`: Insert at head.
- `addLast()`: Insert at tail.
- `removeFront()`: Remove head.
- `removeLast()`: Remove tail (O(n) without tail pointer).
- **Big Three**: Destructor, Copy Constructor, Assignment Operator

## Doubly Linked List

- **Structure**: Nodes have links to both next and previous nodes.
- **Advantage**: Efficient tail removal and bidirectional traversal.
- **Node Structure**: Stores element, `next`, and `prev` pointers.

## Sentinel Nodes

- **Purpose**: Dummy head and tail nodes to simplify edge cases in
- **Header/Trailer**: Special sentinels without stored data.

## Circular Linked List

- **Structure**: Tail links back to head.
- **Cursor**: Pointer to current position in traversal.

## Recursion

- **Definition**: A function calling itself until a base case is
- **Components**:
- **Base Case**: Terminates recursion.
- **Recursive Call**: Processes smaller subproblem.
- **Types**:
- **Linear Recursion**: One recursive call per activation.

- **Binary Recursion**: Two recursive calls per activation.
- **Multiple Recursion**: More than two recursive calls.
- **Tail Recursion**: Recursive call is the last action; can be
- **Applications**:
- Fibonacci sequence.
- Summation.
- Linked list algorithms (e.g., recursive delete).

## Algorithm Analysis & Big-O

- **Algorithm Analysis** – Study of algorithm efficiency in terms of
- **Experimental Approach** – Measuring runtime by running the
- **Theoretical Analysis** – Predicting runtime from pseudocode by
- **Primitive Operations** – Basic actions like assignments, arithmetic
- **Seven Common Growth Rates** –
- **Worst-Case Scenario** – Upper bound of runtime for the hardest
- **Asymptotic Analysis** – Focuses on fastest-growing term of runtime
- **Big-O Notation (O)** – Upper bound on growth rate.
- **Big-Omega (Ω)** – Lower bound on growth rate.
- **Big-Theta (Θ)** – Tight bound; both upper and lower bounds match.
- **Big-O Rules** – Use smallest class, drop lower-order terms, drop
- **Tips for Analysis** – Identify $n$, work from innermost loops

## Stacks

- **Stack (ADT)** – A collection of elements with **First-In, Last-Out
- **Core Operations**:
- **push(x)** – Add to top.
- **pop()** – Remove from top.
- **top()** – Return top without removing.
- **size()** – Number of elements.
- **isEmpty()** – Check if stack is empty.
- **Implementations**:
- **Array-Based Stack** – Fixed-size storage; push/pop at end.
- **Linked List Stack** – Dynamic size; push/pop at head.
- **C++ Standard Library** – Provides `std::stack` (based on `deque` by
- **Performance (Typical Big-O)** – All operations $O(1)$ on both array
- **Common Uses**:
- Undo functionality.
- Browser back button.
- Parsing/matching symbols (e.g., parentheses).

## Queue

- **Queue (ADT)** – Collection of elements with **First-In, First-Out

- **Core Operations**:
- **enqueue(x)** – Add element to rear.
- **dequeue()** – Remove element from front.
- **front()** – Return front element without removing.
- **isEmpty()** – Check if queue has no elements.
- **size()** – Number of elements.
- **Linked List Queue**:
- Front = head, Rear = tail.
- Both enqueue and dequeue in $O(1)$ with tail pointer.

## Deque (Double-Ended Queue)

- Pronounced "deck".
- Allows insertion/removal at both ends.
- **Core Operations**:
- `addFirst(x)` – Insert at head.
- `addLast(x)` – Insert at tail.
- `removeFirst()` – Remove from head.
- `removeLast()` – Remove from tail.
- `getFirst()` – Peek at head.
- `getLast()` – Peek at tail.
- `size()` – Number of elements.
- `isEmpty()` – True if empty.
- **Implementation**: Typically a **doubly linked list**; O(1) for both

## Adapter Pattern

- **Definition**: Structural design pattern that converts one interface
- **How It Works**:
- Wrap an existing class inside another.
- Provide the expected interface while internally using the original.
- **Example**: Implementing `Stack` using a `Deque` (DequeStack).

## Vector (Array List)

- **Vector (ADT)** – Extends array with dynamic resizing.
- **Core Operations**:
- `get(i)` – Access element at index i.
- `set(i, x)` – Replace element at index i.
- `insert(i, x)` – Insert at index i (O(n) worst-case).
- `erase(i)` – Remove at index i (O(n) worst-case).
- **Array-Based Implementation**:
- Store elements in contiguous array.
- Track number of stored elements.
- **Performance**:

- End insertion/removal: O(1) amortized with doubling strategy.
- Front insertion/removal: O(n).

## Dynamic Array Resizing Strategies

- **Incremental**: Increase capacity by a constant $c$ – inefficient
- **Doubling**: Double capacity when full – amortized O(1) per
- **Amortized Analysis**:
- Look at average time over many operations.
- Doubling strategy leads to $O(1)$ amortized insertion at end.

## Containers and Iterators

- **Container**: Data structure supporting element access via iterators.
- **Iterator**: Object that abstracts traversal through a container.
- **Iterator Operations**:
- `*p` – Access current element.
- `++p` – Move to next.
- `--p` – Move to previous (bidirectional).
- **Types**:
- **iterator** – Read/write access.
- **const_iterator** – Read-only.
- **bidirectional iterator** – Can move both ways.
- **random-access iterator** – Supports jumps (p + i).
- **STL Examples**: `vector`, `deque`, `list`.

## Iterators

- **Iterator** – Abstracts the process of scanning through a
- **Container** – Data structure that supports element access via
- **Types of iterators**:
- **iterator** – Read-write.
- **const_iterator** – Read-only.
- **bidirectional iterator** – Supports `++p` and `--p`.
- **random-access iterator** – Supports `p + i` and `p - i`.
- **Array-based iterator** – Uses an index to track position.
- **Linked list-based iterator** – Uses a pointer to the current node.

## Trees

- **Tree** – Non-linear data structure made of **vertices (nodes)**
- **Empty tree** – No vertices.
- **Leaf** – Node with degree 1.
- **Internal node** – Node with degree > 1.
- **Distance** – Number of edges between two nodes.
- **Rooted tree** – A tree with a designated **root** node.

- **Parent / Child** – Immediate neighbors in a root-directed
- **Ancestor / Descendant** – Nodes on the path to/from the root.
- **Subtree** – Node plus all its descendants.
- **Depth** – Number of edges from a node to the root.
- **Height** – Maximum depth of any node.

### Tree ADT

- **Element(p)** – Returns element in node p.
- **Root()** – Returns root node.
- **Parent(p)** / **Children(p)** – Returns parent or children of p.
- **isInternal(p)** / **isExternal(p)** – Checks if p is internal or
- **isRoot(p)** – Checks if p is the root.
- **Size()** – Number of nodes.
- **isEmpty()** – Checks if tree has nodes.
- **Iterator()** – Iterates over elements.
- **Positions()** – Returns all positions in the tree.
- **Replace(p, e)** – Replaces element in p with e.

### Special Types of Trees

- **Ordered tree** – Children have a defined order.
- **Balanced tree** – Height difference between subtrees ≤ 1.
- **Complete tree** – All levels filled except last, filled left to
- **Perfect tree** – All levels completely filled.
- **Binary tree** – Each node has ≤ 2 children.
- **Binary search tree** – Left subtree elements < node's element <

### Traversals

- **Preorder** – Visit node, then children.
- **Postorder** – Visit children, then node.
- **Inorder** (binary trees) – Visit left child, node, then right

### Binary Tree Properties

- **Max nodes**: $n \leq 2^{h+1} - 1$
- **Max external nodes**: $e \leq 2^h$
- **Max internal nodes**: $i \leq 2^h - 1$
- **Height range**: $\log_2(n+1) - 1 \leq h \leq n - 1$

## Binary Tree

- **Binary Tree** – A tree where each node has at most two children

### Linked Structure for Binary Trees

- **Node Structure**:
- `element`: Data stored in the node.
- `parent`: Pointer to the parent node.
- `left`: Pointer to the left child.

- `right`: Pointer to the right child.
- **LinkedBinaryTree**: Class holding a `root` pointer and `size`.

### Array-Based Binary Tree

- **Index Mapping Rules**:
- Root node v: $f(v) = 1$
- Left child of node u: $f(v) = 2f(u)$
- Right child of node u: $f(v) = 2f(u) + 1$
- Index 0 is unused for formula simplicity.
- **Advantages**:
- O(1) access by index.
- No pointers; simpler parent/child calculation.
- Less chance of memory leaks.
- Best for complete or nearly complete binary trees.
- **Disadvantages**:
- Wasted space for sparse trees.
- Resizing overhead.
- Poor for unbalanced trees.
- **Sparse Tree Space**:
- Array size for height $h$: $2^{h+1} - 1$ elements.
- Minimum nodes for height $h$: $h + 1$.

### Tree Traversal – Inorder

- **Inorder Traversal**:
- **Array Implementation**: Index-based recursive calls.
- **Linked Implementation**: Pointer-based recursive calls.

## Binary Search Tree (BST)

- **BST Property**:
- For every node:
- Left subtree elements < node's element.
- Right subtree elements > node's element.
- **Tree Search**:
- Recursively compare key to current node's element.
- Traverse left or right accordingly.
- External nodes (null children) left blank.
- **Search Complexity**:
- $O(h)$ where $h$ is tree height.
- Not guaranteed $O(\log n)$ unless balanced.

### BST Implementation Notes

- **isExternal(v)**: Returns true if node v has no children.
- **treeSearch(key, v)**:
- Base case: External node → return it.

- Recursive case: Compare key and traverse left/right.
- **Insertion**:
- Find external position via search.
- Replace with internal node containing the key.
- Add two external children.

## Priority Queue

- Abstract data type that stores elements with associated priorities.
- Operations:
- **Insert** – Add element with a priority.
- **RemoveMin / RemoveMax** – Remove element with highest or lowest
- **Min / Max** – Access element with highest or lowest priority

## Heap

- **Heap** – Key-based data structure storing keys as a complete binary
- **Max Heap** – Each child's key ≤ parent's key.
- **Min Heap** – Each child's key ≥ parent's key.
- **Complete Tree** – All levels full except possibly the last, filled
- **Insert**: Add at next available spot, swap up until heap property is
- **RemoveMin**: Replace root with last node, remove last node, swap
- **Complexity**:
- Insert – $O(\log n)$
- RemoveMin – $O(\log n)$
- Min – $O(1)$

## Map ADT

- **Map** – Stores unique key–value pairs (entries).
- **Operations**:
- **get(k)** – Retrieve value by key.
- **put(k, v)** – Insert or replace value for key.
- **remove(k)** – Remove entry by key.
- **keySet()** – Return collection of all keys.
- **values()** – Return collection of all values.
- **entrySet()** – Return collection of all entries.

## Hash Table

- **Hash Table** – Map implementation with expected $O(1)$ access
- **Bucket array** – Array of slots ("buckets") to store entries.
- **Hash function** – Maps keys to bucket indices.
- **Hash function steps**:
- **Division method**: $h(k) = k \bmod N$ (N prime recommended).
- **MAD method**: $h(k) = (ak + b) \bmod N$ (a, b chosen to avoid

### Collisions

- **Collision** – Different keys mapping to the same bucket.
- **Collision Handling**:
- **Separate Chaining** – Store multiple entries in the same bucket
- **Open Addressing** – Find another bucket:
- **Linear probing** – Check next slots sequentially.
- **Quadratic probing** – Use squared increments.
- **Double hashing** – Use second hash function to determine step

### Load Factor & Rehashing

- **Load factor (λ)** = $n/N$ (entries / buckets).
- High load factor → increased collisions → **rehashing** (increase N,

### C++ Map Implementations

- **`std::map`** – Ordered map using self-balancing BST (O(log n)
- **`std::unordered_map`** – Hash table implementation (O(1) expected

## Skip List

- **Skip List** – A probabilistic data structure for ordered elements
- **Height (h)** – Number of levels in the skip list.
- **Coin Flip Mechanic** – Determines how many levels a new entry
- **QuadNode Structure** – Each node stores `data`, and pointers to
- **TowerNode Structure** – Each tower represented as a single node
- **Worst-case**: $O(n + h)$ for search, insert, remove.
- **Average**: $O(\log n)$ for search, insert, remove.
- **Space**: Average $O(n)$, worst $O(hn)$.

## Dictionary ADT

- **Dictionary** – Searchable collection of key–element pairs allowing
- **Operations**:
- `get(k)` – Return an entry with key $k$.
- `getAll(k)` – Return all entries with key $k$.
- `put(k, v)` – Insert new entry.
- `remove(e)` – Remove a given entry.
- `entrySet()` – Return all entries.
- `isEmpty()` – Check if empty.
- `size()` – Number of entries.
- **Implementations**:
- Unordered linked list.
- Hash table with separate chaining.
- Ordered array (search table).
- Skip list.

## AVL Tree

- **AVL Tree** – Self-balancing binary search tree.
- **Height-Balancing Property** – Heights of children differ by at most
- **Height Bound** – $h < 2\log_2 n + 2$, $O(\log n)$ complexity.
- **Balance Factor** – height(right) − height(left).
- **Rotations**:
- **Single Rotation** – Left or right.
- **Double Rotation** – Left–Right or Right–Left.
- **Trinode Restructuring** – Rebalancing using three nodes $x, y, z$
- Search, Insert, Remove: $O(\log n)$.
- Restructuring: $O(1)$ with linked structure.

### AVL Trees

- **Meaning**: AVL stands for *Adelson-Velsky and Landis* (inventors).
- **Purpose**: A height-balanced variant of a Binary Search Tree (BST)
- **Height-Balancing Property**: For every internal node *v*, the
- **Node Structure**: Stores the same data as a BST plus the height of
- **Balance Factor**:
- **Rotations**:
- **Left rotation**
- **Right rotation**
- **Double rotations**: Left-Right or Right-Left (a.k.a. Trinode
- **Trinode Restructuring**: Let (*a*, *b*, *c*) be the in-order order
- **Insertion/Removal**: Done like in BST but may require rotations to
- **Performance**: Search, insert, and remove are $O(\log n)$; a single

## Multi-Way Search Tree

- **Definition**: An *ordered* tree where each internal node can have
- **d-node**: Node with *d* children.
- **Rules**:

### (2, 4) Tree

- **Multi-Way Search Tree** – Internal node can have multiple keys and
- **d-node** – Node with $d$ children.
- **Properties**:
- **(2, 4) Tree** – Special multi-way search tree:
- Node Size Property – At most 4 children.
- Depth Property – All external nodes at same depth.
- **Balanced** – No rotations needed.

### (2,4) Trees

- **Also Called**: 2–3–4 Trees (a type of Multi-Way Search Tree).
- **Node Size Property**: Every internal node has **at most 4
- **Depth Property**: All external (leaf) nodes have the **same depth**.

- **Node Types**:
- **2-node**: 2 children, 1 key.
- **3-node**: 3 children, 2 keys.
- **4-node**: 4 children, 3 keys.
- **Advantages**: Perfectly balanced; no rotations needed.
- **Insertion**:
- Insert key into the appropriate leaf.
- If overflow (>3 keys), split into two nodes and push the middle key
- May cause cascading splits up to root.
- **Removal**:
- Remove key from a leaf or swap with predecessor if internal.
- If underflow (<1 key), either **transfer** a key from a sibling or
- **Performance**: Height is $O(\log n)$. Search, insert, remove visit

## General Concepts

- **Sorting Algorithm** – A method for arranging elements of a list or
- **In-place Algorithm** – Performs sorting without requiring
- **Stable Sort** – Preserves the relative order of elements with equal
- **Time Complexity** – Describes how the runtime of an algorithm grows
- **Worst Case** – Maximum time an algorithm could take.
- **Average Case** – Expected time over random input.

## Divide-and-Conquer

- **Divide-and-Conquer Algorithm** – A design pattern involving:

## Insertion Sort

- **Definition** – Sorts by building the final sorted array one element
- **Properties** – In-place, stable, $O(n^2)$ average & worst-case.

## Merge Sort

- **Definition** – A divide-and-conquer algorithm that splits the array
- **Steps** – Divide, recursively sort, merge.
- **Complexity** – O(n log n) average & worst-case, stable, not

## Quicksort

- **Definition** – A divide-and-conquer algorithm that partitions the
- **Randomized Quicksort** – Picks pivot randomly to reduce worst-case
- **Complexity** – O(n log n) average, $O(n^2)$ worst-case, in-place,

## Heapsort

- **Definition** – Builds a max-heap and repeatedly extracts the

- **Steps** – Heapify array, remove root n times.
- **Complexity** – O(n log n) average & worst-case, in-place, unstable.

## Lower Bound for Comparison Sorting

- **Comparison Sorting** – Sorting using only element comparisons
- **Lower Bound** – Any comparison-based sorting requires Ω(n log n)

## Bucket Sort

- **Definition** – Distributes elements into buckets based on their
- **Complexity** – O(n + N) time, O(n + N) space, stable if buckets use

## Radix Sort

- **Definition** – Sorts keys with multiple digits by applying a stable
- **Complexity** – O(dn), where d is number of digits.

## Graph Basics

- **Graph**: A pair $(V, E)$ where:
- $V$ = set of **vertices** (nodes)
- $E$ = collection of **edges** (pairs of vertices)
- **Directed Edge**: Ordered pair $(u, v)$ with origin $u$ and
- **Undirected Edge**: Unordered pair $(u, v)$.
- **Weighted Graph**: Graph with a numerical value (weight) assigned to
- **Connected Graph**: Every pair of distinct vertices has a path
- **Disconnected Graph**: At least two vertices have no path connecting

## Edge List

- Stores a list of vertices and a list of edges.
- **Complexities** (n = |V|, m = |E|):
- Space: $O(n + m)$
- Insert Vertex: $O(1)$
- Insert Edge: $O(1)$
- Remove Vertex: $O(m)$
- Remove Edge: $O(1)$
- Is Adjacent: $O(m)$

## Adjacency Matrix

- n × n matrix $A$ where entry $a_{ij}$ indicates presence/weight of
- **Complexities**:
- Space: $O(n^2)$
- Insert Vertex: $O(n^2)$
- Insert Edge: $O(1)$

- Remove Vertex: $O(n^2)$
- Remove Edge: $O(1)$
- Is Adjacent: $O(1)$

## Adjacency List

- Each vertex has a list of adjacent vertices.
- **Complexities**:
- Space: $O(n+m)$
- Insert Vertex: $O(1)$
- Insert Edge: $O(1)$
- Remove Vertex: $O(\deg(v))$
- Remove Edge: $O(1)$
- Is Adjacent: $O(\deg(v))$

## Depth-First Search (DFS)

- Explores as far as possible along each branch before backtracking.
- Can be implemented recursively or using a stack.
- **Time Complexity**: $O(n+m)$
- Applications:
- Find a path between two vertices
- Detect cycles

## Breadth-First Search (BFS)

- Explores neighbors level by level, using a queue.
- **Time Complexity**: $O(n+m)$
- Applications:
- Find shortest path in terms of number of edges
- Detect cycles

## Dijkstra's Algorithm

- Finds shortest paths from a single source in weighted graphs with
- Uses a priority queue to select the next closest vertex.
- **Complexity**:
- Removing all vertices from PQ: $O(n\log n)$
- Relaxing all edges: $O(m\log n)$
- Connected graph: $O(m\log n)$

## Spanning Tree

- A subset of a graph's edges that connects all vertices **without
- Can be produced by BFS or DFS.

## Minimum Spanning Tree (MST)

- A spanning tree of a **weighted graph** that has the **minimum total

## Prim–Jarnik's Algorithm

- **Purpose**: Finds an MST.
- **Approach**:
- Start from an arbitrary vertex $s$, grow MST as a "cloud" of
- Each vertex $v$ has a label $d(v)$ = smallest weight edge connecting
- At each step:
- **Complexity**: $O((n+m)\log n)$ with adjacency list + heap-based
- **Similar To**: Dijkstra's algorithm (but for MST, not shortest

## Kruskal's Algorithm** (mentioned in comparison)

- Sorts edges by weight, adds edges to MST if they don't form a cycle.
- Uses **Union–Find** data structure.

## Set (Data Structure)

- **Definition**: Collection of unique elements, no order implied.
- **Core Operations**:
- `Add` – Insert an element.
- `Remove` – Delete an element.
- `Contains` – Check membership.
- `Iterator` – Access all elements.

## C++ Implementations

- `std::set`: Ordered set (red-black tree).
- `std::unordered_set`: Unordered set (hash table).

## Set Theory Operations

- **Union**: Combines elements from two sets (no duplicates).
- Example: $\{3,2,6\} \cup \{5,6,9\} = \{3,2,6,5,9\}$
- **Intersection**: Elements common to both sets.
- Example: $\{3,2,6\} \cap \{5,6,9\} = \{6\}$
- **Subtraction (Difference)**: Elements in one set but not the other.
- Example: $\{3,2,6\} - \{5,6,9\} = \{3,2\}$

## C++ Set Operations via `<algorithm>`

- `set_union`
- `set_intersection`
- `set_difference`

- Require **ordered containers** with iterators.
- Since C++17: `.merge()` method for sets.