Trees

SDB

Spring 2025

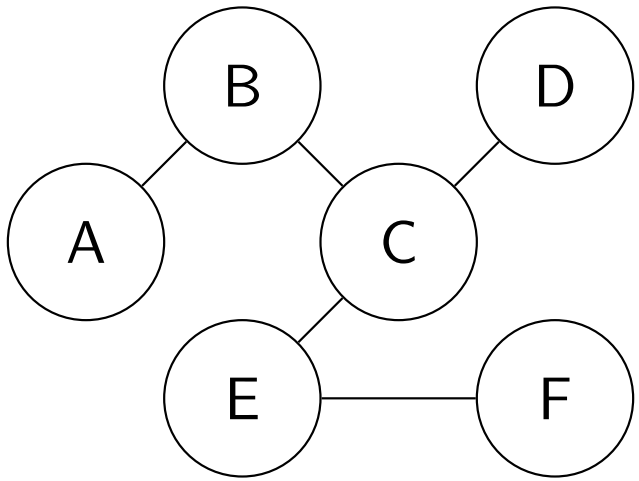
Outline

1. Characterization
2. Types
3. MST

Introduction to Trees

Definition: A tree is a connected, acyclic graph.

- **Properties:**
  - Any two vertices are connected by exactly one path.
  - Adding any edge creates a cycle.
  - Removing any edge disconnects the tree.
- **Example:**



Exercise:

- Prove: A tree with n vertices has $n - 1$ edges.

Characterization of Trees I

Definition: A **tree** is a connected, acyclic, undirected graph. Formally, a graph $T = (V, E)$ is a tree if:

- T is connected (a path exists between any two vertices).
- T contains no cycles.

Equivalent Characterizations:

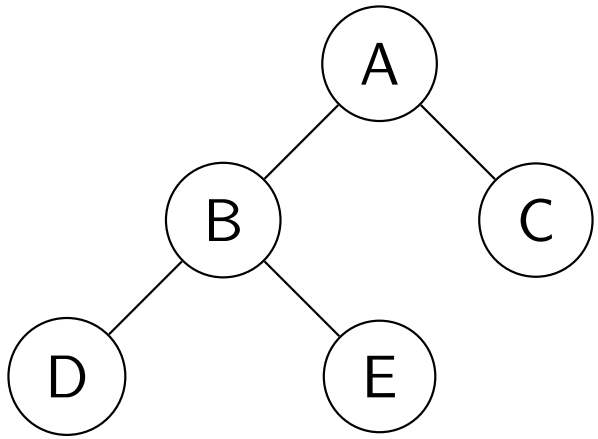
- A graph T is a tree if and only if it has exactly $n - 1$ edges for n vertices.
- There exists a unique path between any two vertices in T.
- Removing any edge from T disconnects it.
- Adding any edge to T creates exactly one cycle.

Characterization of Trees II

Theorem: A Tree with n vertices has $n - 1$ edges.
Proof (Induction):

- **Base Case:** A single-vertex tree has 0 edges.
- **Inductive Hypothesis:** Assume every tree with n vertices has $n - 1$ edges.
- **Inductive Step:** Adding a new vertex and connecting it with an edge maintains a tree structure and increases the edge count by 1.

Example Illustration:



Characterization of Trees III

Exercise:

- Prove that every tree has at least one leaf (a vertex with degree 1).
- Find all spanning trees of the complete graph K_4.

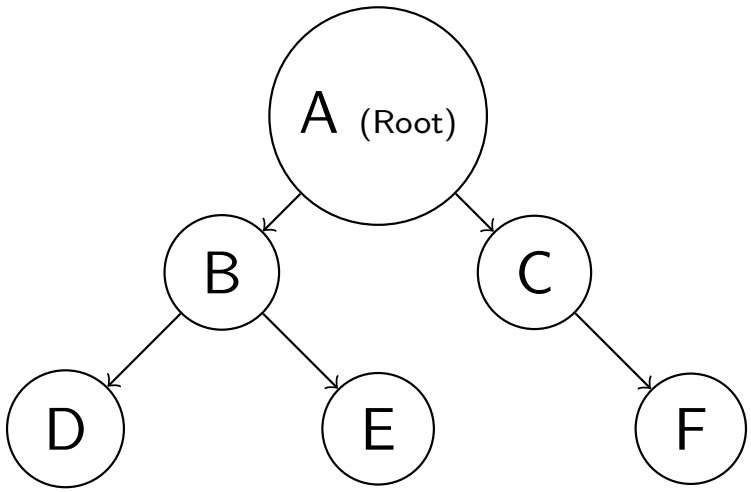
Rooted Trees - Definition and Properties I

Definition: A **rooted tree** is a tree with a designated root vertex. Every other vertex has a unique parent except the root.

Properties:

- The root is the unique vertex with no parent.
- Every edge in a rooted tree is directed away from the root.
- The **depth** of a node is the number of edges from the root to that node.
- The **height** of a tree is the maximum depth of any node.

Example of a Rooted Tree:



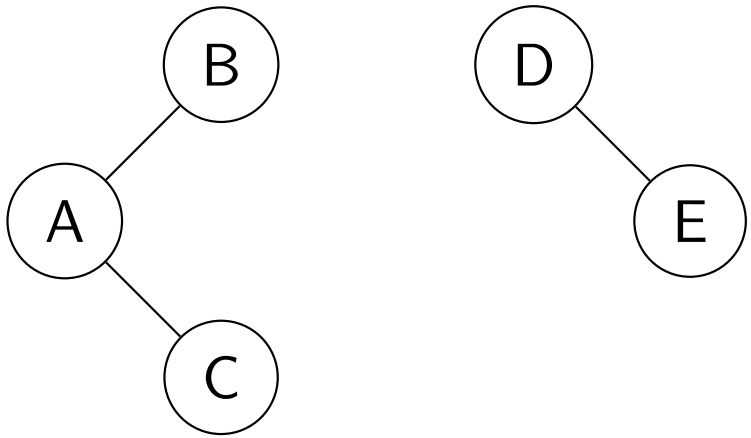
Rooted Trees - Definition and Properties II

Exercise:

- Find the depth and height of the given tree.
- Prove that every rooted tree with at least two vertices has at least one leaf.
- Prove: The height of a complete binary tree with n nodes is $\lfloor \log_2 n \rfloor$.

Relationship Between Trees and Forests

- **Tree:** A connected acyclic graph.
- **Forest:** A disjoint union of trees.
- **Key Properties:**
  - A forest with n vertices and k connected components has $n - k$ edges.
  - Removing an edge from a tree creates a forest.
  - Adding an edge between two trees in a forest creates a tree.
- **Example:**



- **Exercise:**
  - Prove: A forest is acyclic.
  - Verify the edge property for a forest with $n = 6$, $k = 2$.

Outline

1 Characterization

2 Types

3 MST

Types of Trees in Graph Theory I

Definition: A tree is a connected, acyclic graph. Different tree types arise in graph theory based on structure and properties.

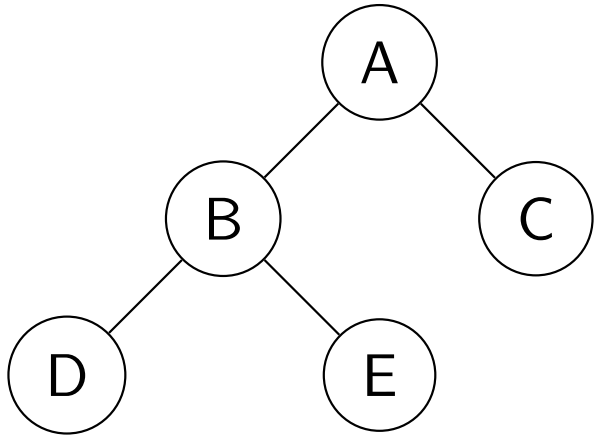
Common Types of Trees:

- **Binary Tree:** Each node has at most two children.
- **Complete Binary Tree:** All levels are fully filled except possibly the last.
- **k-ary Tree:** Each node has at most k children.
- **Spanning Tree:** A tree that spans all vertices of a graph.
- **Caterpillar Tree:** A tree where removing all leaves results in a path graph.
- **Lobster Tree:** A tree where removing all leaves and their immediate neighbors results in a path.
- **Star Tree:** A tree with one central vertex connected to all other vertices.

Types of Trees in Graph Theory II

- **Centroid Tree:** A tree where removing the centroid divides the tree into subtrees of equal size.

Example Visualizations:



Exercise:

- Identify which tree types appear in different network structures.
- Prove that every connected acyclic graph with n vertices has $n - 1$ edges.

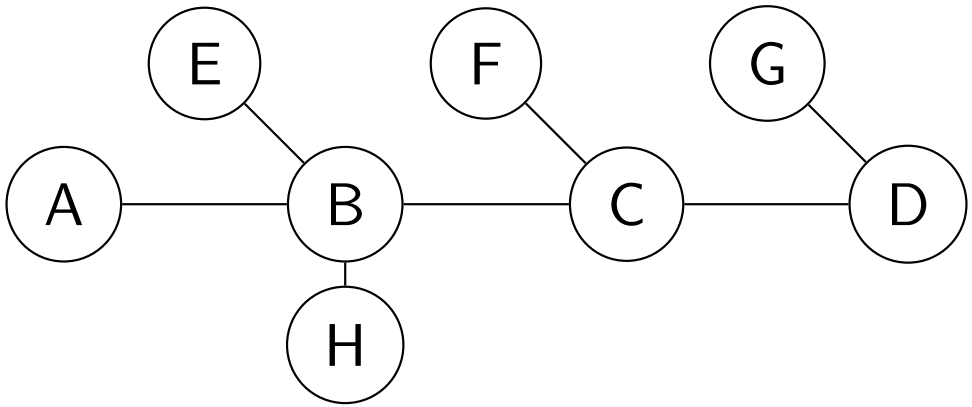
Caterpillar Trees

Definition: A **caterpillar tree** is a tree that becomes a **path graph** when all its leaves are removed.

Properties:

- Has a **central spine** (a main path of vertices).
- All other vertices (leaves) are directly connected to this spine.
- Common in **phylogenetics** and **network topologies**.

Example: Caterpillar Tree



Exercise:

- Prove that every caterpillar tree is **bipartite**.
- Find the height of a caterpillar tree with 10 vertices.

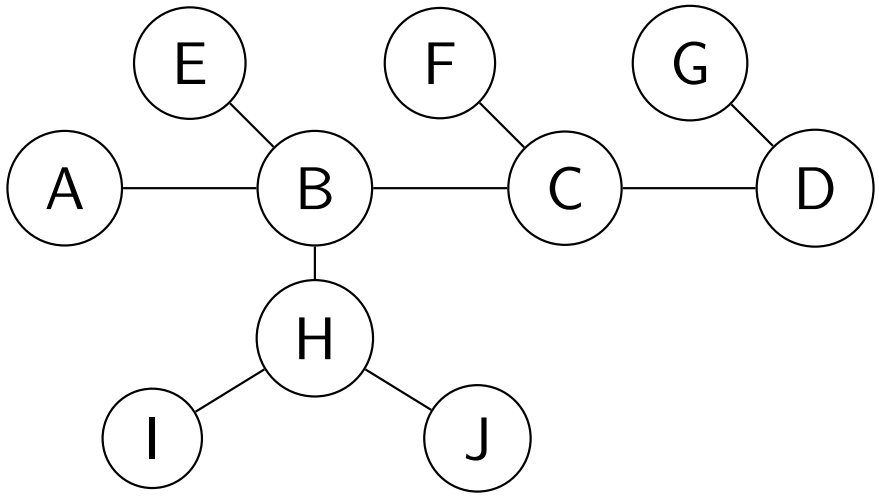
Lobster Trees

Definition: A **lobster tree** is a tree that becomes a **caterpillar tree** when all leaves are removed.

Properties:

- Can be seen as a **generalization of a caterpillar tree**.
- Common in **biology (phylogenetics)** and **network backbone design**.

Example: Lobster Tree



Exercise:

- Prove that every lobster tree has a **Hamiltonian path**.
- Find the **diameter** of a lobster tree with 15 nodes.

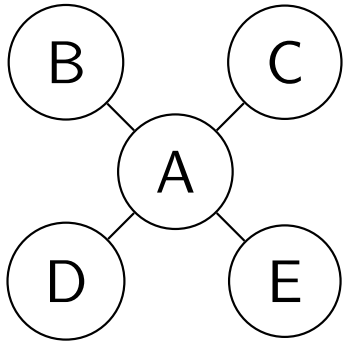
Star Trees

Definition: A **star tree** (or star graph) is a tree with one **central vertex** connected to all other vertices.

Properties:

- Has exactly **one internal node (central vertex)**.
- The **diameter is always 2**.
- Used in **wireless networking, hub-spoke designs, and interconnection networks**.

Example: Star Tree



Exercise:

- Prove that the **minimum spanning tree** of a complete graph K_n with uniform weights is a star tree.
- Calculate the **degree sequence** of a star tree with 6 vertices.

Binary Trees - Definition and Structure I

Definition: A **binary tree** is a rooted tree where each node has at most two children.

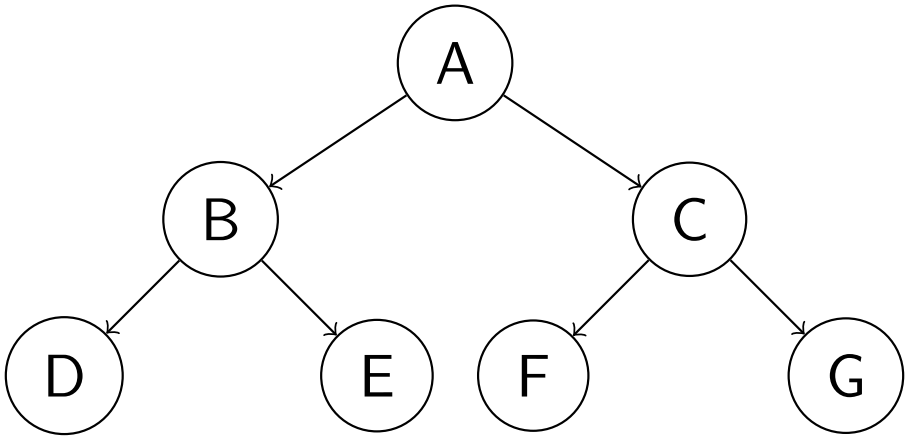
Types of Binary Trees:

- **Full Binary Tree:** Every node has either 0 or 2 children.
- **Complete Binary Tree:** All levels are fully filled except possibly the last.
- **Perfect Binary Tree:** A complete binary tree where all leaf nodes are at the same depth.

Theorem: The number of leaf nodes in a full binary tree with n internal nodes is $n + 1$.

Example: Full Binary Tree

Binary Trees - Definition and Structure II

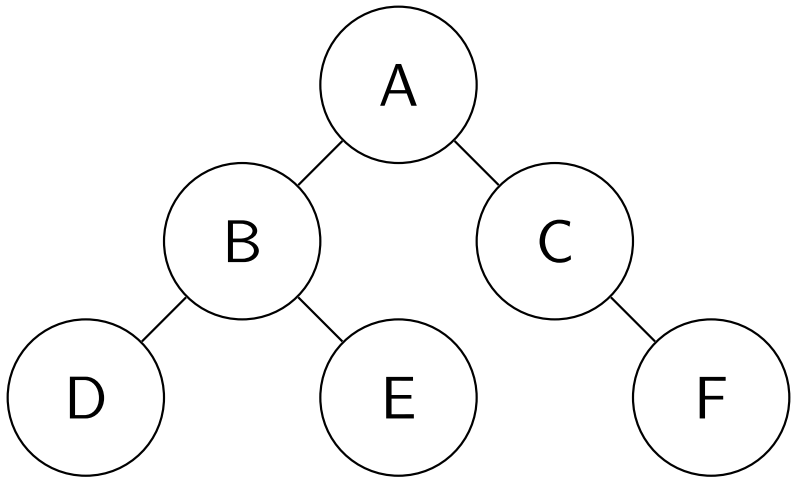


Exercise:

- Prove that the height of a perfect binary tree with n nodes is $\log_2(n+1) - 1$.
- Find the minimum and maximum height of a binary tree with 15 nodes.

Binary Trees and Traversals

- **Binary Tree:** A tree where each node has at most two children (left and right).
- **Traversals:**
  - **Preorder:** Root, Left, Right
  - **Inorder:** Left, Root, Right
  - **Postorder:** Left, Right, Root
- **Example:**



Exercise:

- Perform Preorder, Inorder, and Postorder traversals on the tree above.

Spanning Trees - Definition and Basic Properties I

Definition: A **spanning tree** of a graph $G = (V, E)$ is a subgraph that:

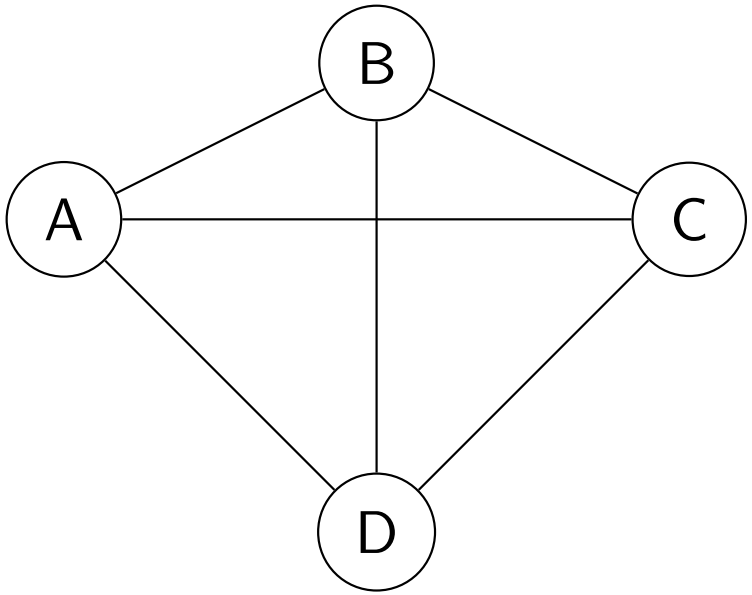
- Includes all vertices of G.
- Is a tree (connected and acyclic).
- Contains exactly $|V| - 1$ edges.

Basic Properties:

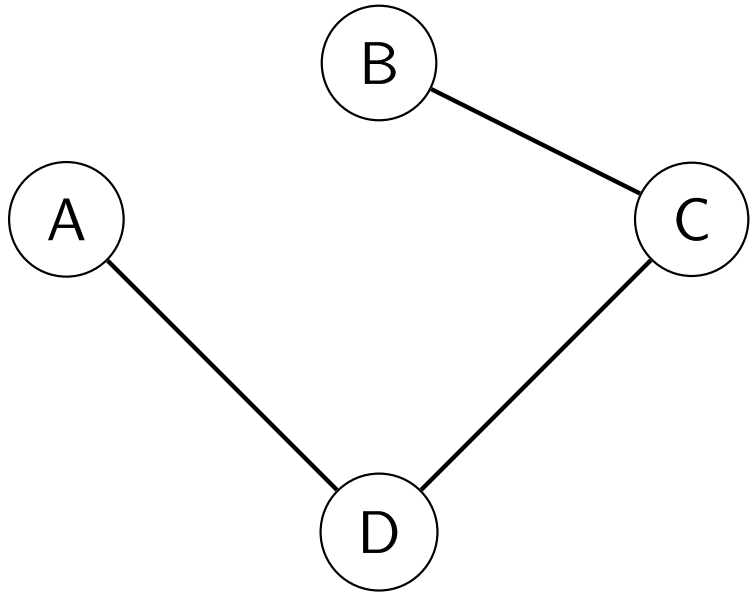
- Every connected graph has at least one spanning tree.
- A spanning tree is a minimal subgraph that preserves connectivity.
- Removing any edge from a spanning tree disconnects the graph.
- Adding any edge to a spanning tree creates a unique cycle.

Spanning Trees - Definition and Basic Properties II

Original Graph:



A Spanning Tree:



Exercise:

- Verify that the spanning tree has exactly $|V| - 1$ edges.
- Find all possible spanning trees of a cycle graph C_4.
- Find all spanning trees for the complete graph K_4.

Properties and Theorems on Spanning Trees

Fundamental Theorem of Spanning Trees:

- A connected graph with n vertices has at least one spanning tree.
- If a graph has a cycle, at least one edge can be removed while maintaining a spanning tree.

Property 1: Unique Path Property In any spanning tree T of a graph G, there exists a unique path between any two vertices.
Property 2: Minimal Connectivity If any edge is removed from a spanning tree, the graph becomes disconnected.
Property 3: Maximum Acyclicity If any new edge is added to a spanning tree, exactly one cycle is formed.
Exercise:

- Prove that a graph with n vertices and $n-1$ edges that is connected must be a tree.
- Construct spanning trees for a complete graph K_4.

Spanning Trees and Cayley's Formula

- **Definition:** A spanning tree of a graph is a subgraph that is a tree and connects all vertices.
- **Cayley's Formula:** The number of spanning trees of a complete graph K_n is n^{n-2}.
- **Proof:** (Sketch)
  - Use Prüfer sequences, which provide a bijection between labeled trees and sequences of length $n-2$ over n labels.
- **Example:** K_4 has $4^{4-2} = 16$ spanning trees.

Exercise:

- Verify Cayley's formula for K_3 and K_4.

AVL Trees I

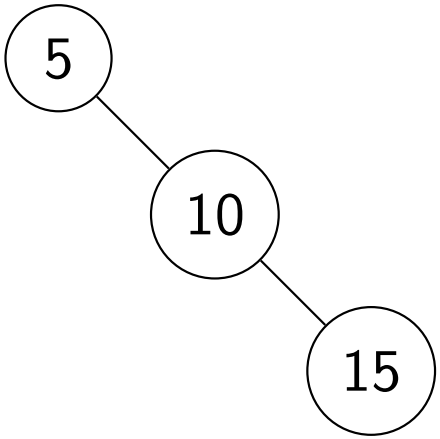
Definition: An **AVL tree** is a self-balancing **binary search tree (BST)** where the height difference (balance factor) between left and right subtrees is at most 1.

Properties:

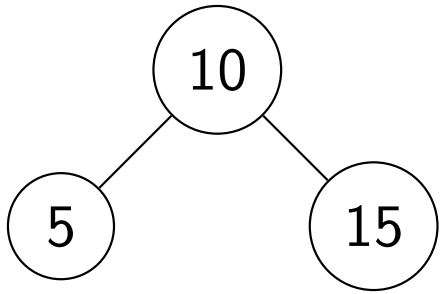
- Maintains $O(\log n)$ search, insertion, and deletion times.
- Uses **rotations (single and double)** to restore balance after insertion/deletion.

Example: Balanced vs Unbalanced Tree

Unbalanced BST:



Balanced AVL Tree:



Rotations for Balancing:

AVL Trees II

- **LL Rotation:** When the left subtree grows too deep.
- **RR Rotation:** When the right subtree grows too deep.
- **LR Rotation:** When left-right imbalance occurs.
- **RL Rotation:** When right-left imbalance occurs.

Exercise:

- Insert elements **(5, 10, 15, 20, 25)** into an AVL tree and show rotations.
- Prove that an AVL tree with n nodes has height at most $O(\log n)$.

Outline

1 Characterization

2 Types

3 MST

Spanning Trees in Weighted Graphs I

Definition: A **Minimum Spanning Tree (MST)** of a weighted graph is a spanning tree that has the minimum possible total edge weight.

Applications:

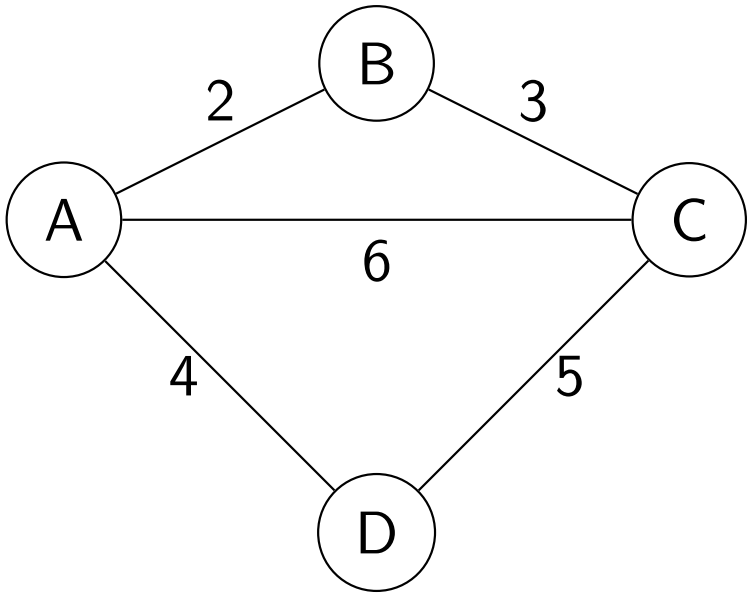
- Network Design (telecommunications, electrical networks)
- Cluster Analysis (data science, machine learning)
- Approximation Algorithms (for NP-hard problems like TSP)

Algorithms for Finding MST:

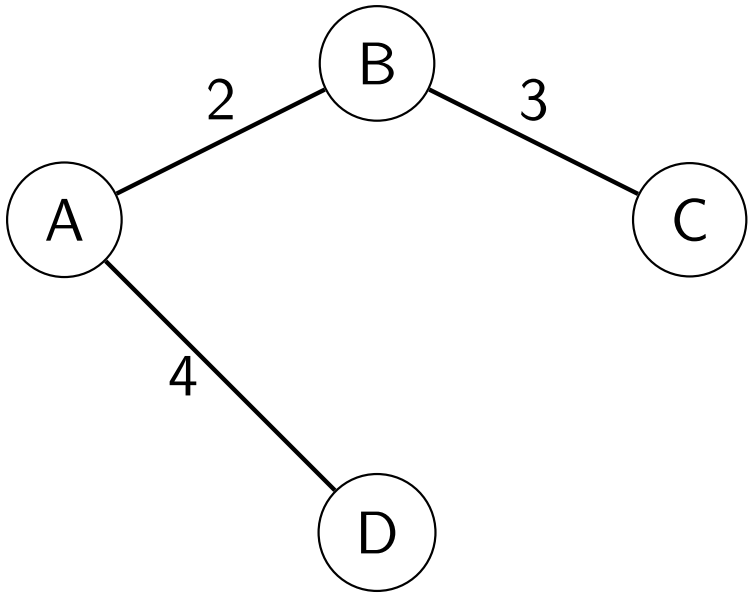
- **Kruskal's Algorithm** - Greedy algorithm that sorts edges by weight and adds edges while avoiding cycles.
- **Prim's Algorithm** - Starts from an arbitrary node and grows the tree by adding the smallest adjacent edge.

Spanning Trees in Weighted Graphs II

Weighted Graph:



Minimum Spanning Tree:



Exercise:

- Prove that Kruskal's algorithm produces an MST.
- Find an MST for a complete graph with random weights.

Proof of Kruskal's Algorithm

Claim: Kruskal's algorithm always produces an MST.

Proof (Greedy Choice and Safe Edge Property):

- **Step 1: Greedy Choice Property**
  - Consider an MST T^* and the edge e added by Kruskal's algorithm.
  - If $e \in T^*$, we are done.
  - Otherwise, adding e forms a cycle. Removing the **heaviest edge in the cycle** still results in an MST.
- **Step 2: Safe Edge Property**
  - Kruskal's algorithm selects the **minimum-weight edge** that connects two disjoint components.
  - This ensures that each added edge is **safe** and maintains the MST structure.

Conclusion: By **inductive argument**, the algorithm constructs an MST by always selecting a safe edge.

Exercise:

- Prove that Kruskal's algorithm maintains acyclic behavior at every step.

Complexity Analysis of Kruskal's Algorithm

Time Complexity Breakdown:

- **Step 1:** Sorting the edges - Sorting E edges takes $O(E \log E)$.
- **Step 2:** Union-Find Operations - Each union or find operation takes $O(\alpha(V))$ time using the Union-Find with path compression. - There are at most $V - 1$ union operations. - Total time: $O(E\alpha(V))$.

Overall Time Complexity: $O(E \log E) + O(E\alpha(V)) = O(E \log E)$ since $\alpha(V)$ **(inverse Ackermann function) grows very slowly**.

Comparison with Prim's Algorithm:

Algorithm	**Best for**	**Time Complexity**
Kruskal's	Sparse graphs	$O(E \log E)$
Prim's (Binary Heap)	Dense graphs	$O(E \log V)$
Prim's (Fibonacci Heap)	Dense graphs	$O(V^2)$

Exercise:

- Compare Kruskal's and Prim's algorithms on a **graph with 1000 vertices and 2000 edges**.
- Explain why Kruskal's algorithm is preferred in **disconnected graphs**.

Prim's Algorithm - Definition and Steps I

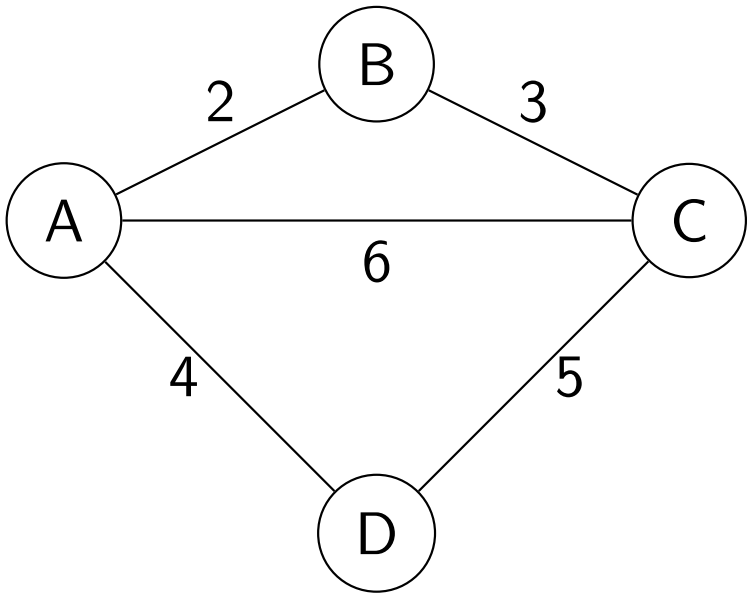
Definition: Prim's algorithm is a **greedy algorithm** that grows the Minimum Spanning Tree (MST) **from a single starting vertex**, adding the smallest edge that expands the tree.

Algorithm Steps:

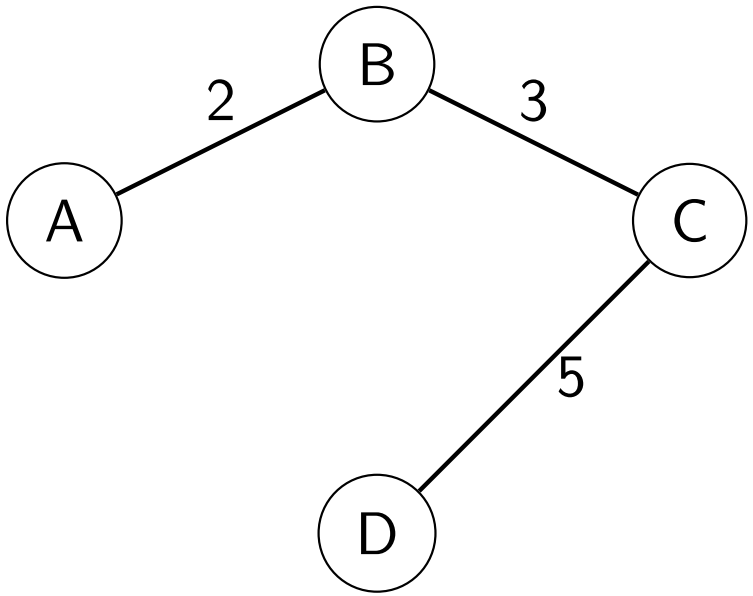
1. Select an arbitrary starting vertex.
2. Maintain a priority queue of edges **connecting the MST to the rest of the graph**.
3. Repeatedly:
   - Pick the **smallest weight edge** that connects an MST vertex to a non-MST vertex.
   - Add this edge and the new vertex to the MST.
   - Update the priority queue with edges from the newly added vertex.
4. Stop when all vertices are included in the MST.

Prim's Algorithm - Definition and Steps II

Graph with Weights:



MST Using Prim's Algorithm:



Exercise:

- Implement Prim's algorithm using a **priority queue**.
- Compare the performance of Prim's algorithm with Kruskal's on **dense graphs**.

Proof of Prim's Algorithm

Claim: Prim's algorithm always produces an MST.

Proof (Greedy Choice and Cut Property):

- **Step 1: Greedy Choice Property**
  - Assume Prim's algorithm has already built a partial MST T.
  - It picks the **smallest edge crossing the cut** $(T, V - T)$.
  - Let $e = (u, v)$ be this edge, and assume T^* is the optimal MST.
- **Step 2: Cut Property**
  - If $e \in T^*$, we are done.
  - If $e \notin T^*$, adding e to T^* creates a **unique cycle**.
  - The heaviest edge in this cycle can be removed while keeping the tree connected.
  - This transformation does not increase total weight, proving correctness.

Conclusion: By **mathematical induction**, Prim's algorithm constructs an MST by always picking a **safe edge**.

Exercise:

- Prove: Prim's algorithm works correctly for **disconnected graphs**.
- Find an example where Prim's and Kruskal's algorithms produce **different MSTs**.

Complexity Analysis of Prim's Algorithm I

Time Complexity Breakdown:

- **Using a Priority Queue (Binary Heap):**
  - Extracting the minimum edge: $O(\log V)$ per vertex.
  - Updating the priority queue: $O(\log V)$ per edge.
  - Total complexity: $O(E \log V)$.
- **Using an Adjacency Matrix:**
  - Finding the minimum edge takes $O(V)$.
  - Total complexity: $O(V^2)$.

Comparison with Kruskal's Algorithm:

Algorithm	**Best for**	**Time Complexity**
Kruskal's	Sparse graphs	$O(E \log E)$
Prim's (Binary Heap)	Dense graphs	$O(E \log V)$
Prim's (Adjacency Matrix)	Extremely dense graphs	$O(V^2)$

Complexity Analysis of Prim's Algorithm II

Key Observations:

- **Kruskal's is preferred for sparse graphs** since it sorts edges first.
- **Prim's (Binary Heap) is better for dense graphs** since it operates directly on vertices.
- **Prim's (Adjacency Matrix) is ideal for graphs where $E \approx V^2$.**

Exercise:

- Compare Prim's and Kruskal's algorithms on a **graph with 500 vertices and 1000 edges**.
- Explain why Prim's algorithm is **more efficient for connected graphs**.

Outline

4 Appendix
- More on Spanning Tree
- More Tree Types
- More
- Handy Proofs and Results

Outline

4 Appendix
- More on Spanning Tree
- More Tree Types
- More
- Handy Proofs and Results

Kirchhoff's Theorem I

- **Statement:** The number of spanning trees in a connected graph $G = (V, E)$ is given by the determinant of any cofactor of its Laplacian matrix.
- **Definitions:**
  - The **Laplacian matrix** L of a graph G is defined as:

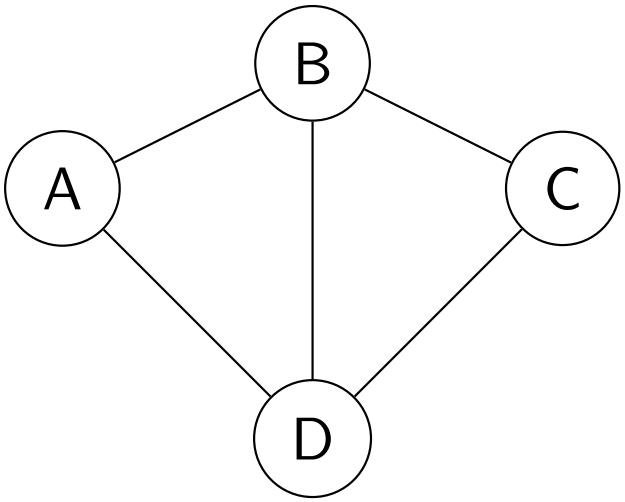
$$L[i][j] = \begin{cases} \text{degree of } v_i, & \text{if } i = j; \\ -1, & \text{if } (v_i, v_j) \in E; \\ 0, & \text{otherwise.} \end{cases}$$

  - A **cofactor** of L is obtained by removing any row i and column i.
- **Theorem:** Let L' be the cofactor of L. The number of spanning trees is:

$$\tau(G) = \det(L').$$

Kirchhoff's Theorem II

Graph:



Steps:

- **Laplacian Matrix:**

$$L = \begin{bmatrix} 2 & -1 & 0 & -1 \\ -1 & 3 & -1 & -1 \\ 0 & -1 & 2 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

- Remove row 1 and column 1 to form L'.
- Compute $\det(L')$ to find the number of spanning trees.
- **Result:** $\tau(G) = 16$.

Exercise:

- Compute the number of spanning trees for K_4 using Kirchhoff's theorem.
- Prove that $\tau(G) \geq 1$ for any connected graph G.

Kirchhoff's Theorem (Extended) I

- **Statement Recap:** The number of spanning trees of a connected graph G is equal to the determinant of any cofactor of its Laplacian matrix.
- **Proof Outline: 1. Laplacian Matrix Representation:**
  - The Laplacian matrix L satisfies:

$$L[i][j] = \begin{cases} \deg(v_i), & i = j; \\ -1, & (v_i, v_j) \in E; \\ 0, & \text{otherwise.} \end{cases}$$

  - L is symmetric and satisfies $\sum_j L[i][j] = 0$ for all i.

  2. Matrix-Tree Theorem:
  - The determinant of any cofactor L' corresponds to the sum of weights of all spanning trees.
  - This is achieved by expanding $\det(L')$ as the sum over all cycle-free subgraphs (spanning trees).

Kirchhoff's Theorem (Extended) II

3. Key Observation:

- Each spanning tree contributes exactly one term in the determinant expansion of L'.
- Thus, $\det(L')$ equals the total number of spanning trees.

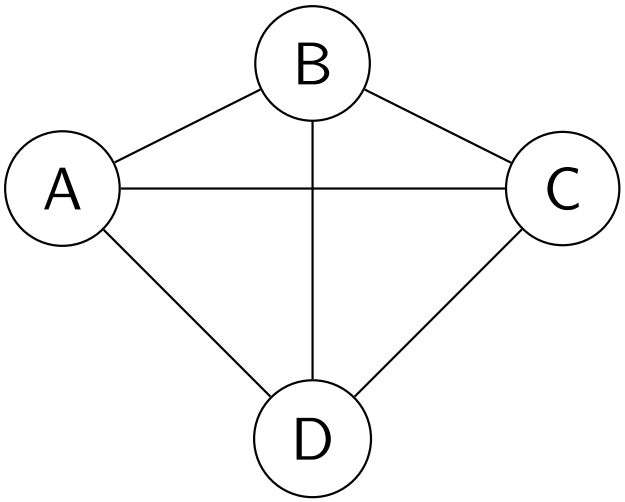
Computing Spanning Trees Using Kirchhoff's Theorem

- **Example:** Compute the number of spanning trees for K_4.

Steps:

- **Laplacian Matrix:**

Graph K_4:



$$L = \begin{bmatrix} 3 & -1 & -1 & -1 \\ -1 & 3 & -1 & -1 \\ -1 & -1 & 3 & -1 \\ -1 & -1 & -1 & 3 \end{bmatrix}$$

- Remove the last row and column to form L'.
- Compute:

$$\det(L') = \begin{vmatrix} 3 & -1 & -1 \\ -1 & 3 & -1 \\ -1 & -1 & 3 \end{vmatrix} = 16.$$

- **Result:** K_4 has 16 spanning trees.

Exercise:

- Compute the number of spanning trees for C_4.
- Verify Kirchhoff's theorem using a star graph $K_{1,n}$.

Proof of Kirchhoff's Theorem I

Statement: The number of spanning trees in a connected graph G is equal to the determinant of any cofactor of its Laplacian matrix.

Proof Outline:

- **Step 1: Laplacian Matrix Construction** The Laplacian matrix L of G is defined as:

$$L[i][j] = \begin{cases} \deg(v_i), & \text{if } i = j; \\ -1, & \text{if } (v_i, v_j) \in E; \\ 0, & \text{otherwise.} \end{cases}$$

- **Step 2: Kirchhoff's Matrix-Tree Theorem**
  - Removing the i-th row and column from L gives a cofactor matrix L', which corresponds to spanning subgraphs.
  - The determinant $\det(L')$ expands into terms corresponding to subsets of E of size $|V| - 1$.

Proof of Kirchhoff's Theorem II

- **Step 3: Contribution of Spanning Trees**
  - For each subset $F \subseteq E$ with $|F| = |V| - 1$:
    - ⋆ If F forms a spanning tree, its contribution to $\det(L')$ is non-zero.
    - ⋆ If F does not form a spanning tree, its contribution cancels out due to symmetry in L.
  - Therefore, $\det(L')$ counts all spanning trees exactly once.
- **Step 4: Conclusion** The determinant $\det(L')$ equals the number of spanning trees in G.

Key Insight: The Laplacian matrix encodes the connectivity of the graph, and its cofactors represent spanning tree contributions.

Exercise:

- Prove that $\sum_{i=1}^{n} \deg(v_i) = 2|E|$ and verify its role in constructing L.
- Compute $\det(L')$ for a path graph P_4 and verify the result using enumeration of spanning trees.

Comparison of MST Algorithms in Literature I

Different MST Algorithms and Their Applications:

Algorithm	Best Suited For	Reasoning
Kruskal's	Sparse graphs	Uses edge sorting and Union-Find, good for graphs with fewer edges.
Prim's	Dense graphs	Operates on vertices, efficient with binary heaps for dense graphs.
Borůvka's	Parallel computing	Merges components in parallel, useful for distributed processing.
Reverse-Delete	Edge-critical graphs	Starts with all edges and removes unnecessary ones, useful in optimization problems.
Fibonacci Heap Prim's	Extremely large graphs	Reduces priority queue operations, making it efficient for very large graphs.
Chazelle's Soft Heap	Theoretical efficiency	Achieves nearly linear time complexity ($O(E\alpha(V))$).
Approximate MST	Large-scale data	Used when exact MST is not needed, faster for real-time applications.

Comparison of MST Algorithms in Literature II

Key Observations:

- Kruskal's is best for edge-based processing, while Prim's is best for vertex-based processing.
- Borůvka's algorithm is useful for distributed computing frameworks.
- Reverse-Delete is the opposite of Kruskal's but useful in certain edge-removal optimization cases.
- Fibonacci Heap improves Prim's performance, making it optimal for large graphs.
- Chazelle's algorithm is mostly of theoretical interest due to its complexity benefits.
- Approximate MST algorithms trade accuracy for speed, useful in AI and networking.

Exercise:

- Compare the performance of **Kruskal's and Prim's algorithms** on a 10,000-node graph.
- Research how **Borůvka's Algorithm** is used in parallel computing.

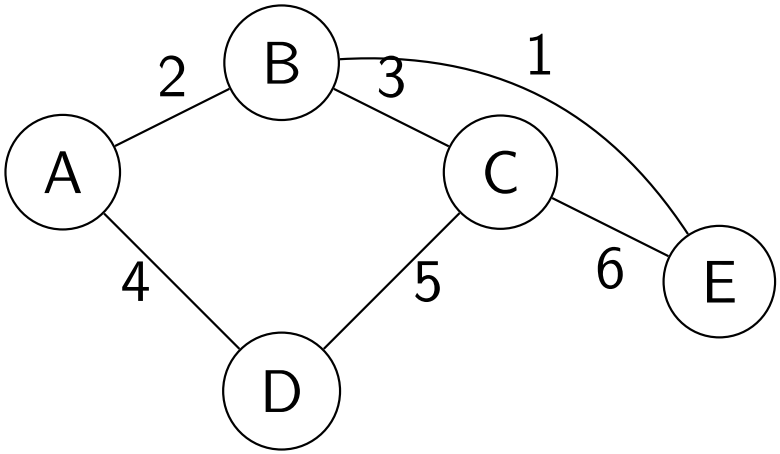
MST Applications in Network Design I

Problem: How can we efficiently design a network (e.g., internet, transportation, power grid) to connect nodes with minimal cost?
MST Solution:

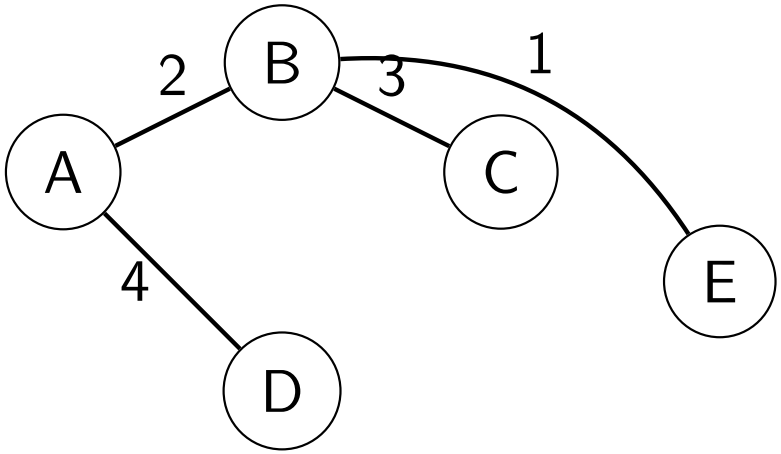
- The MST ensures **all locations (nodes) are connected with minimal wiring (edges).**
- It eliminates redundant connections while preserving connectivity.
- Examples include:
  - **Internet backbone networks** (connecting servers efficiently).
  - **Power distribution grids** (minimizing transmission cost).
  - **Railway networks** (finding the shortest network to connect cities).

MST Applications in Network Design II

Initial Network:



Optimized MST:



Exercise:

- Apply Prim's algorithm to find an MST for a city-wide **fiber optic network**.
- Research how MSTs help in designing **underground metro networks**.

MST in Clustering and Machine Learning I

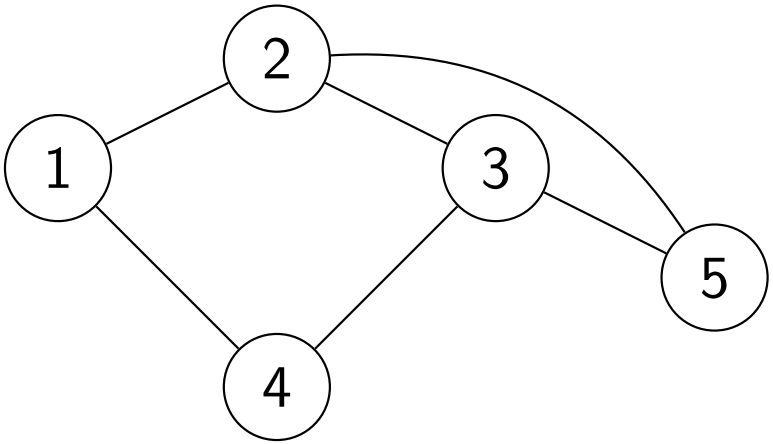
Problem: How can we group similar data points efficiently in clustering algorithms?

MST Solution:

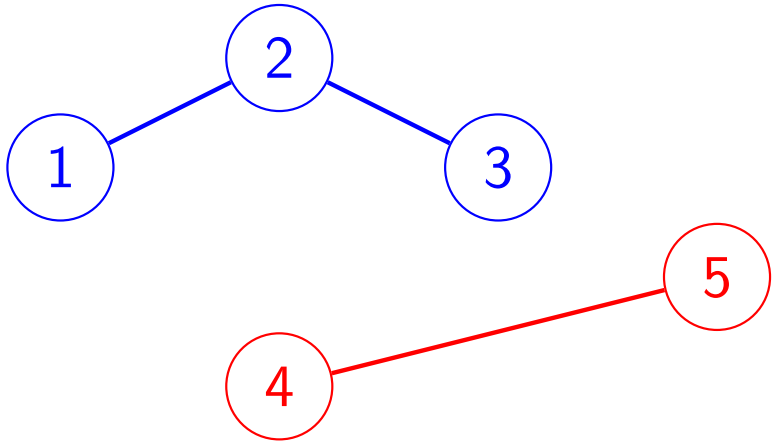
- The MST captures the most important connections in data while minimizing redundancy.
- It is widely used in **hierarchical clustering** to partition data into meaningful groups.
- **Applications in Machine Learning:**
  - Identifying clusters in high-dimensional datasets.
  - Detecting **communities in social networks**.
  - Pattern recognition and feature selection.

MST in Clustering and Machine Learning II

Dataset as a Graph:



Clusters After MST-Based Partitioning:



Exercise:

- Apply MST-based clustering to **customer segmentation in marketing**.
- Research how MST is used in **bioinformatics to classify gene sequences**.

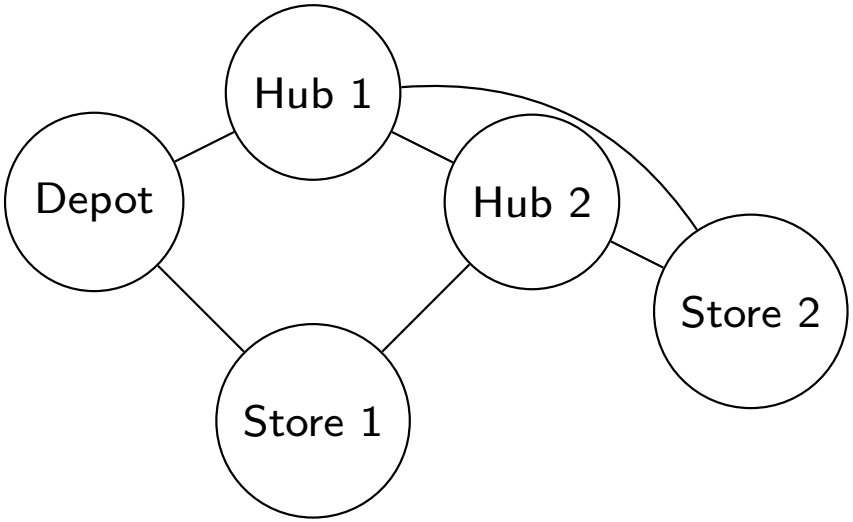
MST in Optimization and Logistics

Problem: How can we optimize transportation and supply chain logistics while minimizing cost?

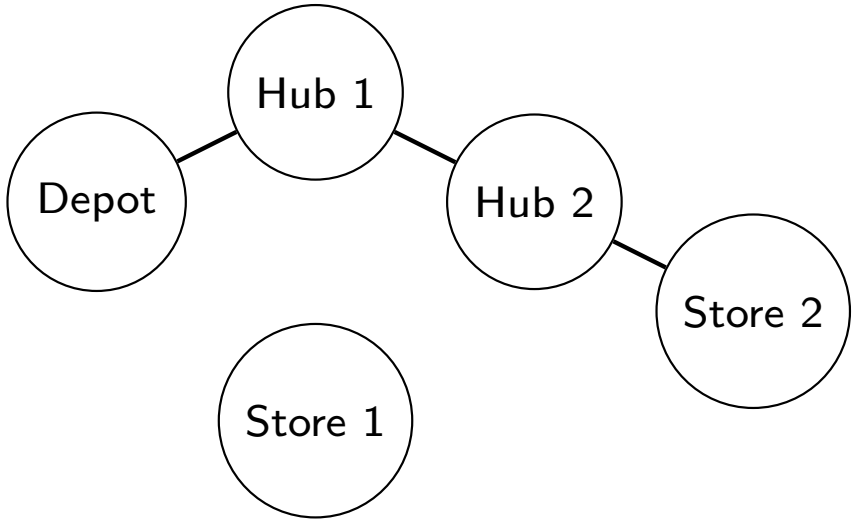
MST Solution:

- Used in **route planning** to minimize road construction costs.
- Ensures all locations are **connected with minimal resources**.
- Common in **airline routing**, **shipping networks**, and **warehouse distribution**.

Initial Routes:



Optimized Supply Chain MST:

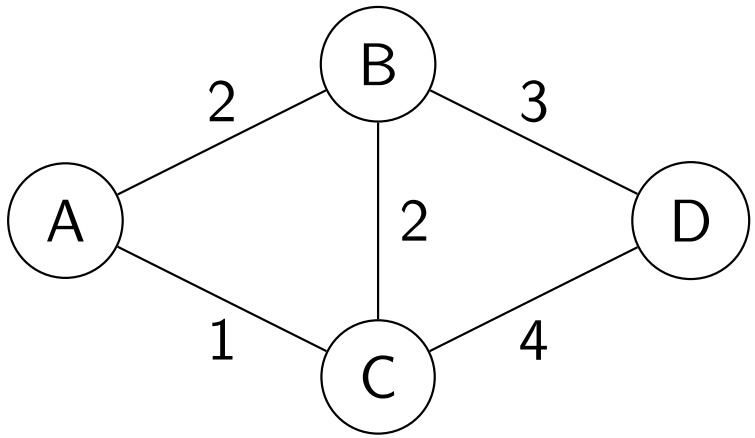


Outline

4. Appendix
   - More on Spanning Tree
   - More Tree Types
   - More
   - Handy Proofs and Results

Steiner Trees and Applications

- **Definition:** A Steiner tree in a graph connects a given subset of vertices (called terminals) with the minimum total edge weight. Non-terminal vertices may also be used to reduce weight.
- **Applications:**
  - Network design (e.g., fiber optics, electrical grids).
  - Circuit layouts in VLSI.
- **Example:**



  Problem: Find the Steiner tree for terminals A, C, and D.
- **Exercise:**
  - Compare the Steiner tree and spanning tree for the above graph.

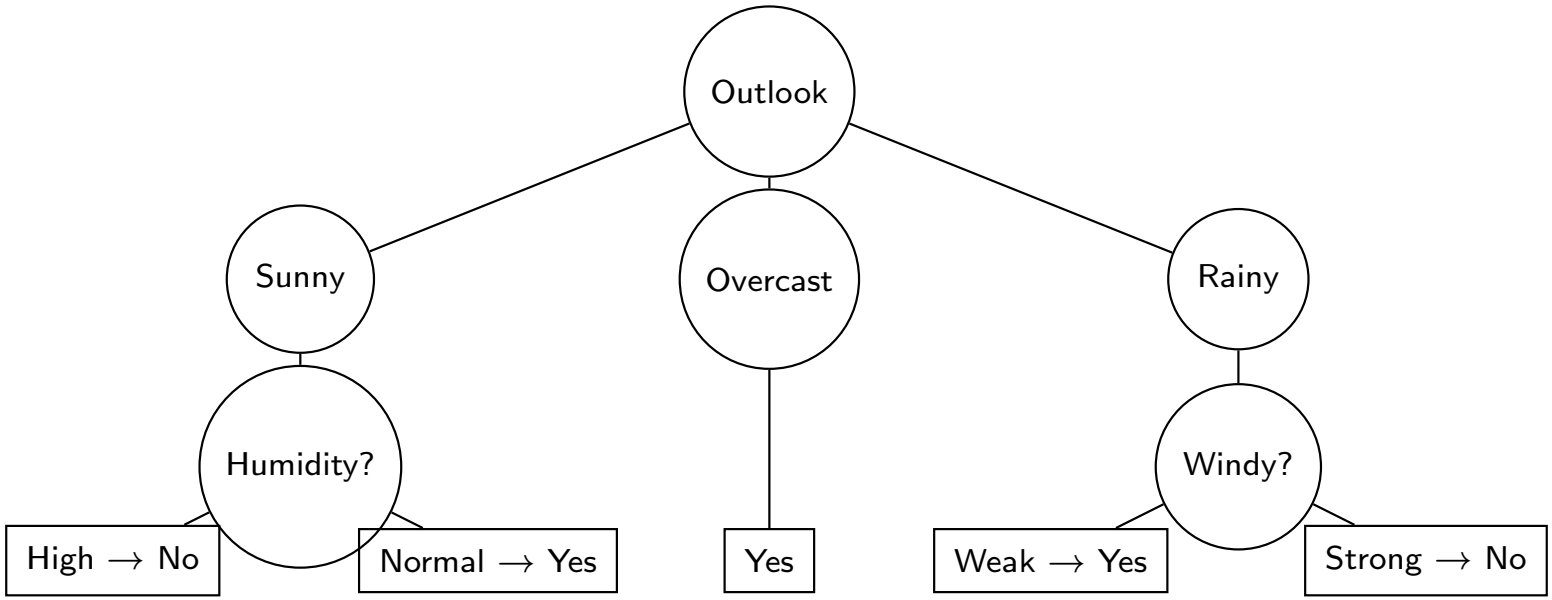
Decision Trees I

Definition: A **decision tree** is a tree-based model used in machine learning for classification and regression tasks.

Structure:

- **Root Node:** Represents the entire dataset.
- **Internal Nodes:** Decision points based on feature values.
- **Leaf Nodes:** Final class labels or predicted values.

Example: Decision Tree for Weather Prediction



Decision Trees II

Applications:

- **Medical diagnosis** (predicting diseases based on symptoms).
- **Fraud detection** (classifying transactions as fraudulent or normal).
- **Recommender systems** (predicting user preferences).

Exercise:

- Construct a decision tree for **loan approval** based on applicant age, income, and credit score.
- Explain why decision trees are **interpretable compared to neural networks**.

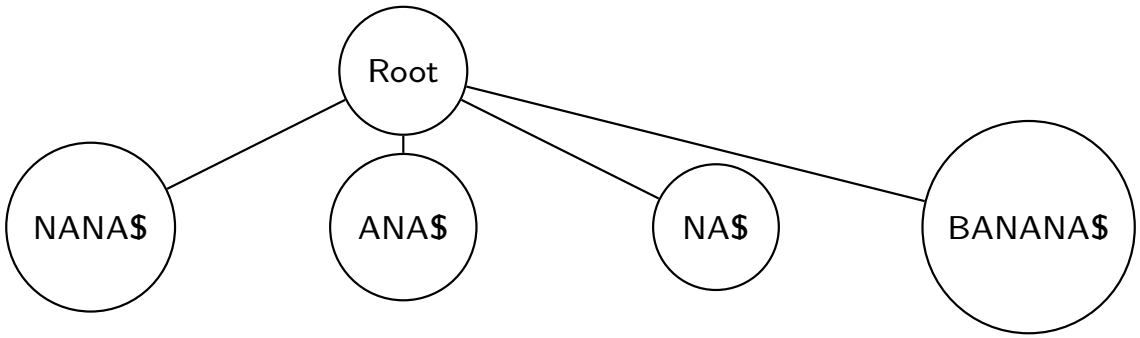
Suffix Trees I

Definition: A **suffix tree** is a compressed trie that stores all suffixes of a given string, allowing for fast substring searches.

Properties:

- The tree contains $O(n)$ **nodes** for a string of length n.
- Allows substring searches in $O(m)$ **time** for a query of length m.
- Useful for **bioinformatics, data compression, and pattern matching**.

Example: Suffix Tree for "BANANA$"



Suffix Trees II

Applications:

- **Genome sequencing** (finding repeated DNA sequences).
- **String matching problems** (longest common substring).
- **Data compression** (Burrows-Wheeler Transform).

Exercise:

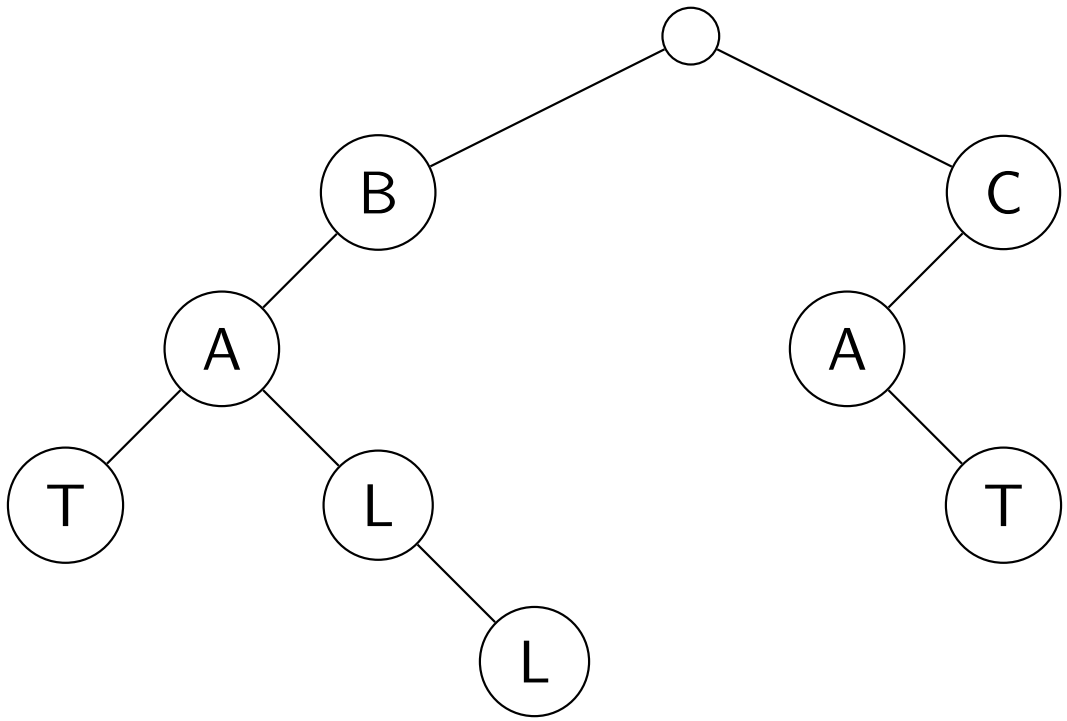
- Construct a suffix tree for **"MISSISSIPPI$"**.
- Explain why suffix trees can be built in $O(n)$ **time** using Ukkonen's algorithm.

Trie Trees (Prefix Trees) I

Definition: A **trie (prefix tree)** is a tree-based data structure for storing a set of strings where:

- Each edge represents a character.
- Each path from root to a node represents a prefix.
- Words are stored as paths from the root.

Example: Trie for "BAT", "BALL", "CAT"



Trie Trees (Prefix Trees) II

Operations and Complexity:

- **Insertion:** $O(m)$ where m is the word length.
- **Search:** $O(m)$.
- **Deletion:** $O(m)$.

Applications:

- Autocomplete and search engines.
- Spell checking and dictionary storage.
- IP routing and genome sequencing.

Exercise:

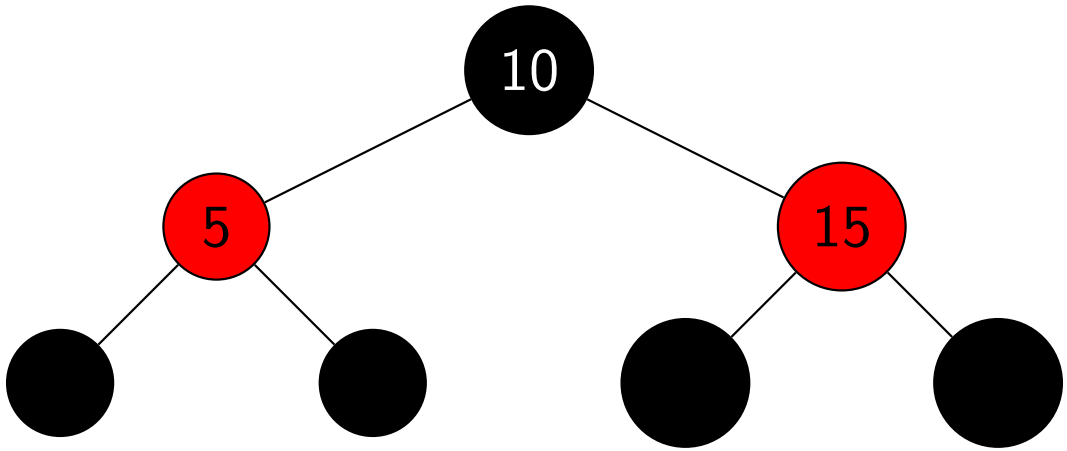
- Insert the words **"DOG", "DOOR", "DOLL"** into a trie.
- Explain why tries are more efficient than hash tables for prefix searching.

Red-Black Trees I

Definition: A **Red-Black Tree (RBT)** is a **self-balancing binary search tree (BST)** where each node has a color (red or black) and satisfies:

- The root is always black.
- No two consecutive red nodes appear.
- Every path from root to a leaf contains the same number of black nodes.
- The height of the tree is at most $2\log(n+1)$.

Example: Balanced Red-Black Tree



Red-Black Trees II

Operations and Complexity:

- **Insertion:** $O(\log n)$ (with rotations to maintain balance).
- **Deletion:** $O(\log n)$.
- **Search:** $O(\log n)$.

Applications:

- Database indexing (e.g., Linux file system ext3/ext4).
- Scheduling algorithms (OS process management).
- Efficient memory allocation in compilers.

Exercise:

- Insert elements **(10, 5, 15, 3, 7, 13, 17)** into an RBT and balance it.
- Explain why Red-Black Trees are preferred in high-frequency trading databases.

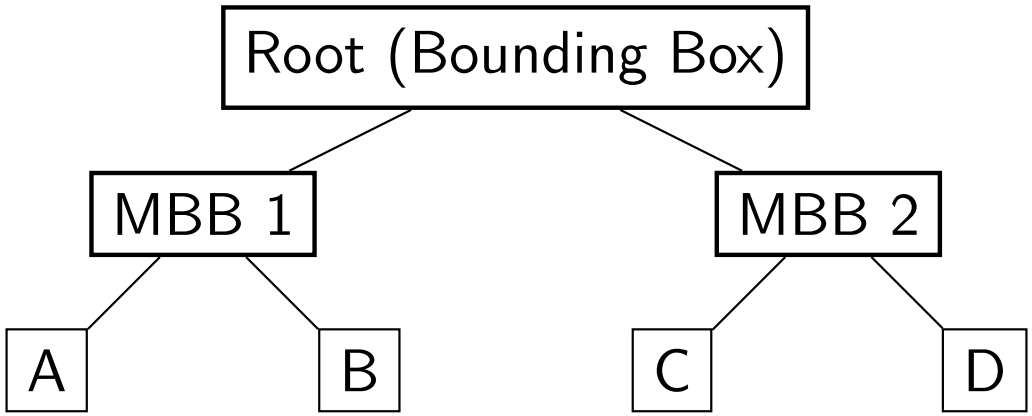
R-Trees (Spatial Data Indexing Trees) I

Definition: An **R-Tree** is a balanced tree data structure designed for indexing **spatial objects** such as rectangles, polygons, and multidimensional data.

Properties:

- Similar to B-Trees but optimized for spatial searches.
- Each node represents a **bounding box** that encloses its child nodes.
- Supports **efficient range queries and nearest neighbor searches**.
- Height is $O(\log n)$, making search operations fast.

Example: R-Tree Structure



R-Trees (Spatial Data Indexing Trees) II

Operations and Complexity:

- **Insertion:** $O(\log n)$ (adjusts bounding boxes to fit new data).
- **Search:** $O(\log n)$ (prunes irrelevant branches based on bounding boxes).
- **Deletion:** $O(\log n)$.

Applications:

- Geographic Information Systems (GIS) (mapping, location-based queries).
- Computer graphics (collision detection in gaming).
- Database indexing for spatial data (e.g., finding nearby restaurants).

Exercise:

- Construct an R-Tree for the rectangles **(1,3,4,6), (2,5,7,8), (6,2,9,5)**.
- Explain why R-Trees outperform Quad-Trees for large datasets.

Outline

4. Appendix
   - More on Spanning Tree
   - More Tree Types
   - More
   - Handy Proofs and Results

Proof of Kruskal's Algorithm Correctness

- **Claim:** Kruskal's algorithm produces a minimum spanning tree (MST).
- **Proof:**
  - Let $G = (V, E)$ be a graph with weights $w(e)$ on edges.
  - Kruskal's algorithm iteratively adds the smallest edge e that doesn't form a cycle.
  - **Greedy Choice:** Adding e minimizes the total weight because no smaller edge can replace it without breaking acyclicity.
  - **Termination:** After $|V| - 1$ edges, the graph is connected and acyclic (a tree).
  - By contradiction, assume a different spanning tree T' has a smaller weight than T. Replace an edge in T with one not in T' to show T must have been optimal.
- **Conclusion:** Kruskal's algorithm is correct.

Exercise:

- Prove Prim's algorithm correctness using similar reasoning.

Applications of Trees

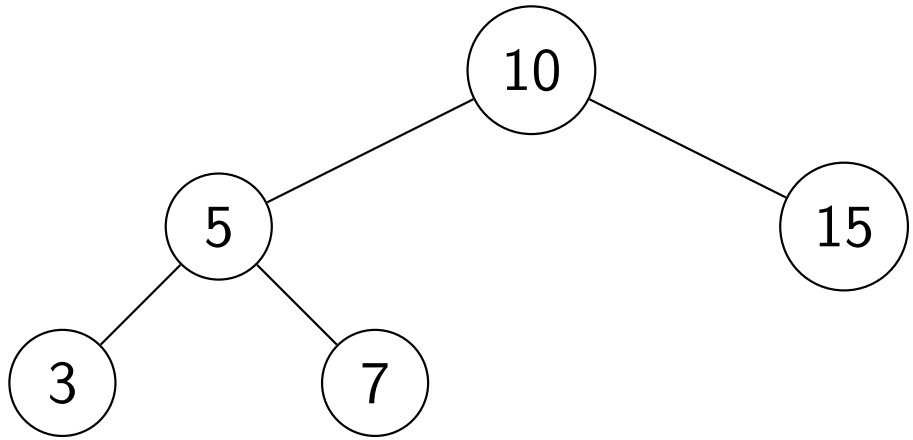
- **Computer Science:**
  - Data Structures: Binary Search Trees, AVL Trees, B-Trees.
  - Parsing: Syntax trees in compilers.
- **Networks:**
  - Spanning Trees for communication protocols.
  - Steiner Trees for cost-efficient network design.
- **Biology:**
  - Phylogenetic Trees for evolutionary relationships.
- **Mathematics:**
  - Cayley's formula and combinatorial enumeration.
- **Exercise:**
  - Research one real-world application of minimum spanning trees and present its significance.

AVL Trees

- **Definition:** An AVL tree is a self-balancing binary search tree where:

$$\text{Balance Factor} = \text{Height of Left Subtree} - \text{Height of Right Subtree}$$

  is in $\{-1, 0, 1\}$ for every node.
- **Key Operations:**
  - **Insertion** and **Deletion**: Maintain balance by performing rotations: - Single Rotations (Left or Right). - Double Rotations (Left-Right or Right-Left).
  - Search: $O(\log n)$ due to balanced height.
- **Example:**



Exercise:

- Perform rotations for inserting 12 into the AVL tree above.

AVL Tree Properties and Proofs

- **Height of AVL Tree:**
  - Let h be the height of an AVL tree with n nodes.
  - Minimum nodes for height h: $N(h) = N(h-1) + N(h-2) + 1$ (similar to Fibonacci sequence).
  - Maximum height for n nodes:

$$h = O(\log n).$$

- **Proof: Height Balance:**
  - Balance factor for any node is in $\{-1, 0, 1\}$.
  - Rotations maintain height balance during insertion or deletion.
- **Exercise:**
  - Prove: The height of an AVL tree with n nodes is bounded by $O(\log n)$.

AVL Trees - Rotations and Proofs

- **Single Rotation:**
  - Occurs when inserting into the left or right subtree of a node.
  - **Proof of Balance:** After rotation, height difference is at most 1.
- **Double Rotation:**
  - Occurs when inserting into the "inside" subtree.
  - **Proof of Balance:** After rotation, subtrees are balanced due to height adjustments.
- **Height Complexity:**
  - Maximum height of an AVL tree with n nodes: $O(\log n)$.
  - Proof uses recurrence relation:

$$h(n) = 1 + \max(h(k), h(n-k-1)) \quad \text{for } k < n.$$

- **Exercise:**
  - Prove the maximum height of an AVL tree for $n = 15$.

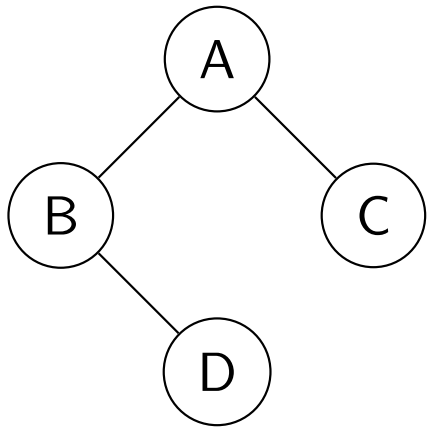
Introduction to Graph Coloring

- **Definition:** Graph coloring is the assignment of labels (colors) to elements of a graph subject to certain constraints.
- **Types of Coloring:**
  - **Vertex Coloring:** Adjacent vertices must have different colors.
  - **Edge Coloring:** Adjacent edges must have different colors.
  - **Face Coloring:** Used in planar graphs, adjacent faces must have different colors.
- **Applications:**
  - Scheduling problems.
  - Register allocation in compilers.
- **Theorems and Algorithms:**
  - Four-Color Theorem for planar graphs.
  - Greedy coloring algorithms.

Note: This topic will be elaborated further in Module 3.

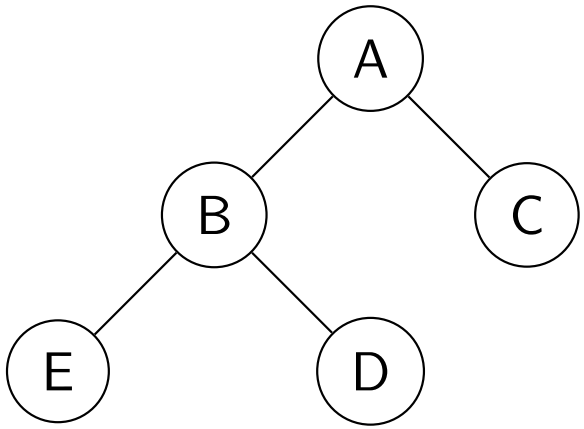
Graph Coloring in Trees

- **Vertex Coloring:** Assign colors to vertices so that no two adjacent vertices have the same color.
- **Theorem:** Every tree is bipartite.
- **Proof:** (By Induction)
  - Base Case: A single vertex is trivially bipartite.
  - Inductive Hypothesis: Assume a tree of k vertices is bipartite.
  - Inductive Step: Adding a vertex connected to the tree preserves bipartiteness by coloring it with the opposite color of its parent.
- **Exercise:**
  - Verify bipartiteness for the following tree:



Dynamic Programming on Trees

- **Problem: Diameter of a Tree**
  - Diameter = Longest path between any two vertices.
- **Approach:**
  1. Pick an arbitrary node and find the farthest node u using DFS.
  2. From u, find the farthest node v. Path between u and v is the diameter.
- **Time Complexity:** $O(V + E)$.
- **Exercise:**
  - Implement the above algorithm for the tree below:



Dynamic Programming on Trees - Additional Algorithms

- **Algorithms:**
  - **Tree Diameter**: Longest path in a tree using two DFS traversals.
  - **Tree Center**: A node minimizing the maximum distance to any other node.
  - **Subtree Queries**: Sum, maximum, or counts in subtrees.
- **Best Complexity for Tree Algorithms:**
  - Many problems can be solved in $O(n)$ using efficient dynamic programming.
  - Examples:
    - ⋆ Diameter of a Tree: $O(n)$.
    - ⋆ Subtree Sums: $O(n)$ preprocessing, $O(1)$ per query.
- **Exercise:**
  - Research the "Heavy-Light Decomposition" technique and its applications.

Tree Diameter - Related Algorithms and Complexity

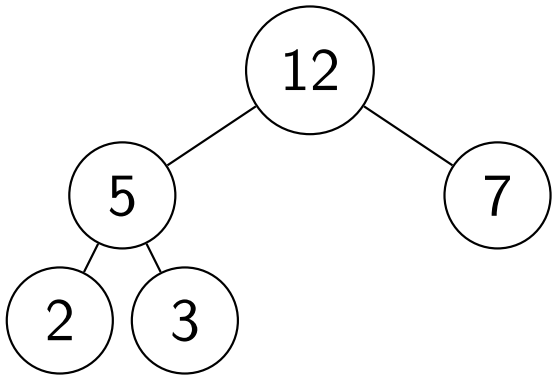
- **Problem:** Find the diameter of a tree (longest path between any two nodes).
- **Algorithms:**
  - **Naive BFS/DFS:** Perform two BFS/DFS operations:
    - ★ Start from any node to find the farthest node u.
    - ★ From u, find the farthest node v. Path $u \leftrightarrow v$ is the diameter.
    - ★ Time Complexity: $O(V + E)$.
  - **Dynamic Programming on Trees:**
    - ★ Divide-and-conquer method.
    - ★ Better for trees with complex weight systems.
    - ★ Time Complexity: $O(V)$.
  - **Centroid Decomposition:** Optimizes path queries for tree diameters. Used for dynamic tree scenarios.
- **State-of-the-Art Complexity:**
  - The best-known algorithm achieves $O(V)$ for static trees.

Exercise:

- Compare BFS and DP approaches for finding the diameter of a given tree.

Huffman Trees

- **Definition:** A binary tree used for lossless data compression.
- **Algorithm:**
  1. Start with a forest of single-node trees, each weighted by the frequency of the symbol.
  2. Combine the two trees with the smallest weights into a new tree, where the root's weight is the sum of its children.
  3. Repeat until only one tree remains.
- **Example:**

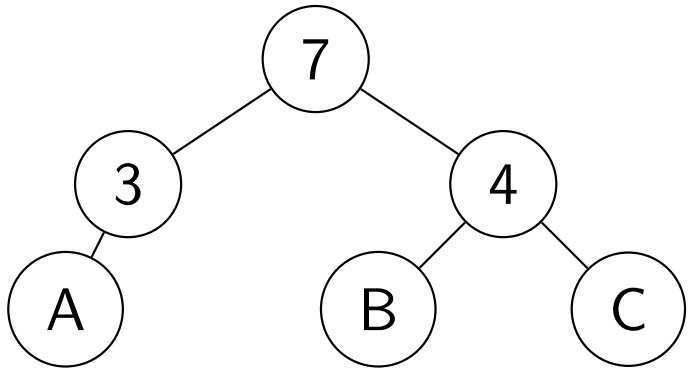


Exercise:

- Build a Huffman tree for the symbols A, B, C, D with frequencies 4, 3, 2, 1.

Huffman Trees - Case Study

- **Use Case: Data Compression**
  - Used in formats like ZIP files and image compression (JPEG).
- **Example: Text Compression**
  - Given text: "AAABBCC".
  - Frequencies: $A:3, B:2, C:2$.
  - Huffman tree:



- **Compression Efficiency:**
  - Original: 7 characters $\times 8$ bits $= 56$ bits.
  - Huffman Encoded: 16 bits (compressed).

Exercise:

- Apply Huffman encoding to "HELLO WORLD".

Huffman Coding and Applications

- **Real-World Utility:**
  - Data compression for text and images.
  - Basis for ZIP file compression.
- **Case Study:**
  - **Scenario:** Compressing text files with varying character frequencies.
  - **Example:** Consider the string ‘ABRACADABRA‘: - Character frequencies: A (5), B (2), R (2), C (1), D (1). - Huffman encoding reduces average bits per character from 3 (fixed encoding) to 2.36. - Savings: 21
  - Application in real-world file systems: - PNG image compression uses Huffman coding.
- **Exercise:**
  - Design a Huffman tree for ‘AAABBCCCCD‘ and calculate average bits per character.

Outline

4 Appendix
- More on Spanning Tree
- More Tree Types
- More
- Handy Proofs and Results

Proof: Average Degree in Trees

Statement: Every tree with average degree a has $\frac{2}{2-a}$ vertices.
Proof Sketch:

- A tree with n vertices has exactly $n-1$ edges.
- The sum of the degrees in a tree is $2(n-1)$.
- The average degree is $\frac{2(n-1)}{n}$.
- Setting this equal to a and solving for n gives $n = \frac{2}{2-a}$.

Proof: Number of Cycles in an n-Vertex Graph

Statement: Every n-vertex graph with m edges has at least $m - n + 1$ cycles.

Proof Sketch:

- A spanning tree of G has $n - 1$ edges.
- The remaining $m - (n - 1) = m - n + 1$ edges form cycles.
- Each extra edge must close a cycle.

Proof: Complement Graph Diameter

Statement: If a simple graph has diameter at least 4, its complement has diameter at most 2.

Proof Sketch:

- A path of length at least 4 means certain vertex pairs are far apart.
- In the complement, any such pair must have a short path.
- Thus, the complement has diameter at most 2.

Proof: Diameter of Graph Square

Statement: The square of a connected graph G has diameter $\lceil \text{diam}(G)/2 \rceil$.

Proof Sketch:

- In G^2, each vertex is connected to neighbors up to distance 2.
- The longest shortest path in G is halved in G^2.
- Thus, the new diameter is $\lceil \text{diam}(G)/2 \rceil$.

Proof: Degree List of a Tree

Statement: A tree with degree list $k, k-1, \ldots, 2, 1, 1, \ldots, 1$ has $2+k$ vertices.

Proof Sketch:

- The sum of degrees in a tree is $2(n-1)$.
- Solving for n with given degrees confirms $n = 2 + k$.

Proof: Leaf in Larger Partite Set

Statement: Every tree has a leaf in its larger partite set (in both if they have equal size).

Proof Sketch:

- A tree is bipartite, meaning its vertex set can be partitioned into two independent sets.
- The sum of degrees equals $2(n-1)$, so one partite set must have a larger sum of degrees.
- The partite set with more vertices must contain at least one leaf.

Proof: Odd Degree Vertices in Trees

Statement: The vertices of a tree T all have odd degree if and only if for all $e \in E(T)$, both components of $T - e$ have odd order.

Proof Sketch:

- Each edge removal splits T into two components.
- If all vertices had odd degree, then each component must contain an odd number of vertices.
- Conversely, if every component after edge removal has odd order, then all original vertices must have had odd degree.

Proof: Spanning Tree with Diameter k

Statement: If G has a tree with diameter 2, then G has a spanning tree with diameter k for $k < l$, where l is the number of vertices.

Proof Sketch:

- If G contains a tree with diameter 2, all vertices are within distance 2 of each other.
- We can construct a spanning tree by strategically removing edges to maintain connectivity.
- The resulting spanning tree will have a smaller diameter.

Proof: Isomorphism Classes of Trees

Statement: For $n \geq 2$, the number of isomorphism classes of n-vertex trees with diameter at most 3 is $\lfloor n/2 \rfloor$.

Proof Sketch:

- Trees with diameter at most 3 are characterized by their center and leaf distribution.
- Each isomorphism class corresponds to a unique partitioning of the vertex set.
- The number of such partitions is $\lfloor n/2 \rfloor$.

Proof: Intersecting Subtrees

Statement: If $G_1, \ldots, G_k$ are pairwise intersecting subtrees of a tree G, then G has a vertex in all of $G_1, \ldots, G_k$. **Proof Sketch:**

- Assume for contradiction no single vertex belongs to all subtrees.
- Since each pair intersects, we can construct a connected sequence of overlaps.
- A common vertex must exist due to the tree structure.

Proof: Characterizing Forests

Statement: A simple graph G is a forest if and only if pairwise intersecting paths in G always have a common vertex. **Proof Sketch:**

- In a tree, cycles cannot exist, so intersecting paths must meet at a common vertex.
- If a graph is not a forest, a cycle exists, contradicting the property of pairwise intersecting paths.

Proof: Orientation of Trees

Statement: If T is an orientation of a tree such that the heads of the edges are all distinct, then T is a union of paths from the root (the one vertex that is not a head), and each vertex is reached by one path from the root.

Proof Sketch:

- Since every vertex except one is the head of exactly one edge, there is a unique path from the root.
- The structure remains a tree, ensuring each vertex is reached by exactly one path.

Independent Set in Graphs

Statement: Every graph with diameter d has an independent set of size at least $\left\lfloor \frac{1+d}{2} \right\rfloor$.

Proof Sketch: Consider the longest path in the graph, which has length d. Select every second vertex on this path to form an independent set. The number of vertices selected is at least $\left\lfloor \frac{1+d}{2} \right\rfloor$.

Spanning Trees in Complete Bipartite Graphs

Statement: $K_{n,n}$ has n^{2n-2} spanning trees.
Proof Sketch: Using the Matrix-Tree Theorem, the number of spanning trees of $K_{n,n}$ is given by n^{2n-2}.

Graceful Labeling and Decompositions

Statement: If a graph G with m edges has a graceful labeling, then K_{2m+1} decomposes into copies of G.
Proof Sketch: A graceful labeling of G ensures that the edges are labeled uniquely from 1 to m. This labeling can be extended to decompose K_{2m+1} into copies of G.

Orientation of Trees

Statement: If T is an orientation of a tree such that the heads of the edges are all distinct, then T is a union of paths from the root.
Proof Sketch: Since the heads of the edges are distinct, each vertex (except the root) has exactly one incoming edge. This structure forms a union of paths originating from the root.

Integer Weighting and Bipartiteness

Statement: In an integer weighting of the edges of K_n, the total weight is even on every cycle if and only if the subgraph consisting of the edges with odd weight is a spanning complete bipartite subgraph.
Proof Sketch: If the total weight is even on every cycle, the edges with odd weights must form a bipartite graph to avoid odd cycles. Conversely, if the subgraph of odd-weight edges is bipartite, every cycle in the original graph has an even total weight.

Existence of Trees with Given Degrees

Statement: For $n \geq 2$, there exists a tree with vertex degrees $d_1, \ldots, d_n$ if and only if $d_n = 1$ and the sum of d_i is $2(n-1)$.
Proof Sketch: A tree with n vertices has $n-1$ edges, and the sum of the degrees of all vertices is $2(n-1)$. For the tree to exist, one vertex must be a leaf ($d_n = 1$).

Bipartiteness of Trees

Statement: Every tree is bipartite.
Proof Sketch: A tree is a connected acyclic graph. To prove it is bipartite, we can use a 2-coloring argument. Start at any vertex and color it with one color (say, red). Color all its neighbors with the other color (blue). Continue this process, alternating colors for each level of neighbors. Since there are no cycles, no two adjacent vertices will have the same color, proving the tree is bipartite.

Spanning Trees in Connected Graphs

Statement: Every connected graph has a spanning tree.
Proof Sketch: A spanning tree of a connected graph G is a subgraph that includes all the vertices of G and is a tree. To construct a spanning tree, start with any vertex and perform a depth-first or breadth-first search, adding edges to the spanning tree until all vertices are included. The resulting subgraph is connected and acyclic, hence a spanning tree.

Graphs with Minimum Degree 2

Statement: If every vertex of a graph G has degree at least two, then G contains a cycle.

Proof Sketch: Assume G is connected and every vertex has degree at least 2. Start at any vertex and perform a walk, always moving to a new vertex. Since each vertex has at least two neighbors, you will eventually revisit a vertex, forming a cycle.

Trees and Cycles

Statement: The graph obtained from a tree by adding an edge has exactly one cycle.

Proof Sketch: Let T be a tree with n vertices and $n-1$ edges. Adding one edge to T creates a graph with n vertices and n edges. The added edge creates exactly one cycle, as it connects two vertices that were previously connected by a unique path in the tree.

Trees and Cut-Vertices

Statement: Every tree has a vertex that is not a cut-vertex.
Proof Sketch: A cut-vertex is a vertex whose removal increases the number of connected components of the graph. In a tree, any leaf is not a cut-vertex, as its removal does not disconnect the graph.

Tree Properties

Statement: If a tree has at least two vertices, then it has at least two vertices that are not cut-vertices.
Proof Sketch: A tree with at least two vertices must have at least two leaves. Leaves are not cut-vertices, so the tree has at least two vertices that are not cut-vertices.

Cut-Vertices in Trees

Statement: A vertex of a tree is a cut-vertex if and only if it has degree at least 2.

Proof Sketch: In a tree, a vertex with degree 1 (a leaf) is not a cut-vertex. A vertex with degree at least 2 is a cut-vertex if its removal disconnects the graph.

Planarity of Graphs

Statement: Every graph with no cycle is planar.
Proof Sketch: A graph with no cycles is a forest, which is a collection of trees. Trees are planar graphs, so a forest is also planar.

Diameter of a Graph

Statement: If G has a tree with diameter 2, then G has a spanning tree with diameter k for all k less than the diameter of G.
Proof Sketch: If G has a tree with diameter 2, it means there is a spanning tree with diameter 2. For any k less than the diameter of G, we can find a spanning tree with diameter k by appropriately selecting edges.

Complete Graphs as Trees

Statement: The complete graph K_n is a tree only when $n = 1$ or 2.
Proof Sketch: A tree is a connected acyclic graph. K_1 and K_2 are trees, but for $n \geq 3$, K_n contains cycles and is not a tree.

Complement of a Tree

Statement: The complement of a tree is never a tree for $n > 4$.
Proof Sketch: The complement of a tree with n vertices has $\binom{n}{2} - (n-1)$ edges. For $n > 4$, this number is greater than $n - 1$, meaning the complement graph has more edges than vertices minus one, which implies it contains cycles and is not a tree.

Graceful Graphs

Statement: The complete graph K_n is graceful only when $n = 1, 2, 3, 4$.
Proof Sketch: A graph is graceful if it can be labeled such that the edge labels (absolute differences of vertex labels) are all distinct. It is known that K_n is graceful only for $n = 1, 2, 3, 4$.

Cycles and Gracefulness

Statement: The cycle graph C_n is graceful only when $n = 3$ or 4.
Proof Sketch: A cycle graph C_n is graceful if it can be labeled such that the edge labels are all distinct. It is known that C_n is graceful only for $n = 3$ or 4.

Path Graphs and Gracefulness

Statement: The path graph P_n is graceful.
Proof Sketch: A path graph P_n can be labeled gracefully by assigning labels $0, 1, 2, \ldots, n-1$ to the vertices in order. The edge labels will be $1, 1, 1, \ldots, 1$, which are distinct.

Cycles and Gracefulness

Statement: If G has a graceful labeling then G has no cycles.
Proof Sketch: A graceful labeling requires that the edge labels (absolute differences of vertex labels) are all distinct. If G had a cycle, it would be impossible to assign distinct edge labels, as the sum of the edge labels around the cycle would need to be zero, which is not possible with distinct positive integers.

Bipartite Graphs and Hypercubes

Statement: Every bipartite graph is a spanning subgraph of some hypercube.

Proof Sketch: A bipartite graph can be embedded into a hypercube by mapping the two partite sets to the two halves of the hypercube. Each edge in the bipartite graph corresponds to an edge in the hypercube.

Spanning Trees and Cycles

Statement: Given a simple graph and a spanning tree T, any edge not in T will produce a cycle by being added to T.
Proof Sketch: A spanning tree T is a connected acyclic subgraph. Adding any edge not in T will connect two vertices already connected by a path in T, creating a cycle.

Cut-Vertices and Cycles

Statement: If G is a simple graph with n vertices and $n-1$ edges that contains a cycle then G has a cut-vertex.
Proof Sketch: If G has n vertices and $n-1$ edges, it is a tree. If it contains a cycle, removing any edge in the cycle will disconnect the graph, making the vertices incident to the removed edge cut-vertices.

Cut-Vertices in Trees

Statement: A tree with n vertices has at least $n/2$ vertices that are not cut-vertices.

Proof Sketch: In a tree, leaves are not cut-vertices. A tree with n vertices has at least $n/2$ leaves, ensuring that at least $n/2$ vertices are not cut-vertices.

Leaves and Cut-Vertices

Statement: The vertices of degree one in a graph are not cut-vertices.
Proof Sketch: A vertex of degree one is a leaf. Removing a leaf does not disconnect the graph, so leaves are not cut-vertices.

Spanning Trees and Edges

Statement: Given a simple graph with n vertices, any spanning tree will have $n - 1$ edges.
Proof Sketch: A spanning tree of a graph with n vertices is a connected acyclic subgraph that includes all vertices. By definition, it has $n - 1$ edges.

Leaves in Trees

Statement: Every tree with at least 2 vertices has at least two vertices with degree 1.
Proof Sketch: A tree with at least 2 vertices must have at least two leaves. If it had only one leaf, removing it would leave a single vertex, contradicting the assumption that the tree has at least 2 vertices.

Spanning Forests

Statement: Every graph has a spanning forest.
Proof Sketch: A spanning forest is a collection of spanning trees, one for each connected component of the graph. Every graph can be decomposed into its connected components, each of which has a spanning tree.

Hypercube Properties

Statement: The hypercube Q_k has a spanning tree with a vertex of degree k.
Proof Sketch: The hypercube Q_k is a k-dimensional cube. It has a spanning tree where one vertex (the root) has degree k, and each of its k neighbors has degree $k-1$, and so on.

Hypercube Decomposition

Statement: The n-cube can be decomposed into n spanning trees.
Proof Sketch: The n-cube Q_n can be decomposed into n spanning trees by considering the n different dimensions. Each spanning tree corresponds to fixing one dimension and varying the others.

Spanning Trees in Connected Graphs

Statement: Every connected graph has a spanning tree.
Proof Sketch: A spanning tree of a connected graph G is a subgraph that includes all the vertices of G and is a tree. To construct a spanning tree, start with any vertex and perform a depth-first or breadth-first search, adding edges to the spanning tree until all vertices are included. The resulting subgraph is connected and acyclic, hence a spanning tree.

Caterpillar Graphs

Statement: The path P_n is a caterpillar.
Proof Sketch: A caterpillar is a tree in which all the vertices are within one edge of a central path. The path P_n itself is a central path, making it a caterpillar.

Star Graphs as Caterpillars

Statement: The star graph $K_{1,n}$ is a caterpillar.
Proof Sketch: A star graph $K_{1,n}$ has a central vertex connected to n leaves. The central vertex and any one of the leaves form a central path, making $K_{1,n}$ a caterpillar.

Orientations of Trees

Statement: There is an orientation of every tree such that the heads of the edges are all distinct.
Proof Sketch: Orient the tree by performing a depth-first search from any root vertex. Direct each edge away from the root. This ensures that the heads of the edges are distinct.

Bridges and Cut-Vertices

Statement: If a connected graph has a bridge then it has at least two vertices that are not cut-vertices.
Proof Sketch: A bridge is an edge whose removal disconnects the graph. The endpoints of the bridge are not cut-vertices, as their removal does not disconnect the graph.

Bridges in Trees

Statement: Every tree has at least one edge that is a bridge.
Proof Sketch: In a tree, every edge is a bridge. Removing any edge from a tree disconnects the graph, as trees are acyclic and connected.

Spanning Trees in Graphs

Statement: Every graph has a spanning tree.
Proof Sketch: A spanning tree of a connected graph G is a subgraph that includes all the vertices of G and is a tree. To construct a spanning tree, start with any vertex and perform a depth-first or breadth-first search, adding edges to the spanning tree until all vertices are included. The resulting subgraph is connected and acyclic, hence a spanning tree.

Leaves in Trees

Statement: Every tree has at least two leaves.
Proof Sketch: A tree with at least two vertices must have at least two leaves. If a tree had only one leaf, removing it would leave a single vertex, contradicting the assumption that the tree has at least two vertices.

Distance in Graphs

Statement: If a connected graph has n vertices and m edges, then the sum of distances between all vertices is at least $\frac{n(n-1)}{2}$.
Proof Sketch: In a connected graph, the minimum distance between any two vertices is 1. The sum of distances between all pairs of vertices is at least the number of pairs, which is $\binom{n}{2} = \frac{n(n-1)}{2}$.