Data structures summarization

Lecture 1: Data Structures and Algorithms

Key Concepts

1. **Data Structures**:
   - **Definition**: Organization of data for efficient usage.
   - **Example**: Array - collection of data types/items stored at contiguous memory locations.
   - **Benefits**: Easier to locate objects due to indexing.
   - **Weaknesses**: Requires contiguous memory block, difficult to update size at runtime, costly insertion and deletion at arbitrary locations.
2. **Algorithms**:
   - **Definition**: A solution method for an algorithmic problem.
   - **Presentation**: In words, flow chart, pseudocode, code in Java, C, C++.
   - **Requirements**: Correctness, concrete steps, unambiguity, finite number of steps, must end.
3. **Object-Oriented Programming (OOP)**:
   - **Benefits**: Organized and structured code, eliminates spaghetti code, reusability of components, simpler interface, reduced complexity.
   - **Key Concepts**: Encapsulation, Abstraction, Inheritance, Polymorphism.
4. **Abstract Data Types (ADT)**:
   - **Definition**: Specifies the type of data stored, operations supported, and types of parameters of the operations.
   - **Implementation**: Realized by a class in C++, containing data members and function members (constructors and destructors).
   - **Example:** An *array* is a collection of objects of the same type:
     - Stores required amount of elements of a specific data type
     - Inserts or modifies the elements in a given position
     - Reads elements at cerain position

        - Supports logical operations like sorting

5. **Algorithm Efficiency**:
    - **Analysis**: *Efficiency* (complexity) is how well computer resources are used.
    - **Types**:
        - *Time complexity* (how long code takes to run).
        - *Space complexity* (how much storage space is needed).
    - **Measurement**:
        - *Experimental studies* (running programs and measuring time).
        - *Theoretical analysis* (determining factors affecting execution time).
6. **Running Time**:
    - Running time grows with the input size.
    - Average case time is often difficult => focus on the wordt case running time
        - Easier to analyze
        - Crucial to applications such as games, finance and robotics
7. **Estemating Running Time**:

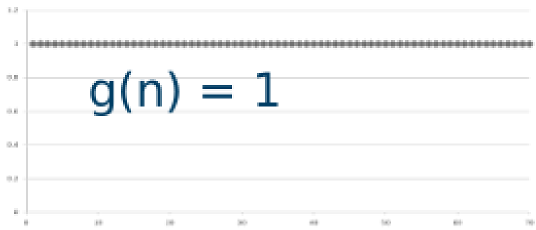
```
Algorithm arrayMax(A, n)
  currentMax   A[0]                     # operations
  for i = 1 to n - 1 do                 # 1
    if A[i] > currentMax then           # n
      currentMax = A[i]                 # n
      i =  i + 1                        # n
  return currentMax                     # 1 >= n

Total: 4n + 2
```

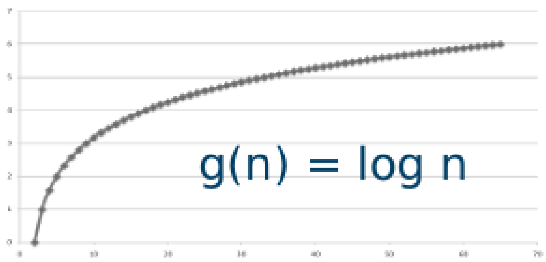
    - Algorithm arrayMax executes 4n + 2 primitive operations in the worst case.
        - Define:
            - a = time taken by the fastest primitive operation
            - b = time taken by the slowest primitive operation
        - T(n) worst-case time of arrayMax: $a(4n+2) \leq T(n) \leq b(4n+2)$
        - Running time T(n) is bounded by two linear functions

8. **Seven Important Functions**:

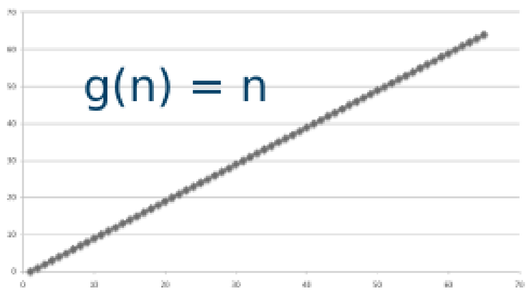
9. - Constant: 1



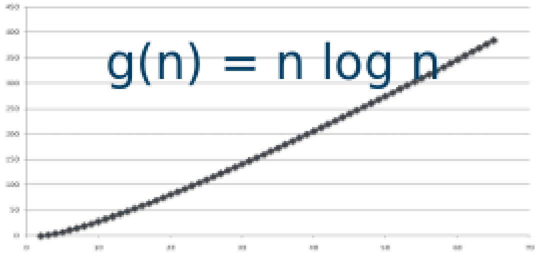
- Logarithmic: $logn$



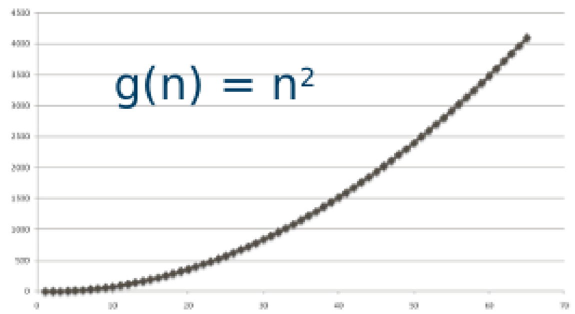
- Linear: n



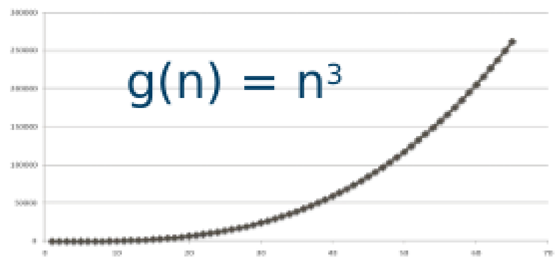
- N-Log-N: $nlogn$



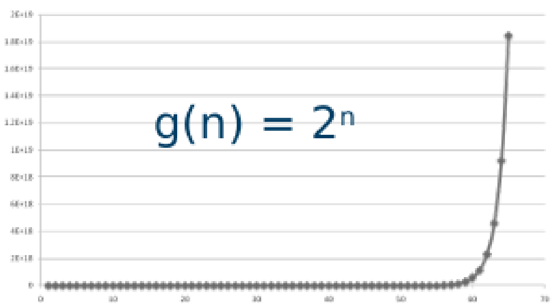
- Quadratic: n^2

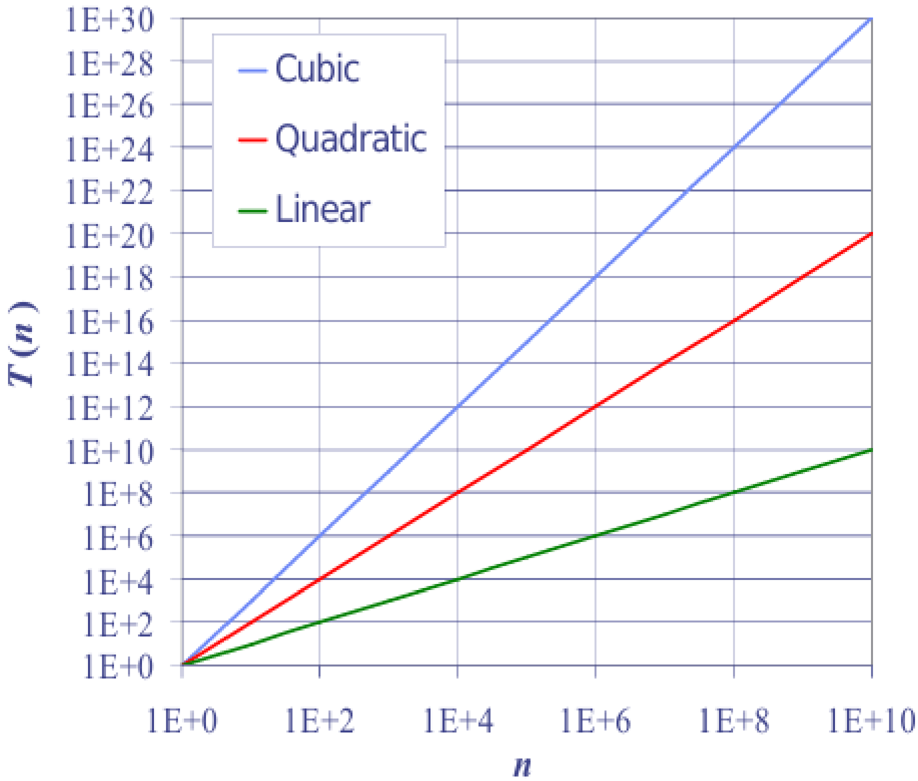


- Cubic: n^3



- Exponential: 2^n





10. **Big O Notation**:
    - **Definition**: Describes the upper bound on the growth rate of a function.
        - $f(n) \leq cg(n)$ for $n \geq n_0$
    - **Examples**:
        - 2n + 10 is O(n)
            - $2n + 10 \leq cn$
            - $10 \leq cn - 2n$
            - $n(c - 2) \geq 10$
            - $n \geq 10/(c - 2)$
            - Pick $c = 3$ and $n_0 = 10$
        - n^2 is not O(n)
            - $n^2 \leq cn$
            - $n \leq c$
            - c must be a constant
        - 7n - 2 is O(n)
            - $c > 3$ and $n_0 \geq 1$

            - $7n - 2 \leq cn$
            - Pick $c = 7$ and $n_0 = 1$
        - $3n^3 + 20n^2 + 5$ is $O(n^3)$
        - $3 \log n + 5$ is $O(\log n)$

11. **Asymptotic Algorithm Analysis**:
    - **Purpose**: Determines the running time in Big O notation.
    - **Process**: Find the worst-case number of primitive operations executed as a function of input size and express it with Big O notation.
12. **Prefix Averages**:
    - **Quadratic Time Algorithm**: Computes prefix averages in $O(n^2)$ time.
    - **Linear Time Algorithm**: Computes prefix averages in $O(n)$ time by keeping a running sum.

Additional Concepts

- **Summations, Logarithms, and Exponents**: Important mathematical concepts for algorithm analysis.
- **Relatives of Big O**:
    - **Big Omega (Ω)**: lower bound.
        - $c > 0$ and $n_0 \geq 1$
        - $f(n) \geq c \cdot g(n)$
    - **Big Theta (Θ)**: tightest bound.
        - $c' > 0$ and $c'' > 0$ and $n_0 \geq 1$
        - $c' \cdot g(n) \leq f(n) \leq c'' \cdot g(n)$

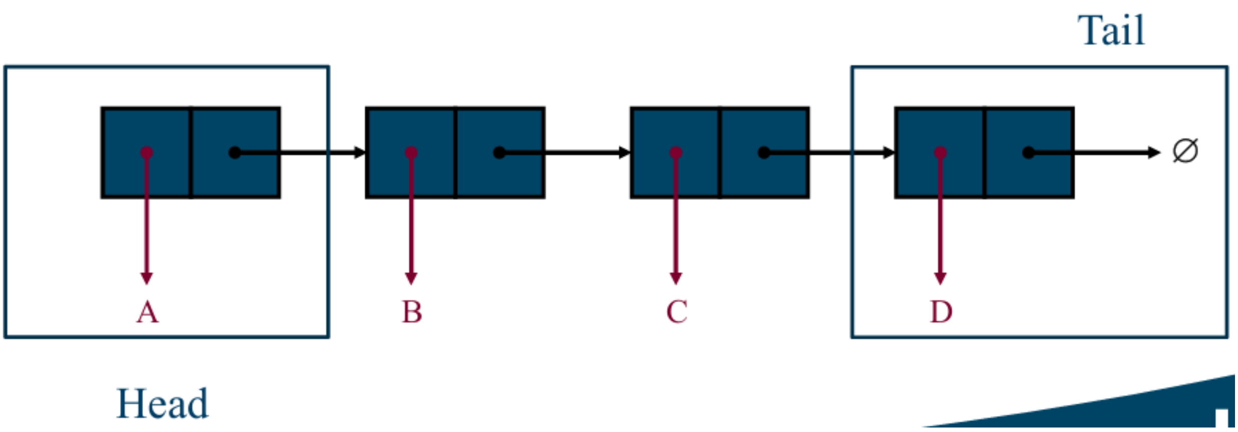
Lecture 2: list-Based Collections

Key Concepts

1. **Vector or Array List ADT**:
    - **Definition**: Extends the notion of an array by storing a sequence of objects.

- **Operations**:
    - `at(i)` : Returns the element at index `i` without removing it.
    - `set(i, o)` : Replaces the element at index `i` with `o` .
    - `insert(i, o)` : Inserts a new element `o` to have index `i` .
    - `erase(i)` : Removes the element at index `i` .
    - Additional methods: `size()` , `empty()` .
- **Applications**:
    - Direct: Sorted collection of objects (elementary database).
    - Indirect: Auxiliary data structure for algorithms, component of other data structures.
- **Performance**:
    - `size` , `empty` , `at` , and `set` run in **O(1)** time.
    - `insert` and `erase` run in **O(n)** time in the worst case.

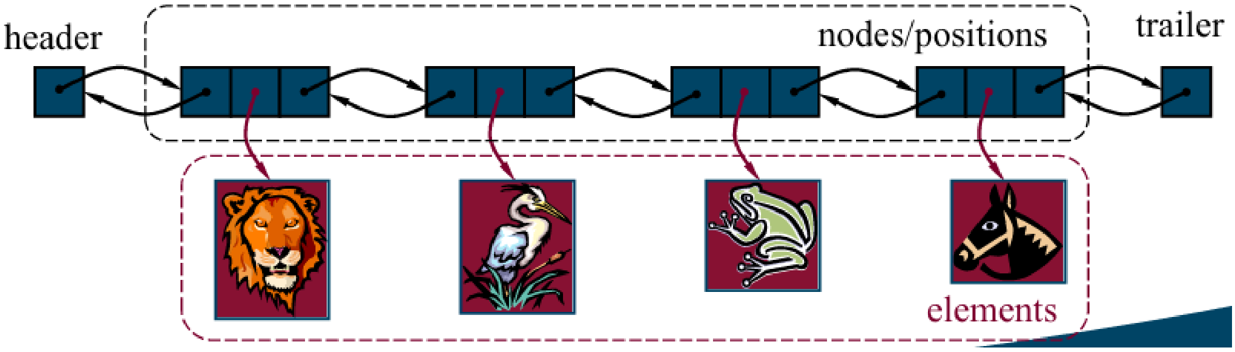
2. **Growable Array-Based Array List**:
    - **Insertion**: Always inserts at the end. When the array is full, it is replaced with a larger one.
    - **Strategies**:
        - Incremental: Increase the size by a constant `c` .
        - Doubling: Double the size.
    - **Analysis**:
        - Incremental Strategy: Amortized time of an insert operation is **O(n)**.
        - Doubling Strategy: Amortized time of an insert operation is **O(1)**.
3. **Linked Lists**:
    - **Singly Linked List**:
        - **Structure**: Sequence of nodes, each storing an element and a link to the next node.



- **Operations**:
  - **Inserting at the Head**:
    1. Allocate a new node
    2. Insert new element
    3. Have new node point to old head
    4. Update head to point to new node
  - **Removing at the Head**:
    1. Update head to point to next node in the list
    2. Allow garbage collector to reclaim the former first node
  - **Inserting at the Tail**:
    1. Allocate a new node
    2. Insert new element
    3. Have new node point to null
    4. Have old last node point to new node
    5. Update tail to point to new node
  - **Removing at the Tail**:
    - Removing at the tail of a singly linked list is not efficient because there is no constant-time way to update the tail to point the previous node.
    - You have to traverse the whole Linked List to find the end / tail
- **Performance**: Traversal, insertion, and deletion run in **O(n)** time.

- **Doubly Linked List**:

- **Structure**: Nodes store an element, a link to the previous node, and a link to the next node.



- **Operations**: Insertion and deletion run in **O(1)** time.
- **Performance**:
    - Space used by a list of n elements is **O(n)**.
    - Space used by each position of the list is **O(1)**.
    - Traversal runs in **O(n)** time.

4. **Stacks**:
    - **Definition**: Stores arbitrary objects, following the last-in first-out (LIFO) scheme.
    - **Operations**:
        - `push(o)` : Inserts an element.
        - `pop()` : Removes the last inserted element.
        - `top()` : Returns the last inserted element without removing it.
        - `size()` , `empty()` .
    - **Applications**: Page-visited history in a web browser, undo sequence in a text editor, chain of method calls in the C++ run-time system.
    - **Performance**:
        - Space used is **O(n)**.
        - Each operation runs in **O(1)** time.
    - **Limitations**:
        - Maximum size of the stack must be defined priori and cannot be changed
        - Pushing a new element into a full stack causes an implementationspecific exception

5. **Queues**:
   - **Definition**: Stores arbitrary objects, following the first-in first-out (FIFO) scheme.
   - **Operations**:
     - `enqueue(o)` : Inserts an element at the end.
     - `dequeue()` : Removes the element at the front.
     - `front()` : Returns the element at the front without removing it.
     - `size()` , `empty()` .
   - **Applications**: Waiting lists, access to shared resources, multicore programming.
   - **Performance**: Each operation runs in **O(1)** time.
6. **Containers and Iterators**:
   - **Definition**:
     - An iterator abstracts the process of scanning through a collection of elements.
     - A container supports element access through iterators. (Stack, Queue, Vector, List, ...)
   - **Operations**:
     - `begin()` : Returns an iterator to the first element.
     - `end()` : Returns an iterator to an imaginary position just after the last element.
     - `*p` : Returns the element referenced by the iterator.
     - `++p` : Advances to the next element.
   - **Applications**: Iterating through a container, implementing iterators for array-based and linked list-based structures.

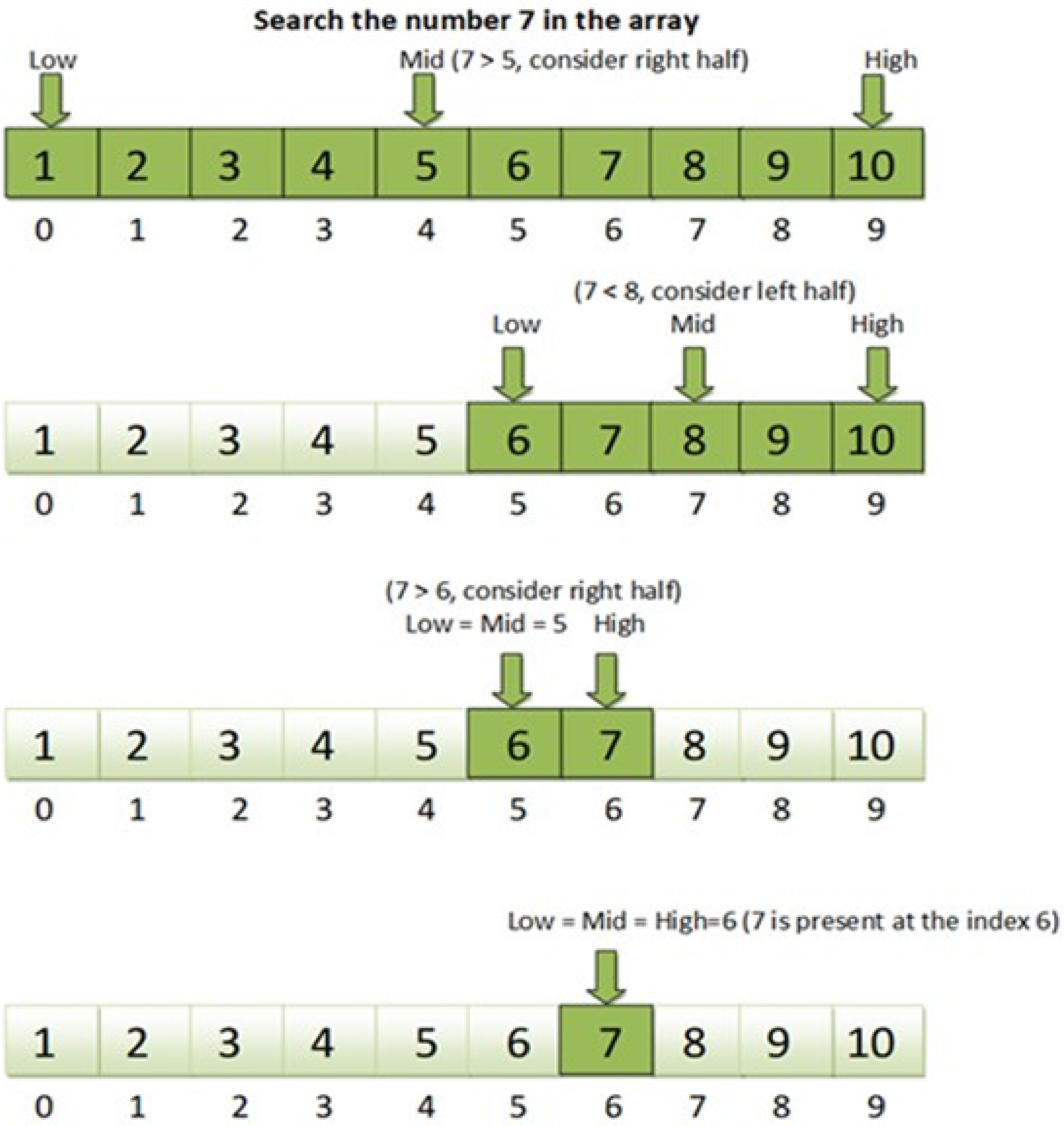
Lecture 3: Searching and Sorting

Key Concepts

1. **Searching Algorithms**:
   - **Linear Search**:
     - **Definition**: Start from the leftmost element of the array and compare each element with the target value.

- **Steps**:
    1. Compare the target value with each element in the array.
    2. If a match is found, return the index.
    3. If no match is found, return -1.
- **Complexity**: **O(n)**

- **Binary Search**:
    - **Definition**: Works by repeatedly dividing the search interval in half.



- **Steps**:
    1. Compare the target value with the middle element of the array.
    2. If the target value is equal to the middle element, return the index.

3. If the target value is less than the middle element, repeat the search on the left half.

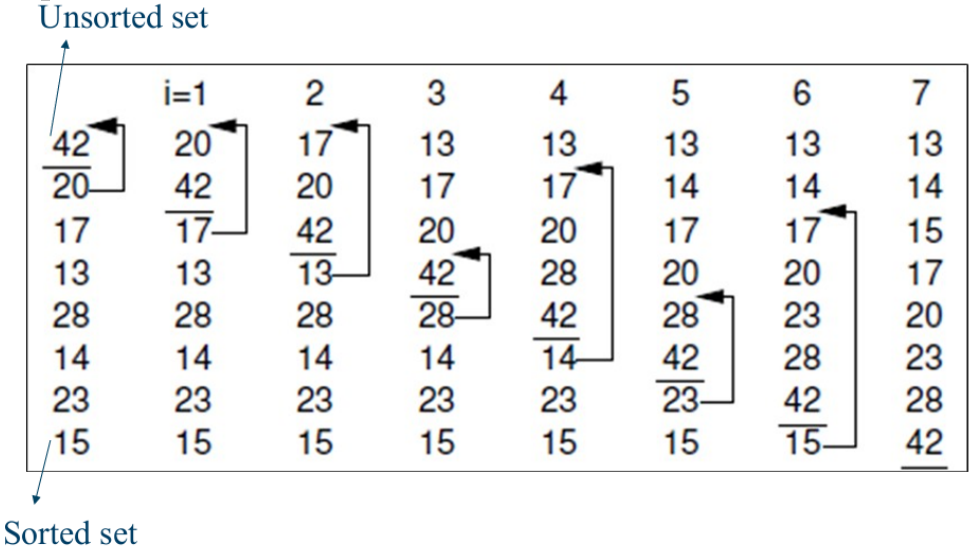
4. If the target value is greater than the middle element, repeat the search on the right half.

- **Complexity**: **O(log n)**

2. **Sorting Algorithms**:

- **Insertion Sort**:

  - **Definition**: Iterates through a list of records, inserting each record in turn at the correct position within a sorted list.



  - **Complexity**:

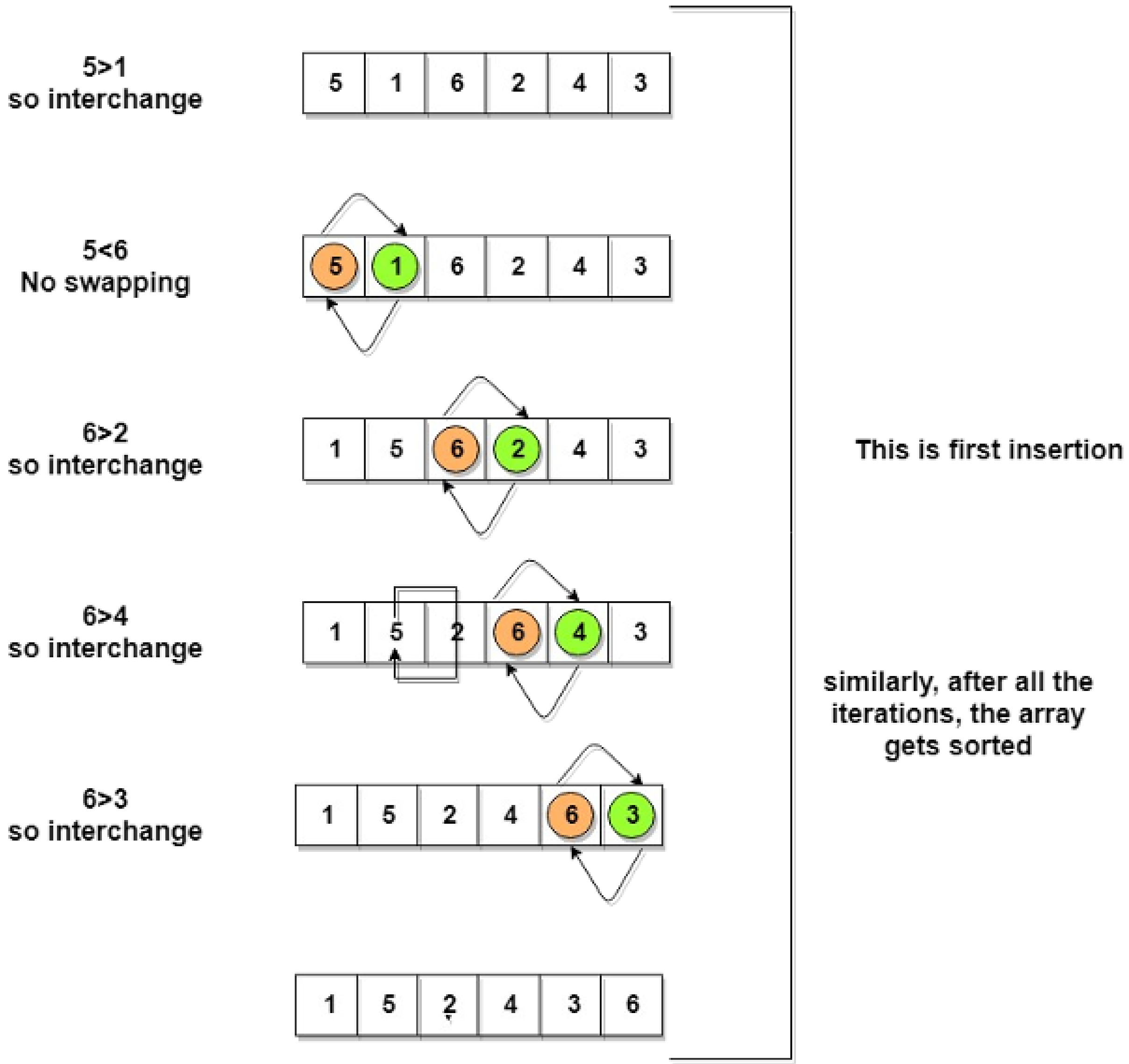
    - Worst case: **$O(n^2)$**

    - Best case: **O(n)**

    - Average case: **$O(n^2)$**

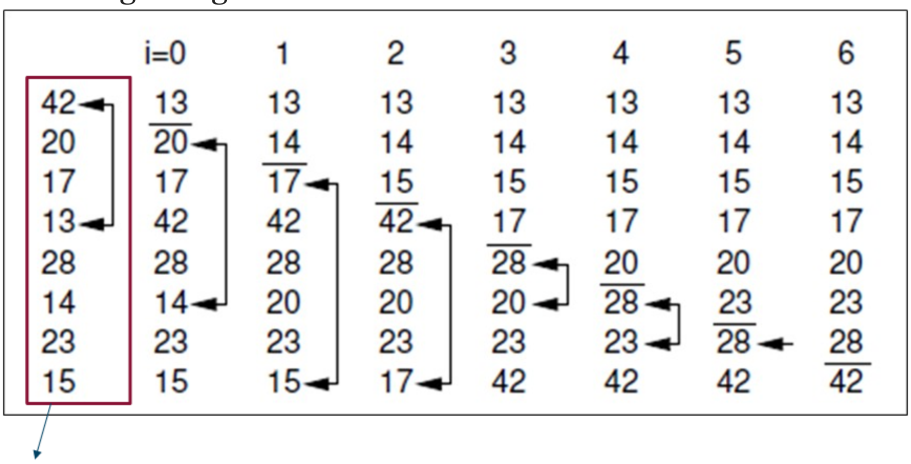
- **Bubble Sort**:

  - **Definition**: Repeatedly steps through the list, compares adjacent elements, and swaps them if they are in the wrong order.



- **Complexity**: $O(n^2)$

- **Selection Sort**:
  - **Definition**: Repeatedly finds the minimum element from the unsorted part and puts it at the beginning.



Scan the element for minimum, compare with top and swap

  - **Complexity**: $O(n^2)$

- **Quick Sort**:
  - **Definition**: Uses a divide-and-conquer strategy to sort the array.
  - **Steps**:
    1. Pick a pivot element.
    2. Partition the array so that elements less than the pivot are on the left, and elements greater than the pivot are on the right.
    3. Recursively apply the above steps to the subarrays.
  - **Complexity**:
    - Best case: **$O(n \log n)$**
    - Worst case: **$O(n^2)$**
- **Merge Sort**:
  - **Definition**: Divides the array into two halves, sorts each half, and then merges the sorted halves.
  - **Merging Procedure**:
    - **Steps**:
      1. Compare the first elements of two sorted arrays.
      2. Copy the smaller element to the result array.
      3. Repeat until all elements are merged.
  - **Complexity**: **$O(n \log n)$**

Additional Concepts

- **Analyzing Sorting Algorithms**:
  - **Running Time**: Measure the number of comparisons made between keys.
  - **Swap Operations**: Measure the number of swap operations when the records are large.

Summary of Sorting Algorithms

Algorithm	Time	Notes
Insertion sort	$O(n^2)$	In-place Slow (small inputs)
Bubble sort	$O(n^2)$	In-place Slow (small inputs)
Selection sort	$O(n^2)$	In-place Slow (small inputs)
Quick sort	$O(n^2)$ expected	In-place, randomized Fastest (large inputs)
Merge sort	$O(nlogn)$	Sequential data access Fast (huge inputs)

Lecture 4: Maps and Hashing

Key Concepts

1. **Maps**:
   - **Definition**: A map stores a collection of (key, value) pairs, where each key appears at most once.
   - **Operations**:
     - `find(k)` : Returns an iterator to the entry with key `k` or the end if not found.
     - `put(k, v)` : Inserts or updates the entry with key `k` and value `v` .
     - `erase(k)` : Removes the entry with key `k` .
     - `size()` , `empty()` , `begin()` , `end()` .
   - **Applications**: Address book, student-record database.
2. **Entry ADT**:
   - **Definition**: An entry stores a key-value pair `(k, v)` .
   - **Methods**:
     - `key()` : Returns the associated key.
     - `value()` : Returns the associated value.
     - `setKey(k)` : Sets the key to `k` .
     - `setValue(v)` : Sets the value to `v` .
3. **List-Based Map**:

- **Implementation**: Using an unsorted list.
- **Performance**:
  - `put`: **O(n)** time.
  - `find` and `erase`: **O(n)** time.
- **Use Case**: Effective for small maps or maps with frequent insertions and rare searches/removals.

4. **Hash Functions and Hash Tables**:
   - **Hash Function**: Maps keys to integers in a fixed interval `[0, N-1]`.
   - **Example**: `h(x) = x mod N`.
   - **Components**: Hash function `h` and an array (table) of size `N`.
   - **Goal**: Store item `(k, o)` at index `i = h(k)`.
5. **Hash Codes**:
   - **Types**:
     - Memory address.
     - Component sum.
     - Integer cast.
     - Polynomial accumulation.
   - **Example**: Polynomial accumulation for strings with `z = 33`.
6. **Compression Functions**:
   - **Types**:
     - Division: `h2(y) = y mod N`.
     - MAD (Multiply, Add, and Divide): `h2(y) = (ay + b) mod N`.
7. **Collision Handling**:
   - **Separate Chaining**: Each cell points to a linked list of entries.
   - **Linear Probing**: Colliding item placed in the next available cell.
   - **Double Hashing**: Uses a secondary hash function `d(k)`.
8. **Performance of Hashing**:

- **Worst Case**: **O(n)** time for searches, insertions, and removals.
- **Load Factor**: Affects performance.
- **Expected Running Time**: **O(1)** for dictionary ADT operations with proper load factor management.
- **Applications of hash tables**:
  - Small databases
  - Compilers
  - Browser caches

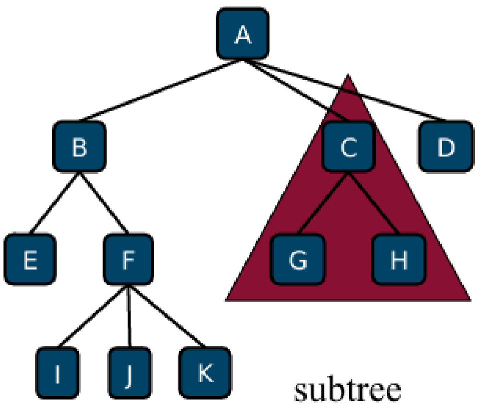
9. **Dictionaries**:
   - **Definition**: Models a searchable collection of key-element entries.
   - **Operations**:
     - `find(k)`, `findAll(k)`, `put(k, o)`, `erase(k)`, `begin()`, `end()`, `size()`, `empty()`.
   - **Applications**: Word-definition pairs, credit card authorizations, DNS mapping.
10. **Priority Queues**:
    - **Definition**: Stores entries with a key indicating priority.
    - **Operations**:
    - `insert(e)`: Inserts an entry.
    - `removeMin()`: Removes the entry with the smallest key.
    - `min()`, `size()`, `empty()`.
    - **Applications**: Standby flyers, auctions, stock market.

Lecture 5: Trees

Key Concepts

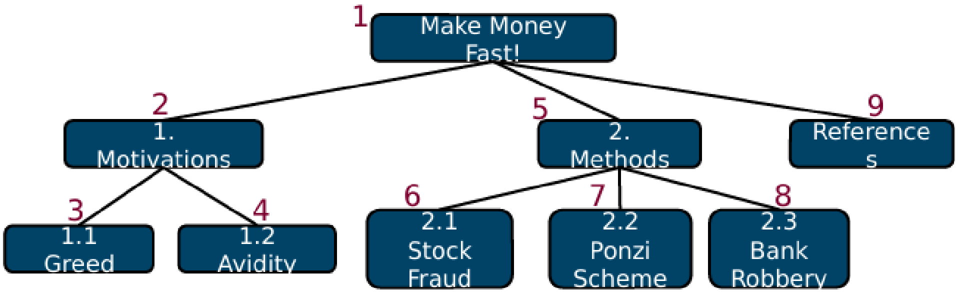
1. **Trees**:
   - **Definition**: An abstract model of a hierarchical structure consisting of nodes with a parent-child relation.
   - **Applications**: Organization charts, file systems, programming environments.

2. **Tree Terminology**:
    - **Root**: Node without a parent.
    - **Internal Node**: Node with at least one child.
    - **External Node (Leaf)**: Node without children.
    - **Ancestors**: Parent, grandparent, etc.
    - **Depth**: Number of ancestors.
    - **Height**: Maximum depth of any node.
    - **Descendant**: Child, grandchild, etc.
    - **Subtree**: Tree consisting of a node and its descendants.

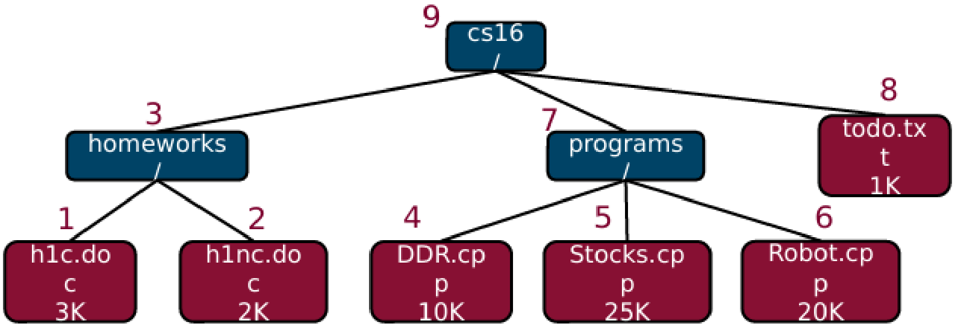


3. **Tree ADT**:
    - **Methods**:
        - `size()` : Returns the number of nodes.
        - `empty()` : Checks if the tree is empty.
        - `root()` : Returns the root node.
        - `positions()` : Returns a list of all nodes.
        - `parent(p)` : Returns the parent of node `p` .
        - `children(p)` : Returns the children of node `p` .
        - `isRoot(p)` : Checks if node `p` is the root.
        - `isExternal(p)` : Checks if node `p` is an external node.
4. **Tree Traversals**:

- **Preorder Traversal**: Visits nodes in a top-down manner.

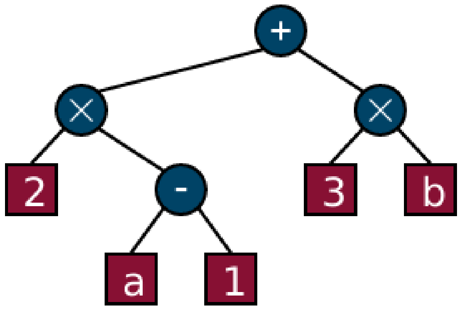


- **Postorder Traversal**: Visits nodes in a bottom-up manner.



- **Inorder Traversal**: Visits nodes in a left-root-right manner (specific to binary trees).

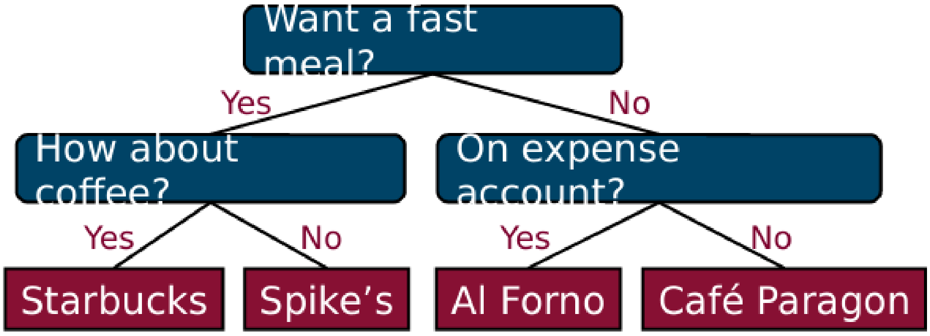
5. **Binary Trees**:
   - **Definition**: A tree where each internal node has *at most* two children.
   - **Applications**:
     - Arithmetic expressions
     - Decision processes
     - Searching
   - **Properties**:
     - Proper Binary Tree: Each node has either *0 or 2* children.
     - Height: **O(log n)** for a heap storing n keys.
6. **Binary Tree ADT**:
   - **Methods**:
     - `left(p)` : Returns the left child of node `p` .
     - `right(p)` : Returns the right child of node `p` .
7. **Arithmetic Expression Tree**:
   - **Definition**: Binary tree associated with an arithmetic expression.



- **Nodes**: Internal nodes are operators, external nodes are operands.

8. **Decision Tree**:

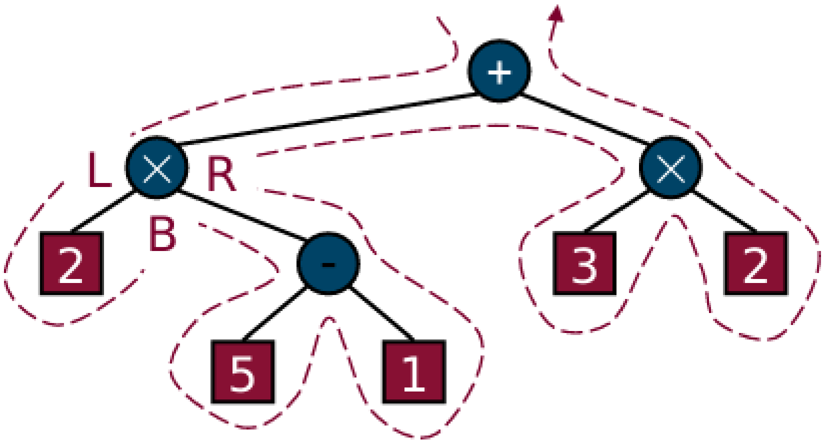
   - **Definition**: Binary tree associated with a decision process.



   - **Nodes**: Internal nodes are questions, external nodes are decisions.

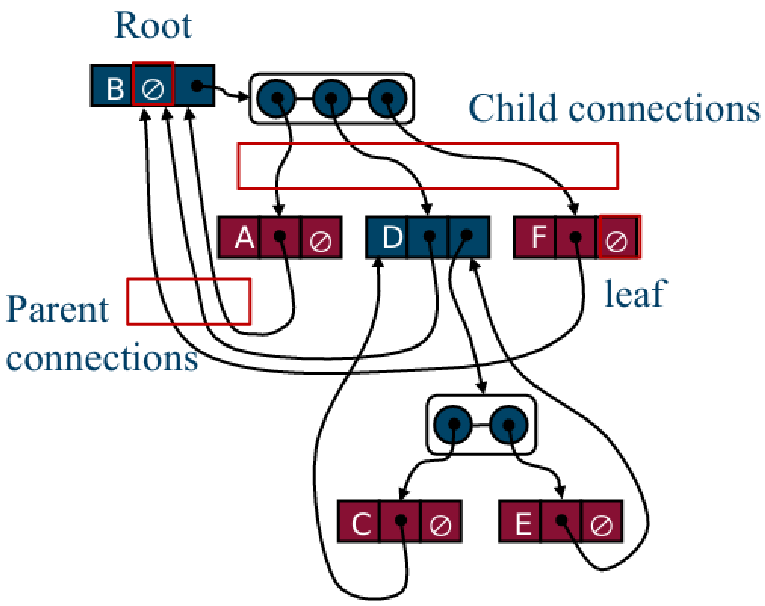
9. **Euler Tour Traversal**:

   - Generic traversal of a binary tree.
   - Includes special cases like preorder, postorder and inorder traversals.



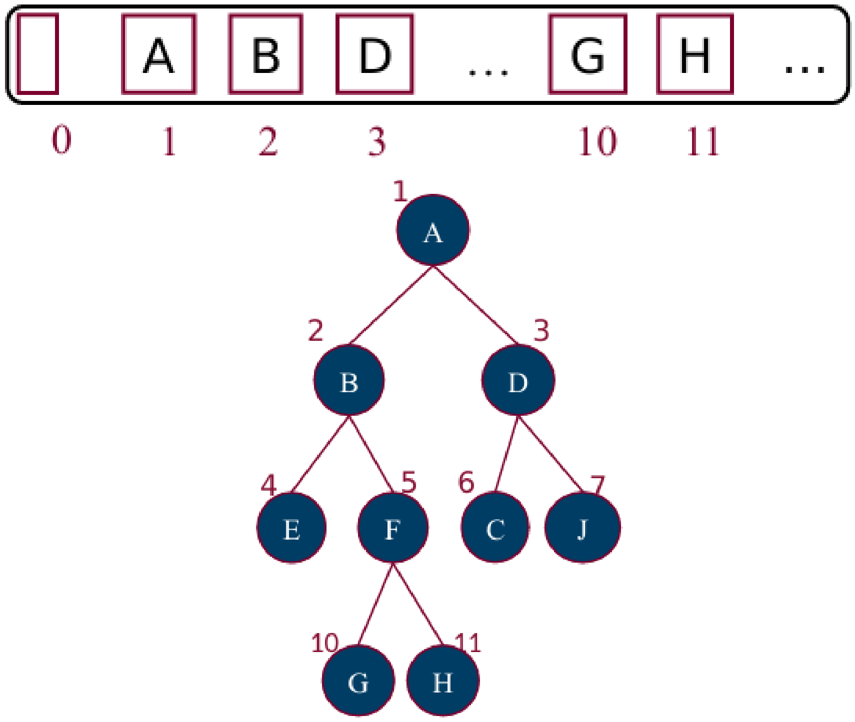
10. **Linked Structure for Trees**:

    - **Node Representation**: Stores element, parent node, and children nodes.
    - **Binary Trees**: Stores element, parent node, left child, and right child.



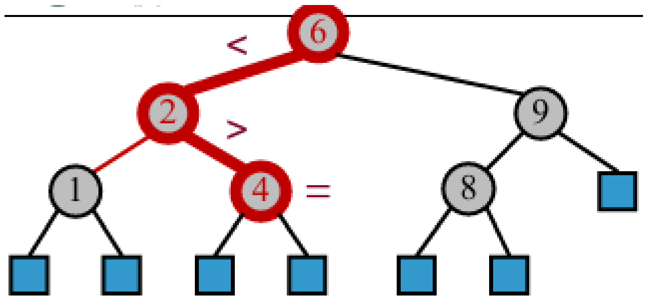
11. **Array-Based Representation of Binary Trees**:

    - **Nodes Stored in Array**: Node `v` stored at `A[rank(v)]`.
    - **Rank Calculation**: Based on parent-child relationship.



12. **Binary Search Trees (BST)**:

    - **Definition**: Binary tree storing keys at internal nodes with the property that keys in the left subtree are less than the root, and keys in the right subtree are greater.



- **Operations**:
  - `TreeSearch(k, v)`: Searches for key `k` starting from node `v`.
  - `insert(k, o)`: Inserts key `k` with value `o`.
  - `erase(k)`: Removes key `k`.

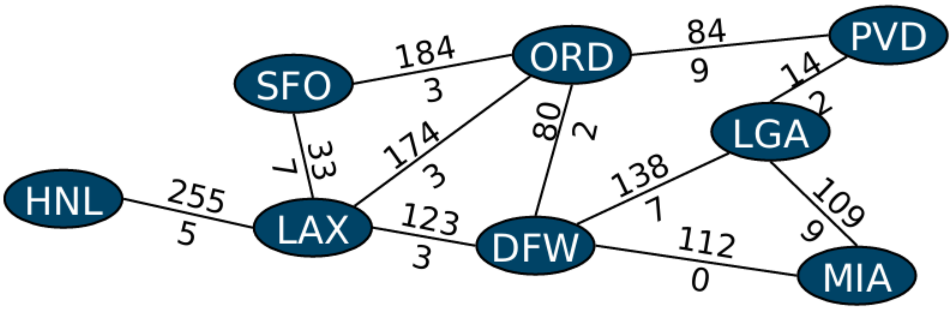
13. **Heaps**:
    - **Definition**: Binary tree storing keys with the heap-order property (parent key is less than or equal to child keys).
    - **Operations**:
    - `insertItem(k)`: Inserts key `k`.
    - `removeMin()`: Removes the smallest key.
    - **Properties**: Complete binary tree, height **O(log n)**.
14. **Heap Operations**:
    - **Upheap**: Restores heap-order property after insertion by swapping *k* along an upward path from the insertion node.
    - **Downheap**: Restores heap-order property after removal by swapping *k* along a downward path from the root.
    - **Heap-Sort**: Sorting algorithm using a heap-based priority queue which is much faster than quadratic sorting algorithms, this has **O(n log n)** time.

Lecture 6: Graphs

Key Concepts

1. **Graphs**:
   - **Definition**: A graph is a pair (V, E), where V is a set of nodes (vertices) and E is a collection of pairs of vertices (edges).
     - **Vertices**: nodes.

- **Edges**: connections between vertices.

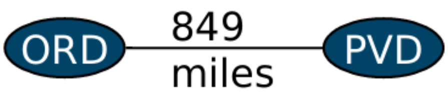


- **Applications**: Electronic circuits, transportation networks, computer networks, databases.

2. **Edge Type**:

- **Directed Edge**: Ordered pair of vertices (u, v) where u is the origin and v is the destination.

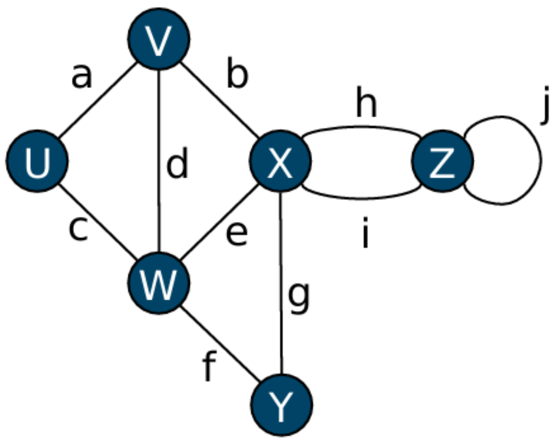
- **Undirected Edge**: Unordered pair of vertices (u, v).



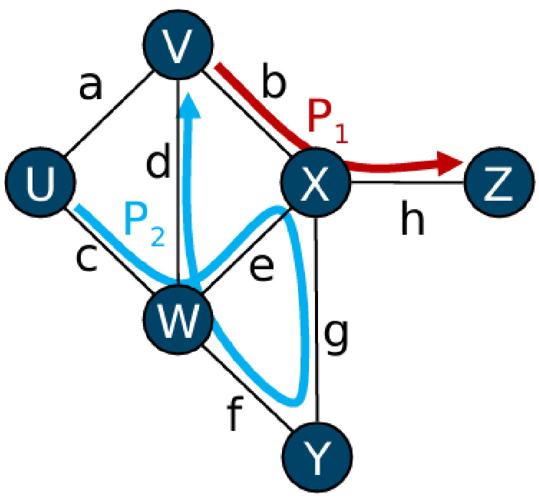
- **Directed Graph**: All edges are directed. (route network)
- **Undirected Graph**: All edges are undirected. (flight network)

3. **Terminology**:

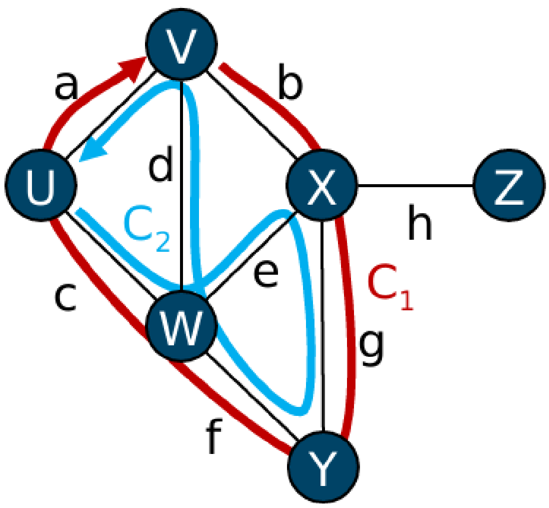
- **End vertices (endpoints)**: U and V are endpoints of a.
- **Edges incident**: a, d, and b are incident on V.
- **Adjacent vertices**: U and V are adjacent.
- **Degree of a Vertex**: X has degree 5 (5 edges: b, e, g, h, i).
- **Parallel edges**: h and i are parallel edges.
- **Self-loop**: j is a self-loop



- **Path**: Sequence of alternating vertices and edges. (P2)
- **Simple path**: Path with only distinct vertices and edges. (P1)



- **Cycle**: Circular sequence of alternating vertices and edges. (C2)
- **Simple cycle**: Cycle with only distinct vertices and edges. (C1)

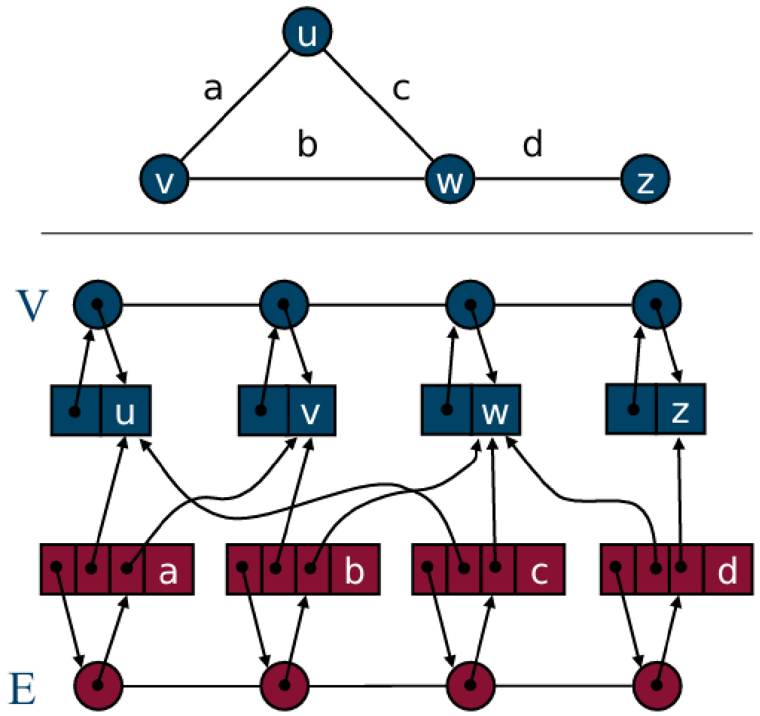


4. **Graph Properties**:

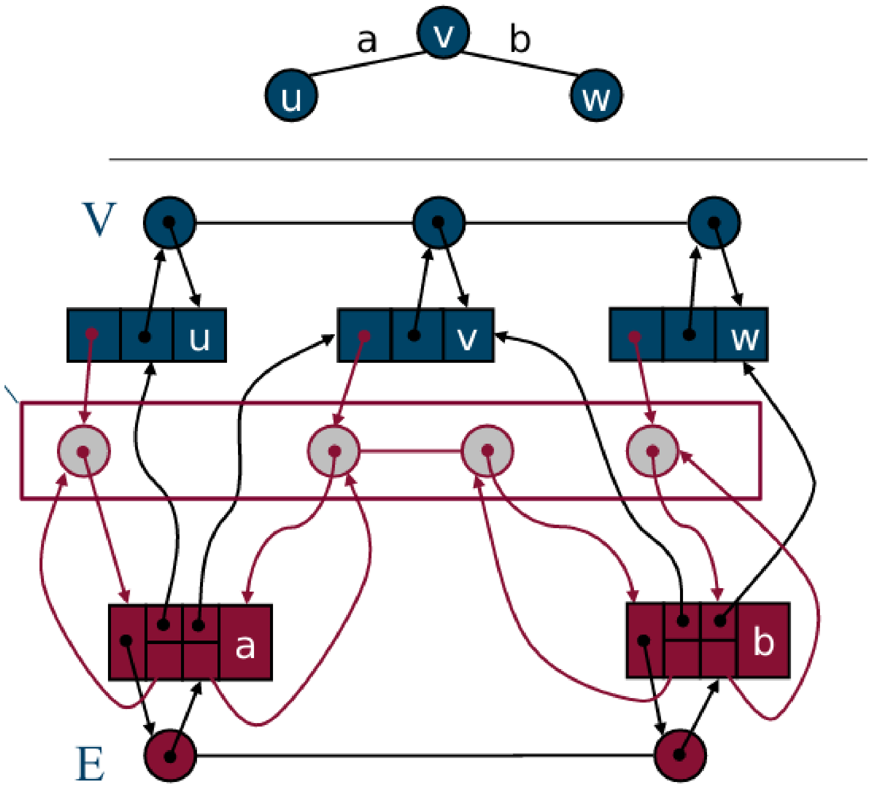
- n = number of vertices
- m = number of edges
- deg(v) = degree of vertex v

- **Property 1**: $\sum_v deg(v) = 2m$, sum of the degrees of all vertices is twice the number of edges.
- **Property 2**: In an undirected graph with no self-loops and no multiple edges, the number of edges is at most $m \leq n(n-1)/2$.

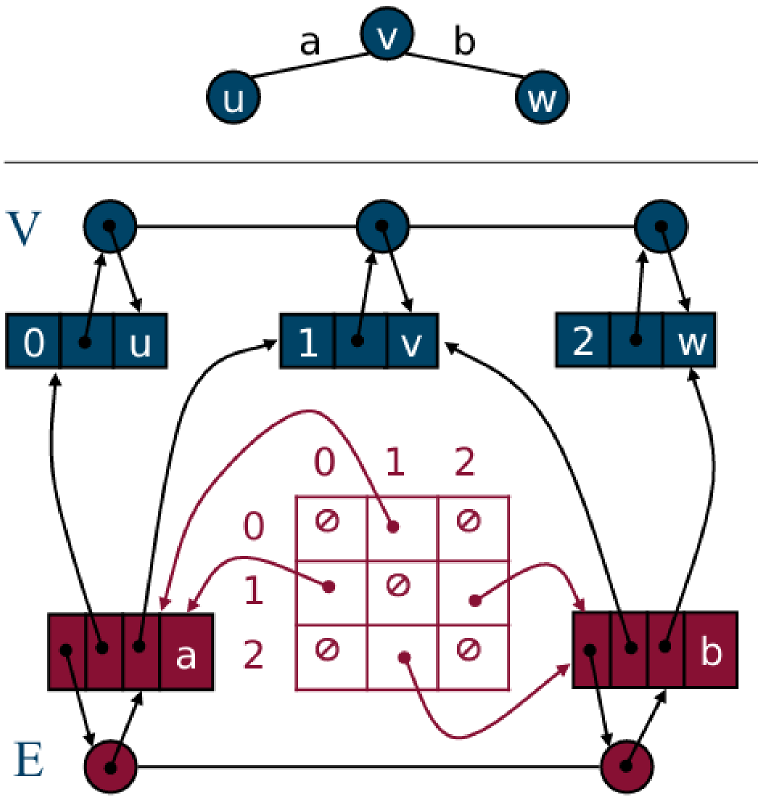
5. **Graph ADT Methods**:
   - **Accessor Methods**:
     - `e.endVertices()` : Returns the two end vertices of edge e.
     - `e.opposite(v)` : Returns the vertex opposite of v on edge e.
     - `u.isAdjacentTo(v)` : Checks if vertices u and v are adjacent.
   - **Update Methods**:
     - `insertVertex(o)` : Inserts a vertex storing element o.
     - `insertEdge(v, w, o)` : Inserts an edge (v, w) storing element o.
     - `eraseVertex(v)` : Removes vertex v and its incident edges.
     - `eraseEdge(e)` : Removes edge e.
6. **Graph Representations**:
   - **Edge List Structure**: Stores vertices and edges as objects with references to their positions in sequences.



- **Adjacency List Structure**: Each vertex has a sequence of references to its incident edges.

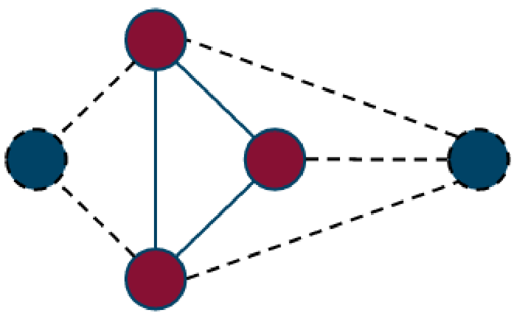


- **Adjacency Matrix Structure**: 2D-array where each cell represents the presence or absence of an edge between vertices.

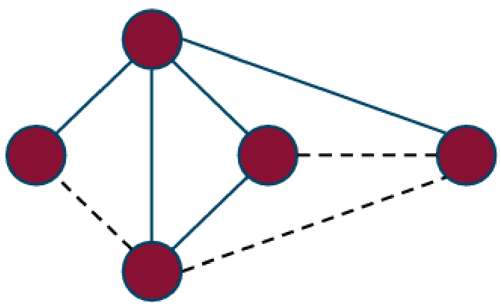


7. **Subgraphs**:

- **Definition**: A subgraph S of a graph G is a graph where the vertices and edges of S are subsets of the vertices and edges of G.

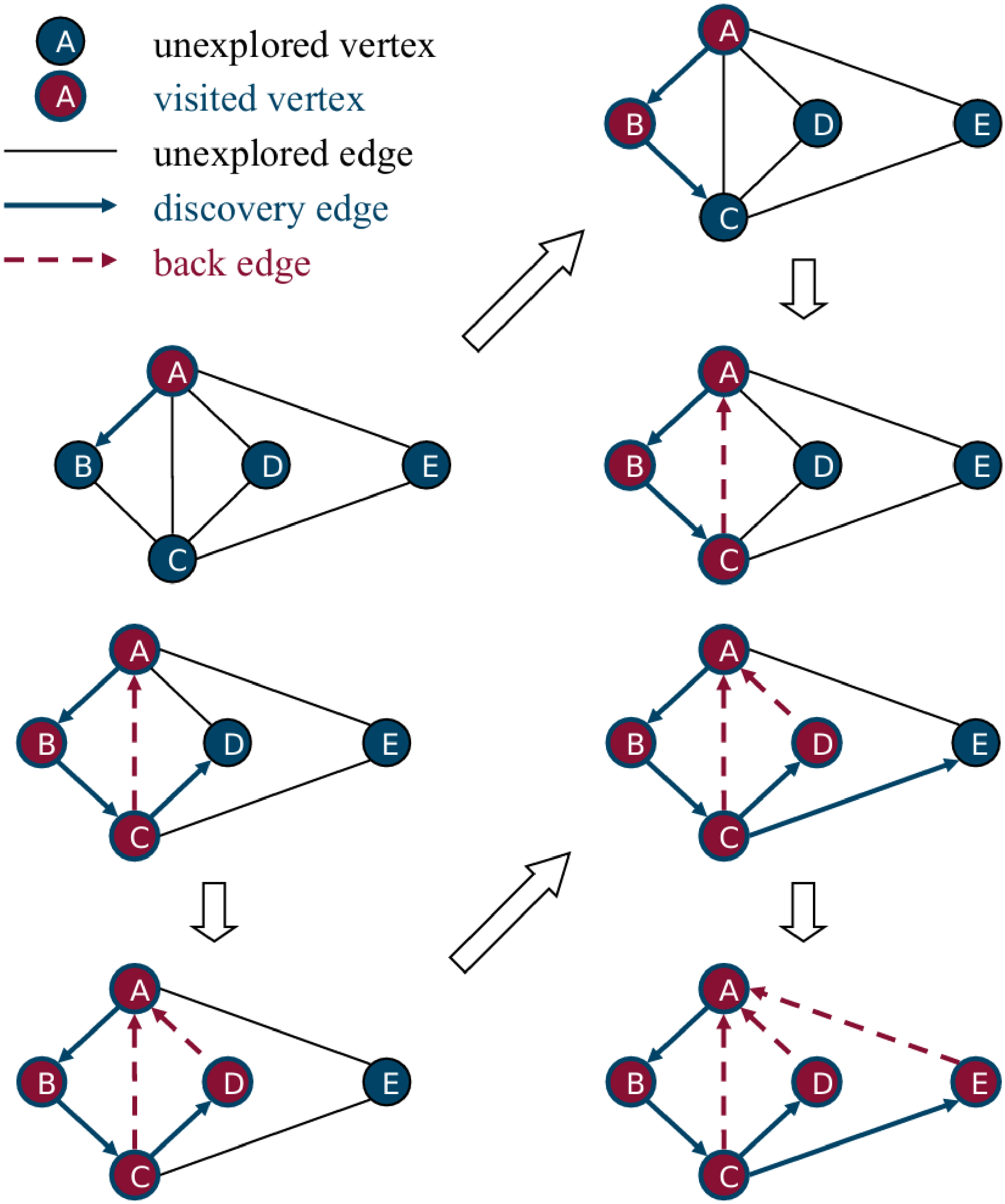


- **Spanning Subgraph**: Contains all the vertices of G.

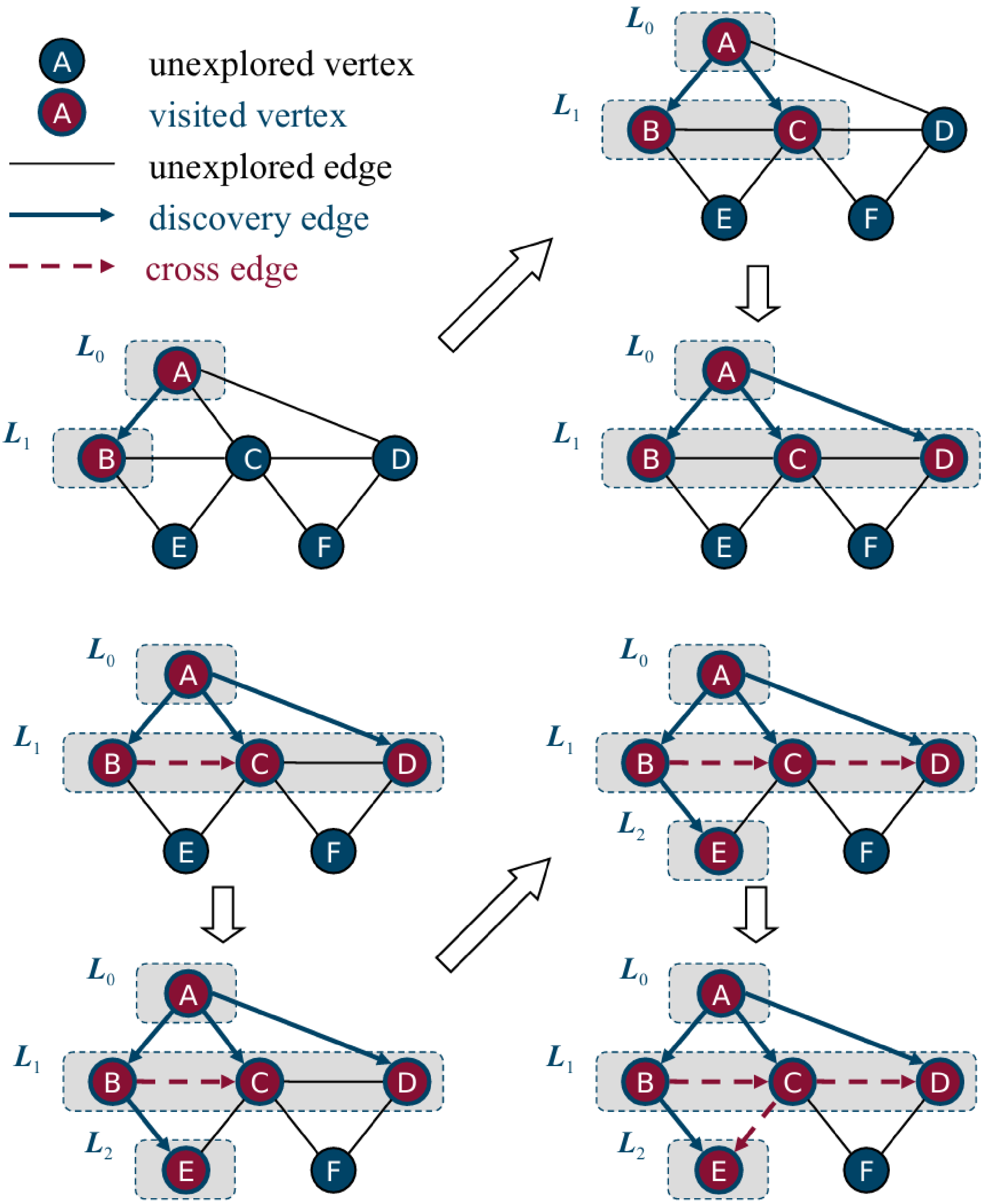


8. **Graph Traversals**:

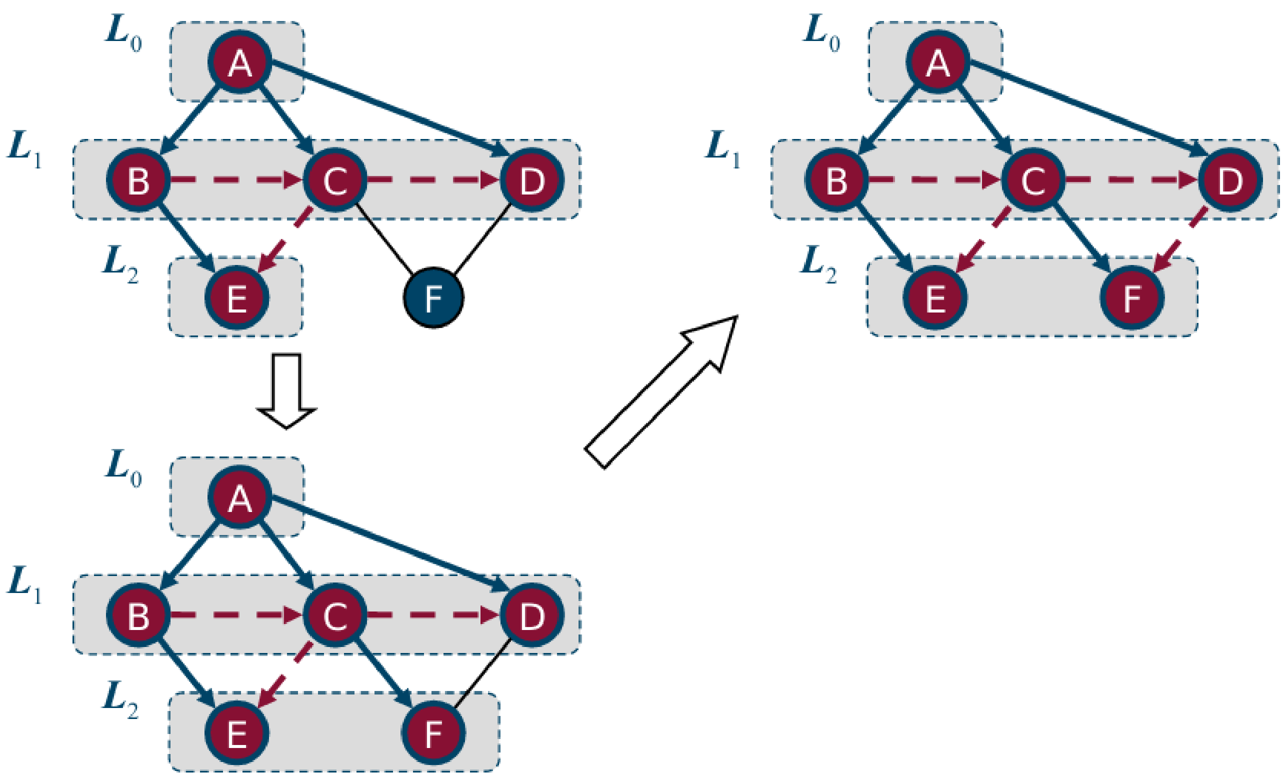
- **Depth-First Search (DFS)**: Visits all vertices and edges of a graph, computes connected components, and forms a spanning tree. (DFS is to graph what Euler tour is to binary trees)



- **Breadth-First Search (BFS)**: Visits all vertices and edges of a graph, computes connected components, and forms a spanning tree.



unexplored vertex
visited vertex
unexplored edge
discovery edge
cross edge
L_0
L_1
L_2
A
B
C
D
E
F



9. **DFS vs BFS**:

Applications	**DFS**	**BFS**
Spanning forest, connected components, paths, cycles	x	x
Shortest paths		x
Biconnected components	x	

10. **Directed Graphs (Digraphs)**:

    - **Definition**: A graph where all edges are directed. (means task a must be completed before b can be started)
    - **Applications**: Task scheduling, one-way streets, flights.
    - **Strong Connectivity**: Each vertex can reach all other vertices.
    - **Transitive Closure**: Provides reachability information about a digraph.

11. **Shortest Paths**:

    - **Definition**: Path of minimum total weight between two vertices in a weighted graph.
    - **Dijkstra's Algorithm**: Computes shortest paths from a start vertex to all other vertices in **O((n + m) log n)** time.
        - Based on the greedy method, it adds vertices by increasing distance.