02 - Day

Qualities of a Good Algorithm

Here are some essential qualities that define a good algorithm:

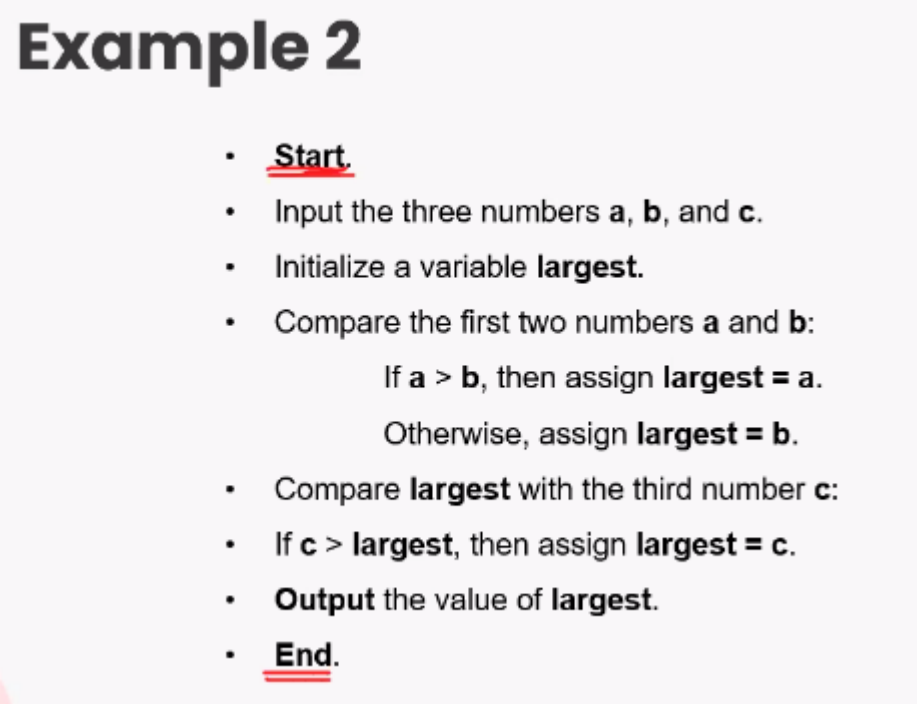
- Input and output should be defined precisely.
- Each step in the algorithm should be clear and unambiguous.
- Algorithms should be mostly effective among many different ways to solve a problem.
- An algorithm shouldn't include computer code.Instead, the algorithm should be written in such a way that it can be used in different programming languages.

Key Characteristics of algorithms include

- **Input**
  - Algorithms take zero or more inputs. These inputs can be numbers, strings, arrays, or any other data type relevant to the problem being solved.
- **Output**
  - Algorithms produce one or more outputs base d on the inputs and the steps performed. The output can be a result, a modified input, or simply a signal indicating the completion of a task.
- **Definiteness**
  - Algorithms are precise and unambiguous, with each step clearly defined and executable. This ensures that the algorithm can be followed exactly and consistently to produce the desired outcome.
- **Finiteness**
  - Algorithms must terminate after a finite number of steps. They cannot run indefinitely, ensuring that the problem-solving process eventually reaches a conclusion.
- **Effectiveness**
  - Algorithms must be effective in solving the problem they are designed for.This means they should correctly solve the problem, preferably in an efficient manner, within reasonable time and resource constraints.

Question 2

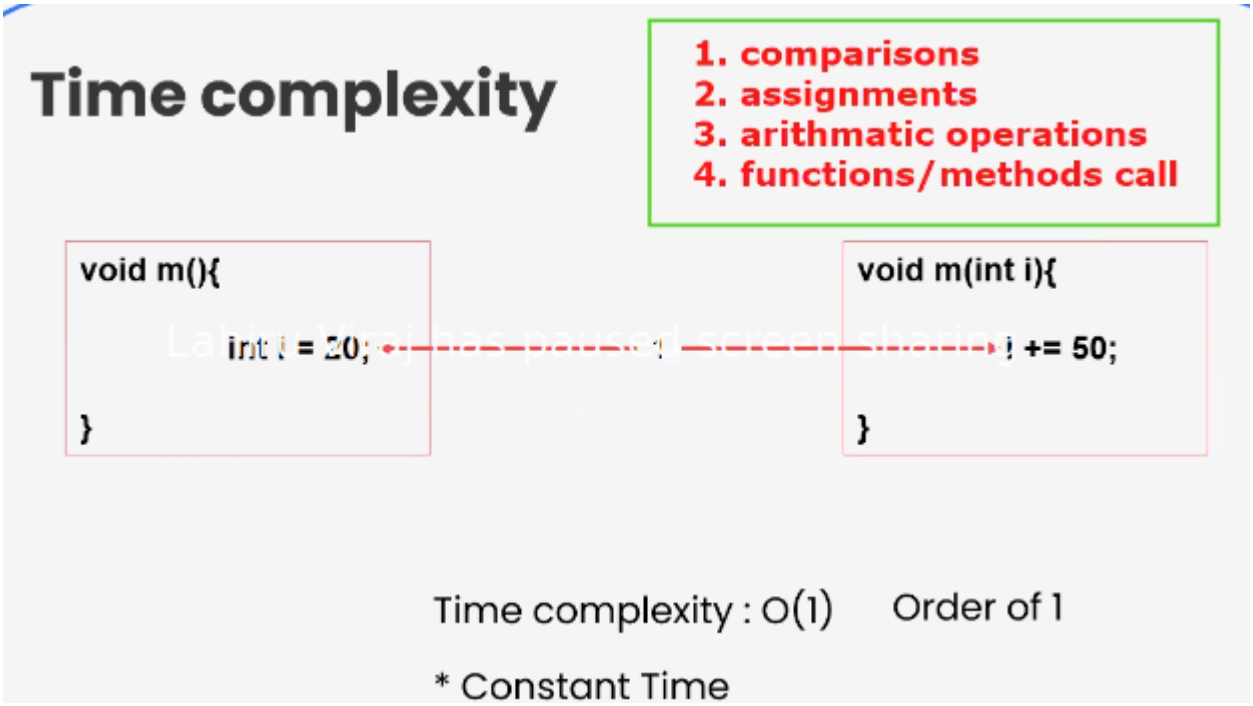
write an algorithm to find the largest number



Efficiency of Algorithm

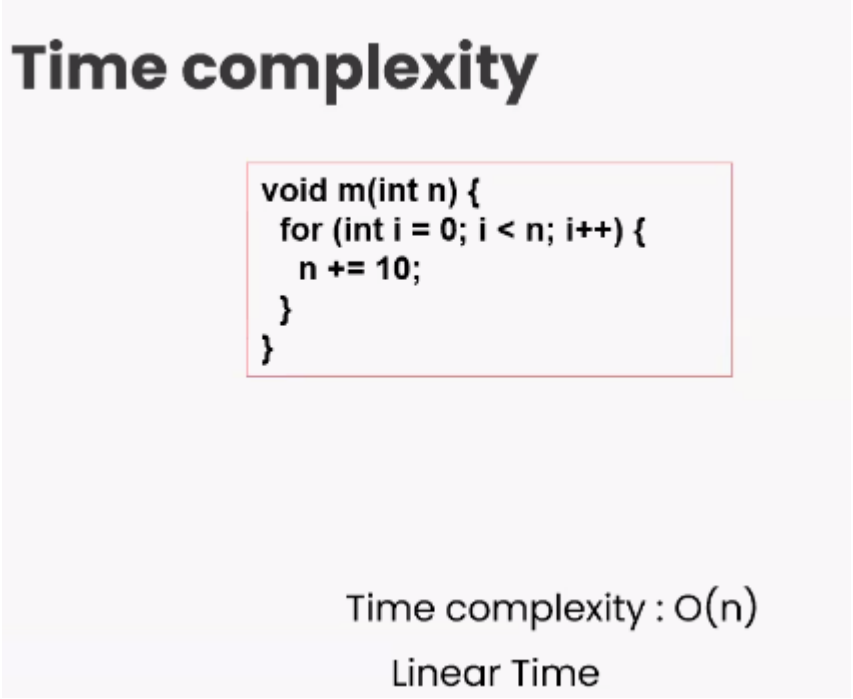
- Time complexity
  -Amount of time required for execution
- Space complexity
  - Amount of memory space required when it runs

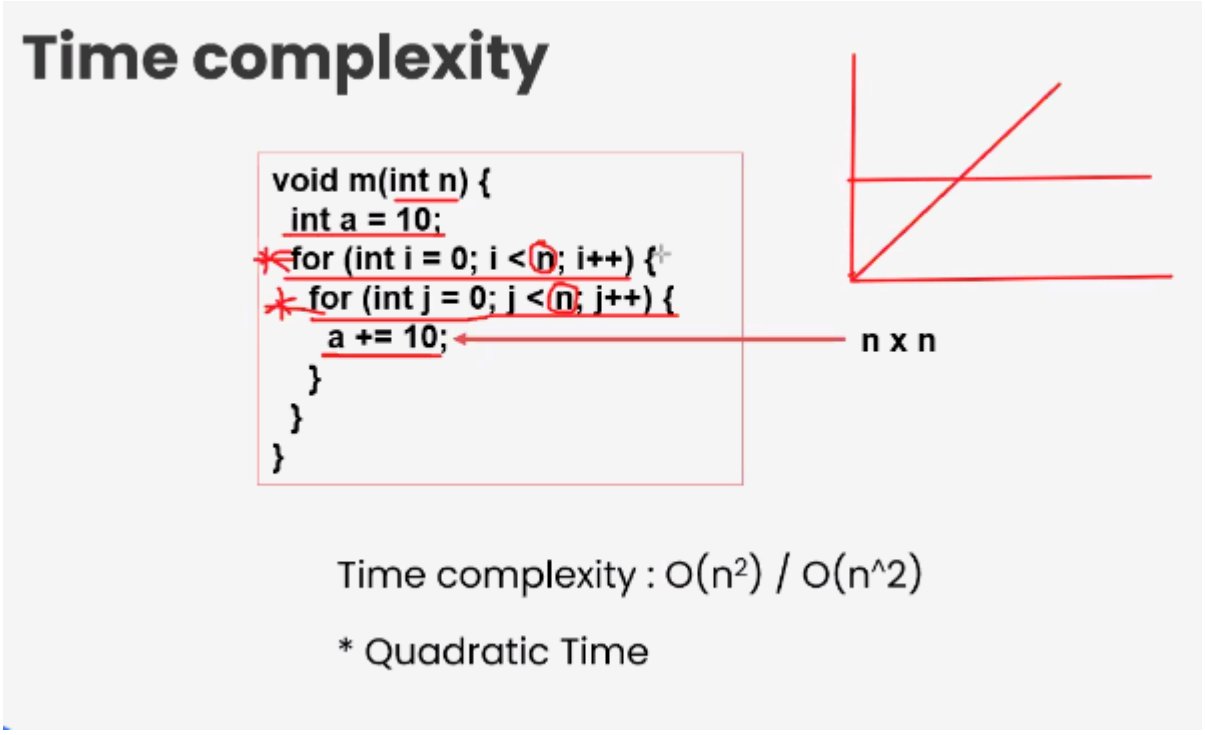
Time Complexity

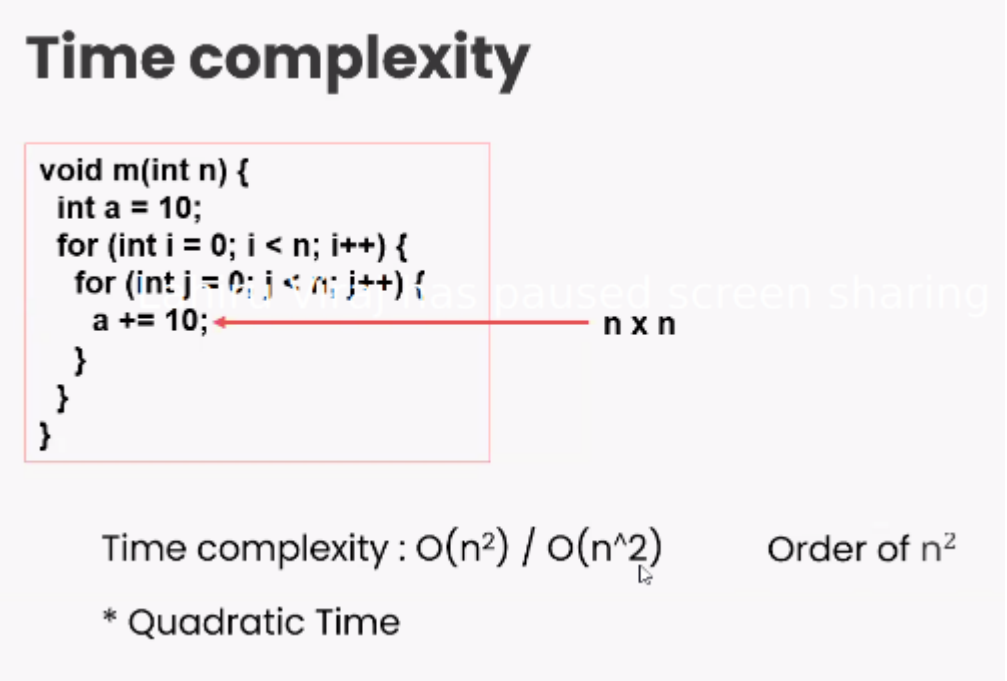


inside a algorithm , these things would be done.

1. comparisons
2. assignments
3. arithmetic operations
4. functions / method calls







If n = 10,100,1000:

O(1) -> always 1
O(n) -> 10, 100, 1000
O(n^2) -> 100, 10000, 100000

01 - Day

Data Structures and Algorithms

Empowering Developers with Essential Data Skills for Efficient Problem Solving

Data Structures

Data structures are fundamental constructs used in software engineering to organize and manage data efficiently. They provide a way to store and manipulate data so that operations like searching ,sorting , inserting, and deleting can be performed quickly and effectively.

Data structures are used to organize and store data to use it in an effective way when performing data operations.

Note :

Depending on your requirement and project, it is important to choose the right data structure for your project.

Algorithms

Algorithms are step-by-step procedures or sets or rules flowed t solve problems or accomplish specific tasks. In software engineering, algorithms are essential for performing

computations, data processing, and automated reasoning. They service as the backbone for computer programs,determining how data is processed, manipulated, and transformed.

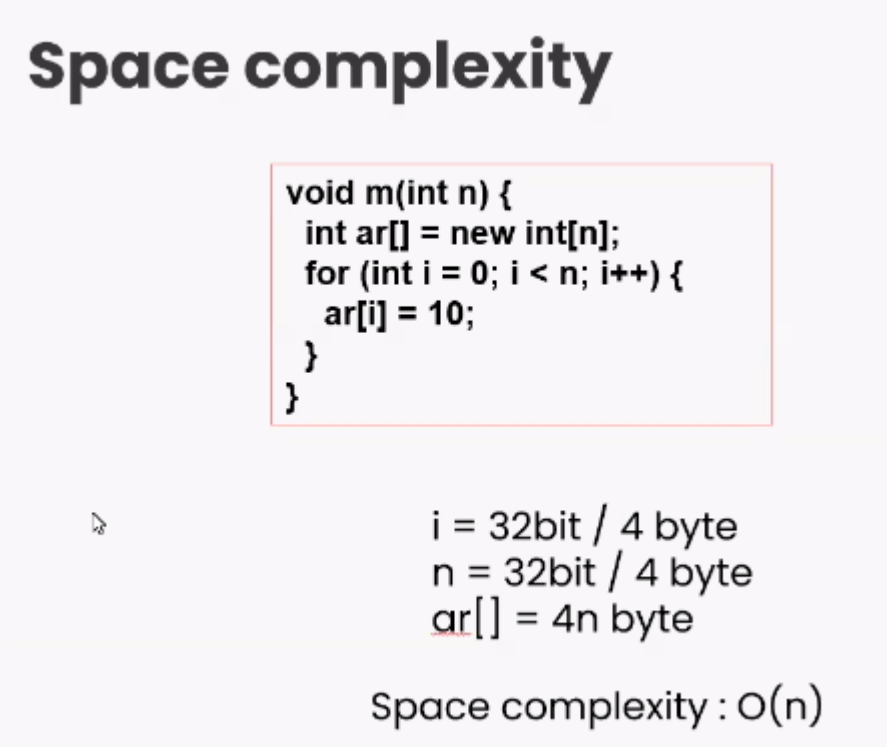
Algorithm is a set of well-defined instructions to solve a particular problem. It takes a set of input(s) and produces the desired output .

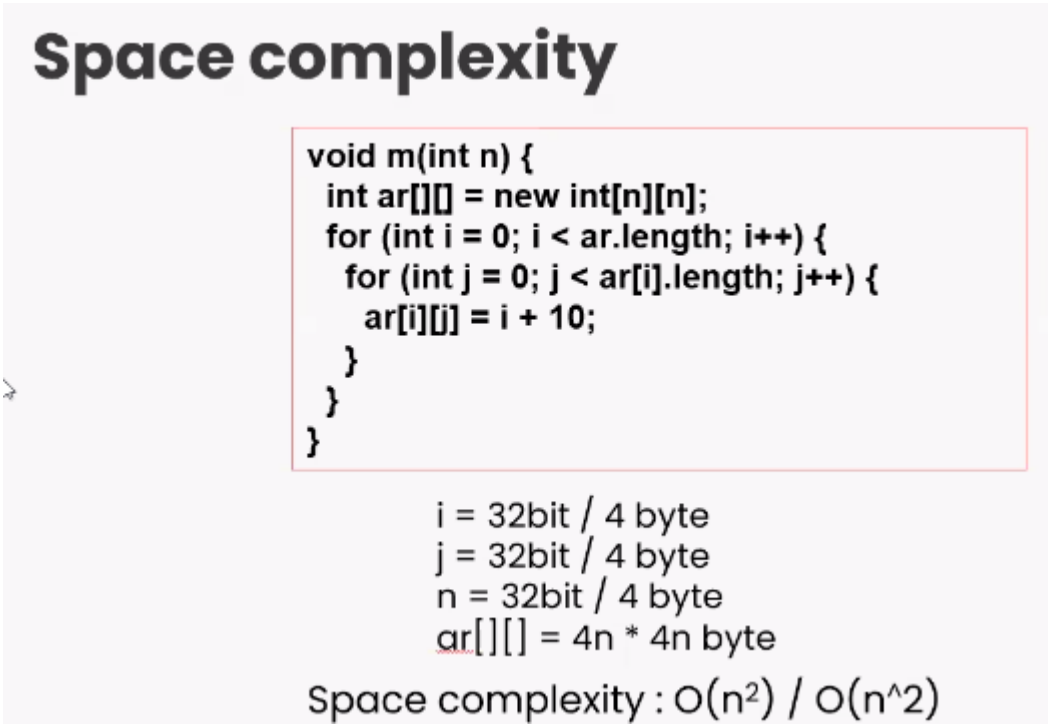
Example 1

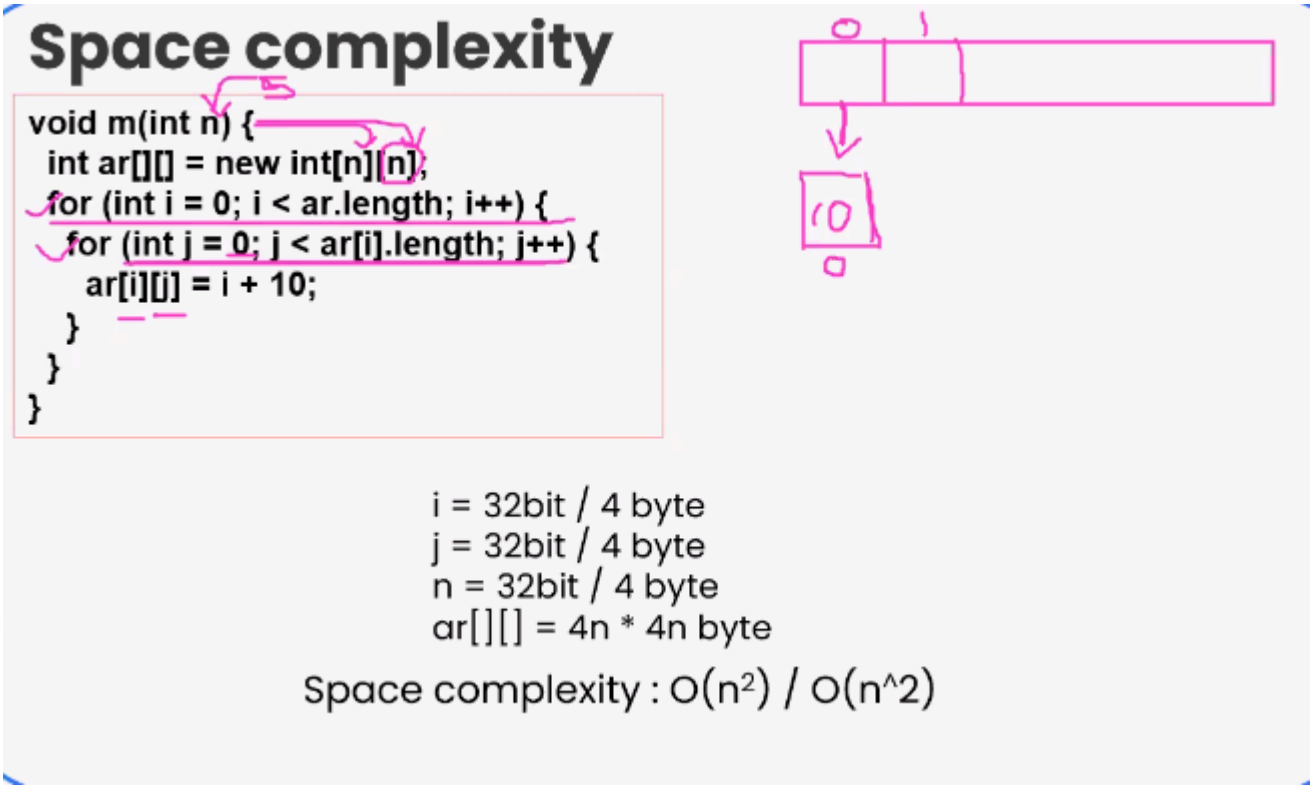
an algorithm to add two numbers :

1. Take two number inputs.
2. Add numbers using the + operator.
3. Display the result.

03 - Day







Quadratic complexity / Square Complexity

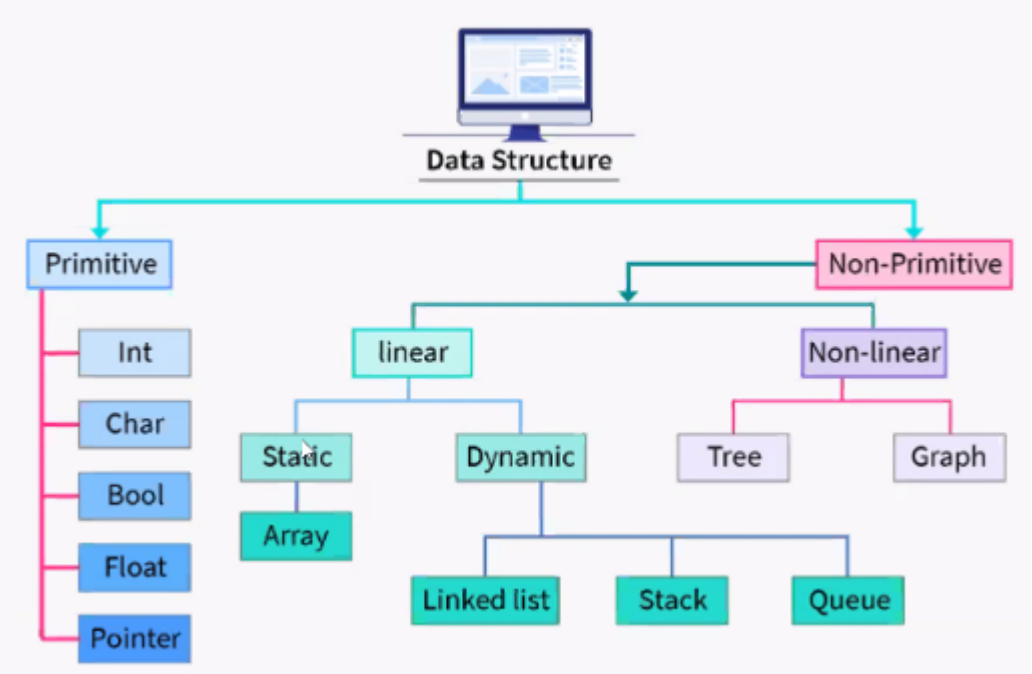
Data Structure and types

Primitive Data Types

- The primitive data structure,also referred to as build-in data types,can only store data of a single type.This includes integers, floating points, characters ,and similar types.

Non-Primitive Data Types

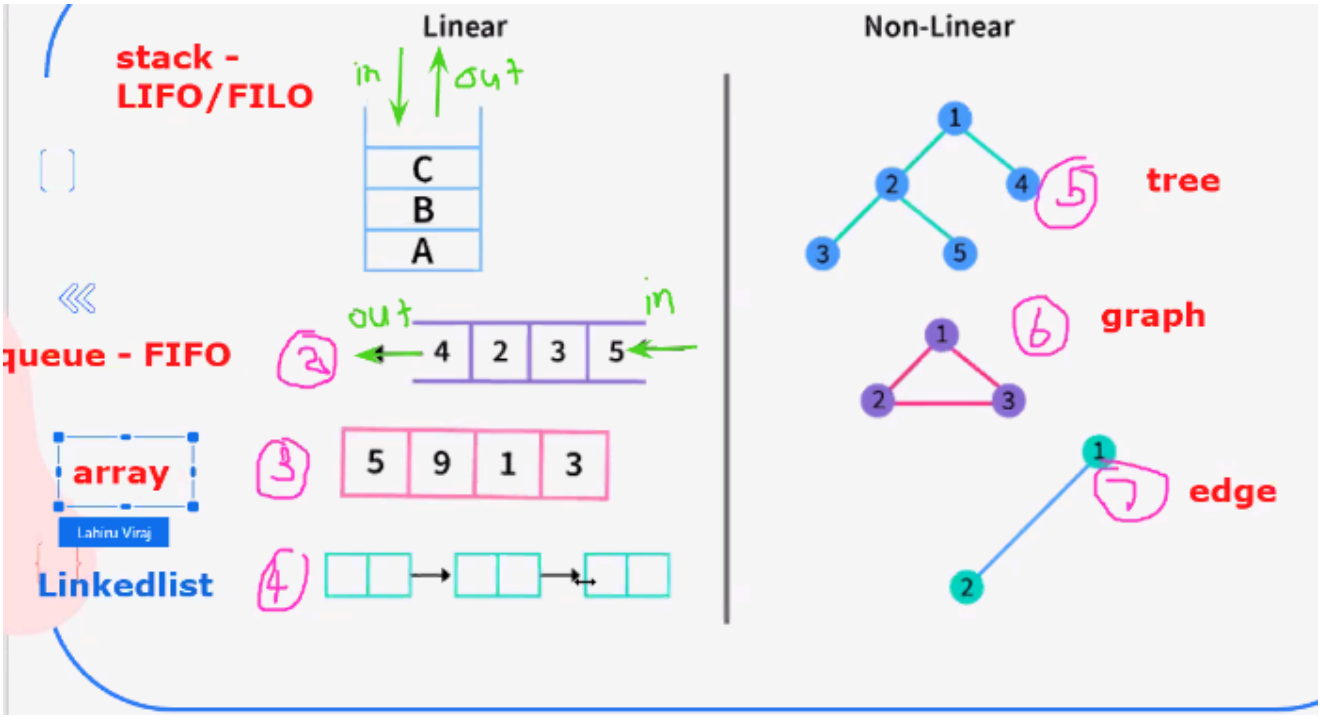
- Non-primitive data structures,unlike their primitive counterparts, can store data of multiple types. Examples include arrays, linked lists, stacks, queues, trees, and graphs. These are often turned derived data types.



Types of Data Structures

Data structures are divided into two categories:

- Linear data structure.
- Non-linear data structure.

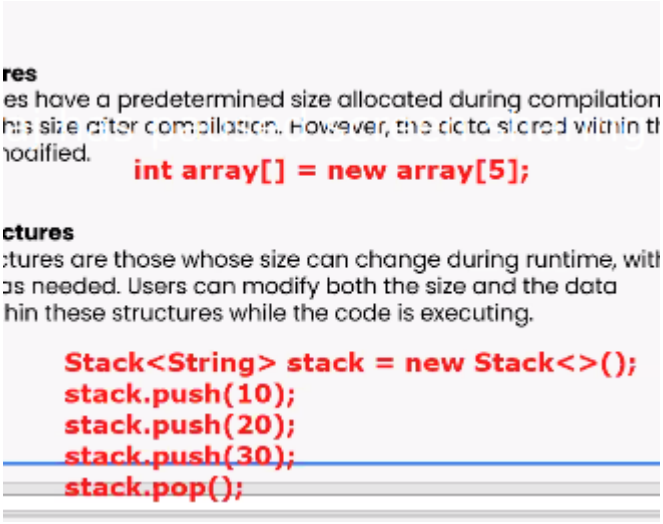


Linear data structures

A Linear Data Structure maintains a straight-line connection between its data elements, where each element is linked to its predecessors and successors, except for the first and last elements. However, the physical arrangement in memory may not follow this linear sequence.

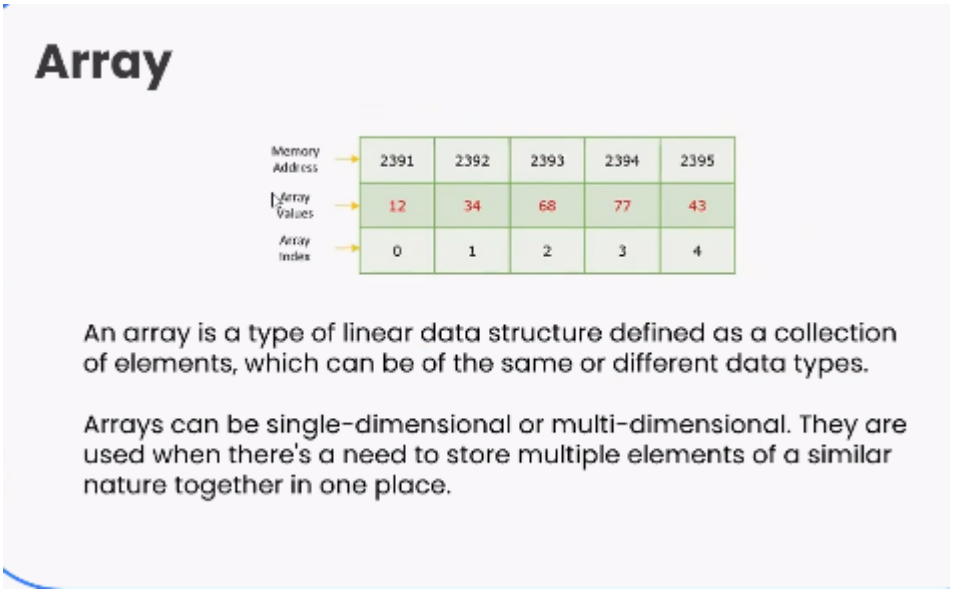
Linear Data Structures are categorized into two types based on memory allocation.

- **Static Data Structures**
  Static Data Structures have a predetermined size allocated during compilation, and users cannot alter this size after compilation. However, the data stored within these structures can be modified.
- **Dynamic Data Structures**
  Dynamic Data Structures are those whose size can Users can modify both the size and the data elements stored within these structures while the code is executing.

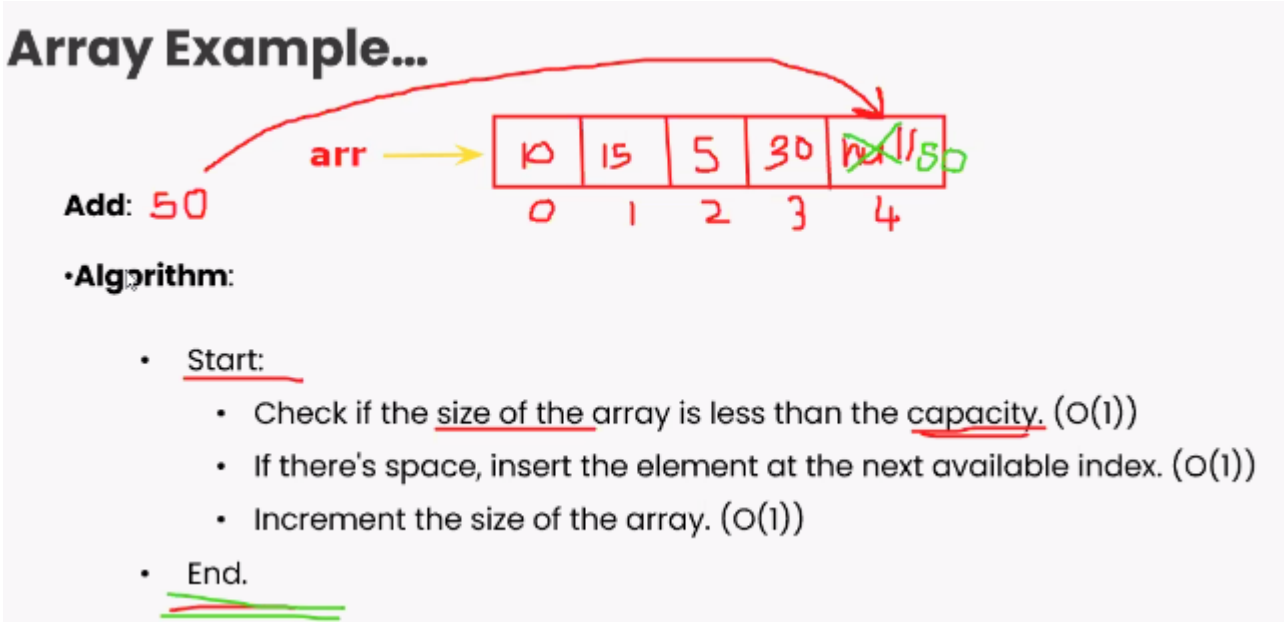


Non-Linear Data Structures

Non-linear data structures store data hierarchically,allowing it to exist at multiple levels and making traversal more complex than in linear structures, each element occupies an entire memory block, potentially wasting space if the element doesn't fully utilize the block. Non-linear data structures optimize memory usage,reducing space complexity.



0(1) random access lookup time



04 - Day

Making Array

```java
public class Array{
    private int[] data;
    private int size;
    private int capacity;

    public Array(int capacity){
        this.capacity = capacity;
        data = new int[capacity];
    }

    //Add
```

```
    public void add(int value){
        if(index >= 0 && size < capacity){
            for(int i = size;i > index; i--){
                data[i] = data[i-1];
            }
            data[index] = value;
            size++;
        }else{
            throw new ArrayIndexOutOfBoundsException();
        }
    }

    @Override
    public String toString(){
    String out = "[";
     for(int i = 0;i < size; i++){
        out += data[i];
        if(i == size -1){
            out += "]";
        }else{
            out += ", ";
        }
     }
     return out;
    }
}
```

Array Example 2

Find:

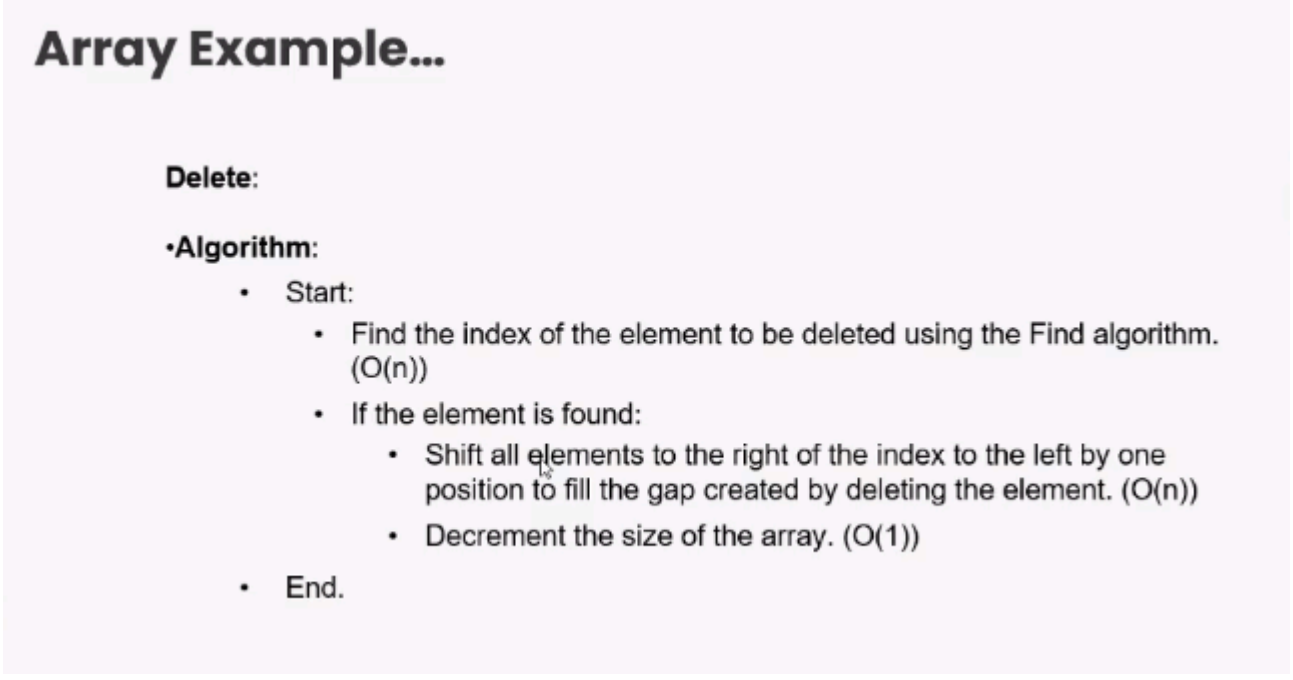
•Algorithm:

- Start:
  - Iterate over each element of the array. (O(n))
  - Compare each element with the given element. (O(1))
  - If a match is found, return the index. (O(1))
  - If no match is found, return -1. (O(1))
- End.

```
class Array{
private int[] arr;
private int arrSize;
```

```
public Array(){
}

public Array(int ArraySize){
    int arr = new int[arrSize];
    this.arrSize = ArraySize;
}
public int find(int givenElement){
    for(int i = 0;i < arr.length;i++){
        if(arr[i] === givenElement){
            return arr[i];
        }
    }
    return -1;
}
}

class Main{
public static void main(String args[]){
    Array array = new Array(10);
    array.
    array.find();

}
}
```

Delete Function



```
//Delete
public int Delete(int value){
Array arr = new Array(10);
//...

int index = array.find(value);
if(index == -1){
    throw new ArrayOutOfBoundsException();
}

for(int i = index; i< size; i++){
    data[i] = data[i+ 1];
}
data[--size] ==0;
}
```

Array - Insertion Operation

In the insertion operation, one or more elements are added to the array. Depending on the requirement, a new element can be added at the beginning, end, or at any given index of the array. This is typically accomplished using input statements in programming languages.

Insert at the end:
In an unsorted array, the insert operation is faster as compared to a sorted array because we don't have to care about the position at which the element is to be placed.

Time Complexity: O(1)
Space Complexity : O(1)

Insert at any position:
Insert operation in an array at any position can be performed by shifting elements to the right, which are on the right side of the required position

Time Complexity: O(n)
Space Complexity : O(1)

Array - Search Operation

In an unordered array, the elements are not arranged in any particular order. When you need to find a specific item in such an array, you typically have to check each element one by one until you find the desired item. This process is called linear traversal or linear search.

Search operation:
In an unsorted array, the search operation is less efficient compared to a sorted array because we cannot rely on the order of elements for quicker access.

Time Complexity: O(n)
Space Complexity : O(1)

Array - Delete Operation

During the delete operation, the element slated for removal is located via linear search, after which the deletion is executed, leading to a subsequent adjustment of the array by shifting the remaining elements.

Delete operation:
In an unsorted array, the deletion operation can be less efficient compared to a sorted array due to the need for linear search to locate the element to be deleted, followed by the subsequent adjustment of the array by shifting the remaining elements

Time Complexity: O(n)
Space Complexity : O(1)

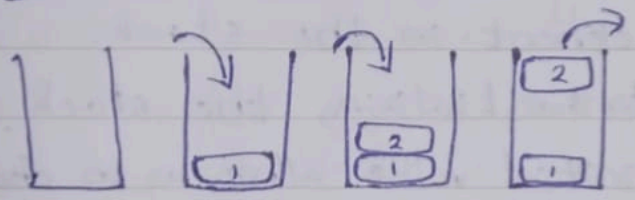
05 - Day

04/10/2023.

Stack Data structure.

A stack is a fundamental data structure in computer science that follows the Last In, First Out (LIFO) principle. It's similar to a stack of plates in a cafeteria: you can only take the top plate off, and to add a new plate, you place it on top of the stack. Stack is a Linear data structure

Stack Data structure



This structure supports two main operations.

- push
- pop

Additional auxilliary operations: ~~Peek or Top~~

- Peek or Top
- isEmpty
- size
- isFull

Working of Stack Data Structure

The operations for managing a stack work as follows:

Initialization:
- A pointer called **TOP** is used to keep track of the top element in the stack.
- When initializing the stack, set **TOP** to **-1**. This initial value allows us to check if the stack is empty by comparing **TOP == -1**.

Push Operation:
- Before pushing an element, check if the stack is already full. If the stack is not full, increase the value of **TOP** by **1**.
- Place the new element in the position pointed to by **TOP**.

Pop Operation:
- Before popping an element, check if the stack is already empty. If the stack is not empty, return the element pointed to by **TOP**.
- Decrease the value of **TOP** by **1**.

Implementing a Stack

```
public class Stack{
private int[] data;
private int capacity, size, top;

public Stack(int capacity){
    this.capacity = capacity;
    this.data = new int(capacity);
    this.top = -1;
}

public boolean isFull(){
    return top == capacity -1;
}

public void push(){
    if(!isFull()){
        data[++top] = value;
        size++;
    }else{
        throw new StackOverFlowError("Stack is full");
    }
}

public int pop(){
    if(!isEmpty()){
        size++;
```

```
        return data[top--];
    }else{
        throw new EmptyStackException();
    }
}

public int peek(){
    if(!isEmpty()){
    return data[top];
    }
}

public int size(){
    return size;
}
```

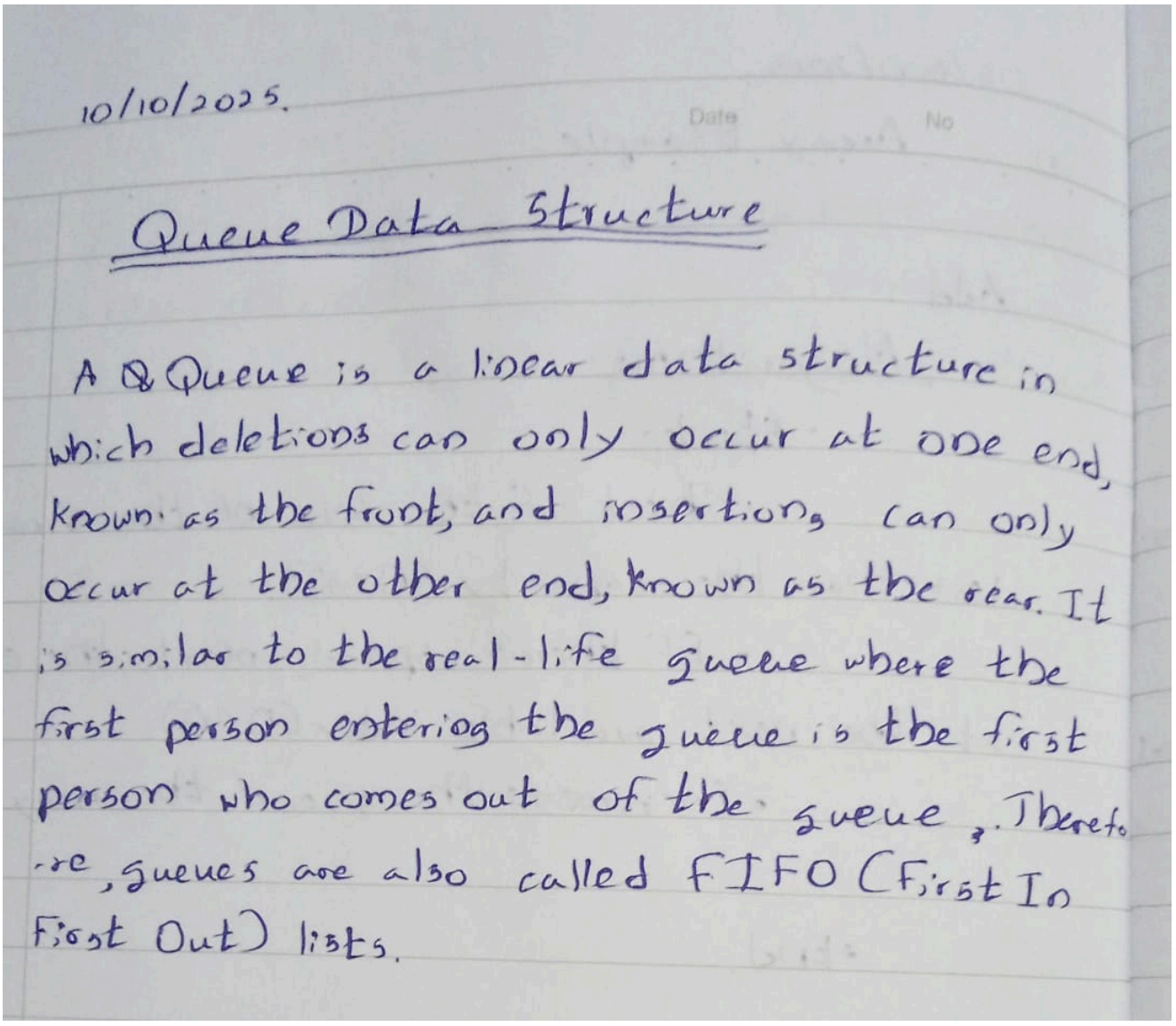
Stack implemented by Node

```
public class Node{
public int data;
public Node next;

public Node(int data){
    this.data = data;
    this.next = null;
}
}


public class Stack{
    private Node top;

    public Stack(){
        this.top = null;
    }

    public void push(int value){
        Node temp = new Node(value);
        temp.next = this.top;
        top = temp;
    }
```

```
    public int pop(){
        if(top != null){
            Node temp = top;
            top = top.next;
            int value = temp.data;

            temp = null;
            size--;
            retun value;
        }else{
            return -1;
        }
    }

    @Override
    public String toString(){
    StringBuffer sb = new StringBuffer();
    sb.append("Stack (top -> bottom)");

    Node current = top;
    while(current != null){
    sb.append(current.data);
    if( current.next != null) sb.append(" -> ");
    current = current.next;
    }

    return sb.toString();
    }
}
```

06 - Day

Queue Data Structure



10/10/2025.

Date No

Queue Data Structure

A Queue is a linear data structure in which deletions can only occur at one end, known as the front, and insertions can only occur at the other end, known as the rear. It is similar to the real-life queue where the first person entering the queue is the first person who comes out of the queue. Therefore, queues are also called FIFO (First In First Out) lists.

Basic Operations of Queue

A queue allows the following basic operations:

- **Enqueue:** Add new element to the end of the queue.
- **Dequeue:** Remove an element from the front of the queue.

Additionally, queue typically support auxiliary operations such as:

- **Peek:** Get the value of the front of the queue without removing it.
- **isEmpty:** Checks if the queue is empty.
- **IsFull:** Check if the queue is full.

Every queue includes two pointers, **FRONT** and **REAR**, which indicate the position where deletion and insertion can be performed. Initially, values of **FRONT** and **REAR** is set to **-1**.

Working of Queue Data Structure

The operations for managing a queue data structure are as follows:

Enqueue Operation:

- Before adding an element, check if the queue is already full.
- If the queue is not full, proceed as follows:
  - If the queue is empty, set **FRONT** to **0**.
  - Increase the value of **REAR** by **1.**
  - Place the new element in the position pointed to by REAR.

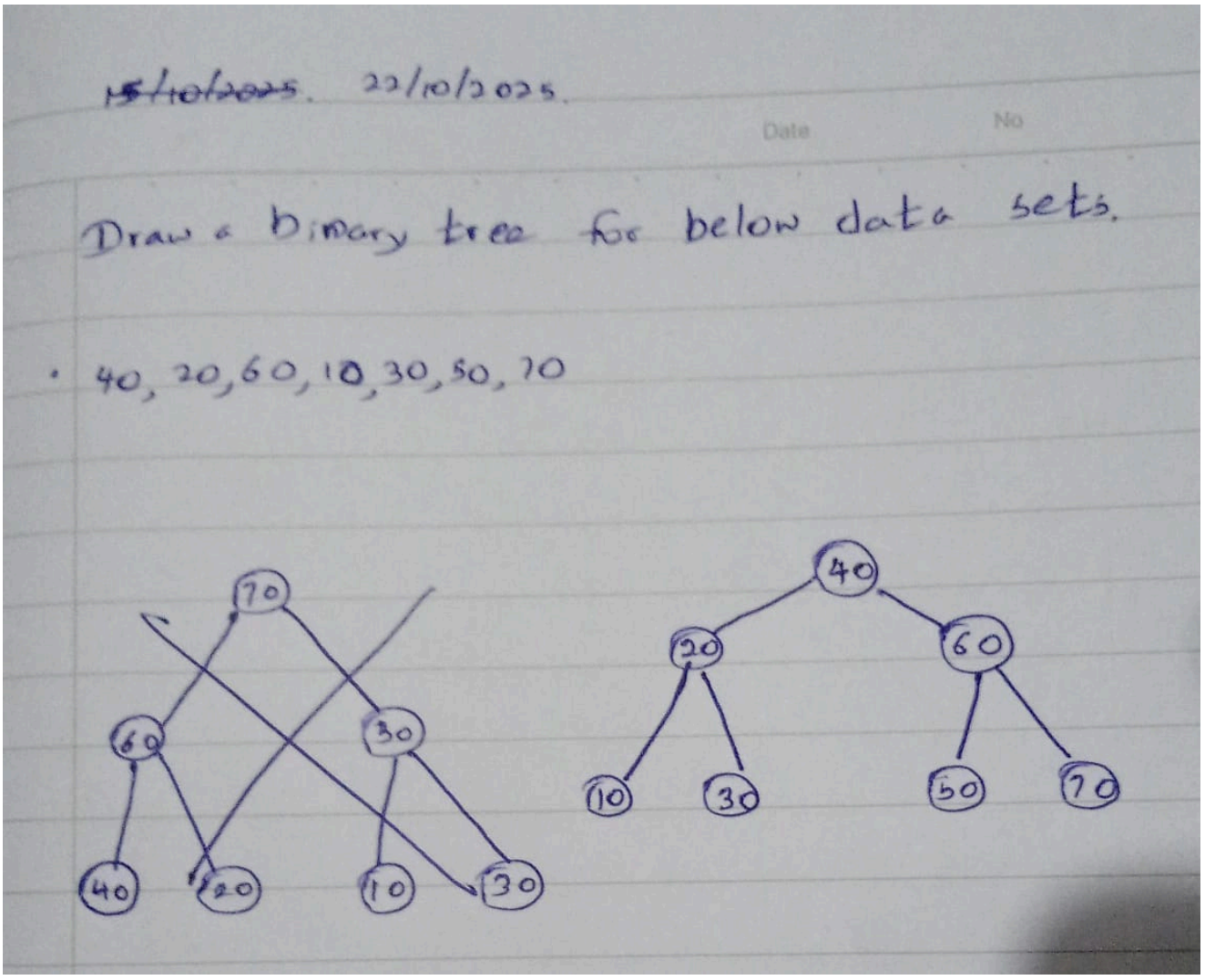
Dequeue Operation:

- Before removing an element, check if the queue is already empty.
- If the queue is not empty, proceed as follows:
  - Return the element pointed to by **FRONT**.
  - If **FRONT** is equal to **REAR**, reset both **FRONT** and **REAR** to **-1** indicate that the queue is empty.
  - Otherwise, increase the value of **FRONT** by **1**.

Types of Queues

There are four different types of queues:

- Simple Queue
- Circular Queue
- Priority Queue
- Double Ended Queue(Deque)



07 - Day

Working of Circular Queue Data Structure

The operations for managing a Circular Queue data structure are as follows:

Initialization:

- Allocate memory for an array **items** to hold the queue elements.
- Initialize **FRONT** and **REAR** pointers to **-1** to indicate that the queue is empty.
- Set the **size** of the queue to **0**.
- Store the maximum number of elements (**capacity**) that the queue can hold.

Working of Circular Queue Data Structure

The operations for managing a Circular Queue data structure are as follows:

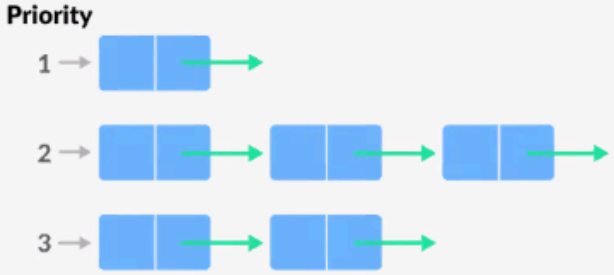
Enqueue Operation:

- If **(REAR + 1) % capacity == FRONT**, the queue is full
- If the queue is empty **(FRONT == -1)**, set **FRONT** to **0**.
- Update **REAR** to the next position in a circular manner using **(REAR + 1) % capacity**.
- Insert the new element at the position pointed to by **REAR**.
- Increase the **size** of the queue by **1**.

Dequeue Operation:

- If **FRONT == -1**, the queue is empty.
- Retrieve the element at the position pointed to by **FRONT**.
- If **FRONT** equals **REAR**, reset both **FRONT** and **REAR** to **-1** to indicate that the queue is now empty.
- Otherwise, update **FRONT** to the next position in a circular manner using **(FRONT + 1) % capacity**.
- Decrease the **size** of the queue by **1**.

Priority Queue



- Nodes have predefined priorities.
- Node with the least priority is removed first.
- Insertion based on the order of node arrival.
- Used in algorithms like Dijkstra's shortest path and Prim's algorithm, and in data compression techniques like Huffman coding.

Insertion occurs based on the arrival of the values and removal occurs based on priority.

Working of Deque Data Structure

The operations for managing a deque data structure are as follows:

Insertion Operations:

- **Insert at Front (push_front or addFirst):** Adds an element to the front of the deque.
- **Insert at Back (push_back or addLast):** Adds an element to the back of the deque.

Deletion Operations:

- **Remove from Front (pop_front or removeFirst):** Removes and returns the element from the front of the deque.
- **Remove from Back (pop_back or removeLast):** Removes and returns the element from the back of the deque.

08 - Day

Linked list Data Structure

- a linked list is a linear data structure that includes series of connected nodes .Here each node stores the data and the address of the next node.
- Unlike arrays, linked lists allow for efficient insertion and deletion of elements because they do not require elements to be stored in contiguous memory locations

Types of Linked Lists

1. **Singly Linked List :** Each node points to the next node. The last node points to **null**.
2. **Doubly Linked List :** Each node has two pointers, one pointing to the next node and another pointing to the previous node.
3. **Circular Linked List :** The last node points back to the first node, forming a circle.
    - **Singly Circular Linked List :** Only the next pointer connects to the first node.
    - **Doubly Circular Linked List :** Both next and previous pointers connect accordingly.

Basic Concepts

- Node : the basic unit of a linked list.Each node contains:
    - Data: The value stored in the node.
    - Pointer (at reference) : the address of the next node in the sequence.
- Head: The first node in a linked list.
- Tail : The last node in a linked list.Whose pointer is typically null.

Operations on Linked Lists.

- **Insertion :** Adding a new node to the list.
    - At the beginning.
    - At the end.
    - At a specific position.
- **Deletion :** Removing a node from the list.
    - From the beginning.
    - From the end.
    - From a specific position.
- **Traversal :** Accessing each node in the list to perform an operation.
- **Searching :** Finding a node with a specific value.
- **Reversal :** Reversing the order of nodes in the list.

Insertion at the End :

- Create a new node with the given data.
- If the list is empty, make the new node the head.
- Otherwise, traverse to the last node.
- Set the last node's next to the new node.

Insertion at the Beginning :

- Create a new node with the given data.
- Set the new node's next to the current head.
- Update the head to be the new node.

Insertion at a Specific Position :

- If the position is 0, perform insertion at the beginning.
- Otherwise, traverse to the node just before the specified position.
- Create a new node with the given data.
- Set the new node's next to the current node's next.
- Set the current node's next to the new node.

Deletion by Value :

- If the list is empty, return.
- if the head node has the specified data, update the head to be the next node.
- Otherwise, traverse the list to find the node with specified data.
- Update the next pointer of the previous node to skip the node with specified data.

Date No

Operation in a singly Linked list.

Deletion by position:

- If the list is empty, return.
- If the position is 0, update the head to be the next node.
- Otherwise, traverse to the node just before the specified position.
- Update the next pointer of the previous node to skip the node at the specified position.

Traversal:

- start from the head node.
- visit each node, processing the node's data as needed.
- Move to the next node using the next pointer.
- Continue until the end of the list (when the next node is null)

Search:

- Start from the head node.
- Traverse the list, comparing each node's data with the target data.
- If a node with the target data is found, return true.
- If the end of the list is reached without finding the target data, return false.

Reverse:

- Initialize three pointers: previous, current and next.
- Start previous to null and current to the head node.
- Loop through the list:
  - Save the next node (next = current.next)
  - Reverse the next pointer of the current node (current.next = previous).
  - Move previous and current one step forward (previous = current, current = next).
- Update the head to be the previous node (the new head of the reversed list).

These algorithms provide a structured approach to manipulating a singly linked list, adhering to standard DSA principles. The itme complexity for each operation varies.

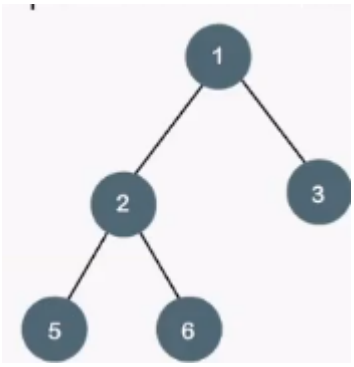
- Insertion at the end : O(n)
- Insertion at the beginning : O(1)
- Insertion at a specific position : O(n)
- Deletion by value : O(n)
- Deletion by position : O(n)
- Traversal : O(n)
- Search : O(n)
- Reverse : O(n)

09 - Day

Tree Data Structure

a tree is a nonlinear hierarchical data structure that consists of nodes connected by edges.

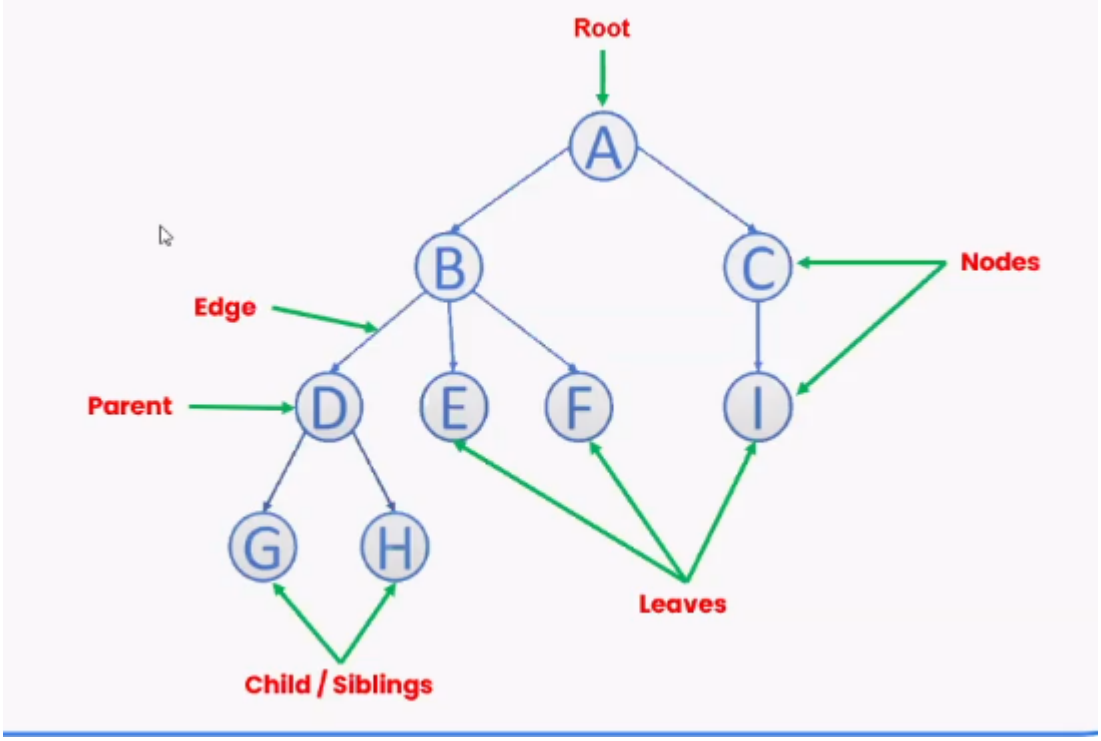
a tree is a widely used abstract data type (ADT) that simulates a hierarchical tree structure with a set of linked nodes.it is a non linear data structure, as opposed to arrays, linked lists, stacks, and queues which are linear data structures.



Key Terms

- **Node :** The fundamental part of a tree. Each node contains a value or data, and it may have a link to other nodes.
- **Root :** the top node of a tree. it does not have a parent.
- Edge : The connection between two nodes.
- Child : A node directly connected to another node when moving away from the root.
- Parent : A node directly connected to another node when moving towards the root.
- Leaf : A node that does not have any children.
- Siblings : A nodes that belong to the same parent.
- Depth : The depth of a node is the number of edges from the node to the tree's root node.

- Height : The height of a node is the number of edges on the longest path from the node to a leaf.
- Degree : The number of children a node has.
- Subtree : A tree consisting of a node and its descendants.



Types of Trees

- Binary Tree : Each node has at most two children.
- Binary Search Tree (BST) : A binary tree where the left child contains values less than the parent and the right child contains values greater than the parent.
- AVL Tree : A self-balancign binary search tree where the height of the two child subtrees of any node differs by at most one.
- Red-Black Tree : A self-balancing binary search tree with additional properties to ensure the tree remains balanced.
- B- Tree : A self-balancing search tree in which nodes can have multiple children, often used in databases and file systems.
- Trie : A tree like data structure used to store a dynamic set of strings, where the keys are usually strings.

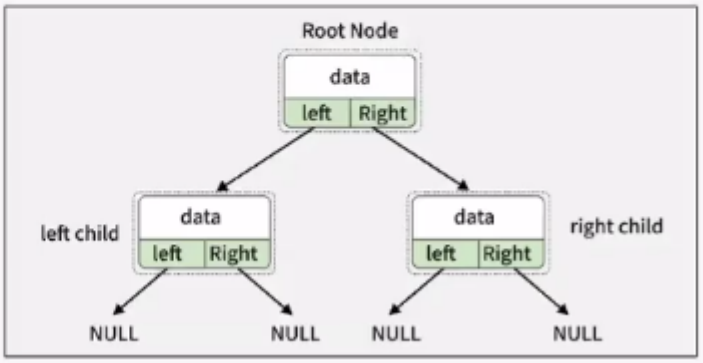
Basic Operations

- Insertion : Adding a new node to the tree.
- Deletion : Removing a node from the tree.
- Traversal : Visiting all the nodes in a specific order.
  - In-order Traversal : Left, Root, Right (for BST, gives sorted order)
  - Pre-order Traversal : Root, Left, Right.

- Post-order Traversal : Left, Right, Root.

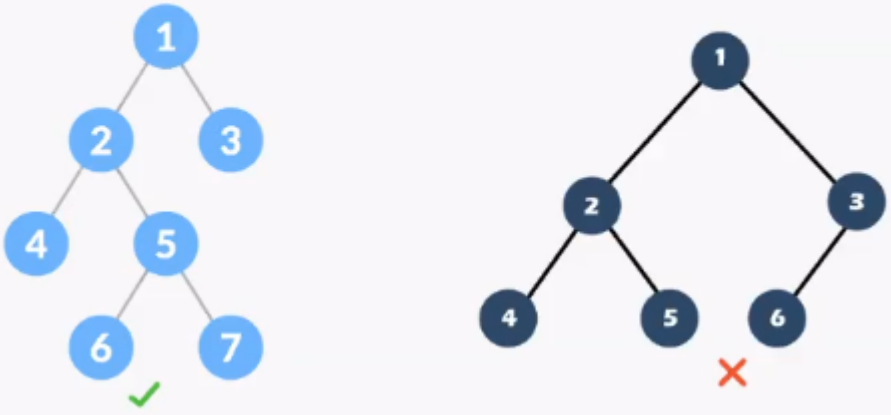
Binary Tree

A Binary Tree Data Structure is a hierarchical data structure in which each node has at most two children, referred to as the left child and the right child. It is commonly used for efficient storage and retrieval of data, with various operations such as insertion, deletion, and traversal.



Full Binary Tree

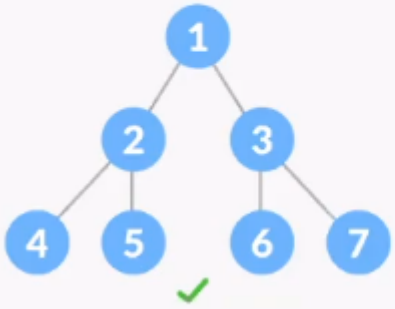
A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children. Also called as proper binary tree.

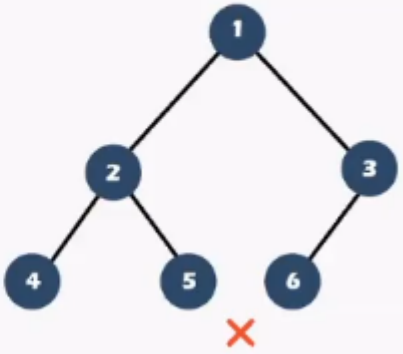


Perfect Binary Tree

A perfect binary tree is a type of binary tree in which every internal node has exactly two child nodes and all the leaf nodes are at the same level.

Leaf Nodes count = Internal Node Count + 1

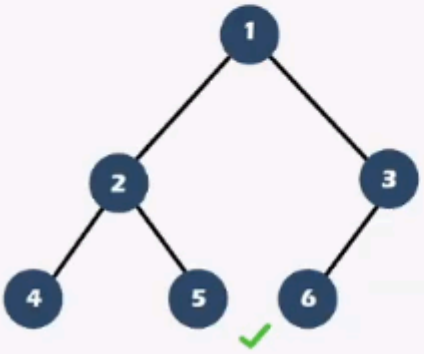


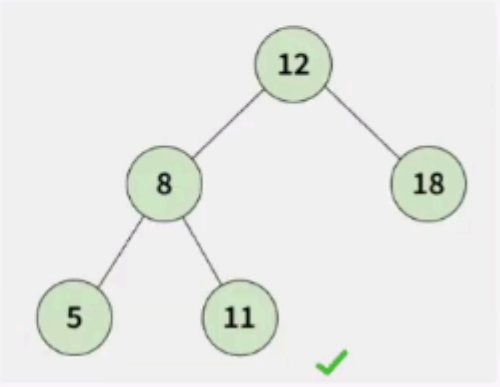


Complete Binary Tree

A complete binary tree is just like a full binary tree, but with two major differences

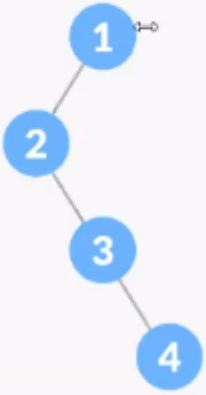
- Every level must be completely filled
- All the leaf elements must lean towards the left.
- The last leaf element might not have a right sibling.





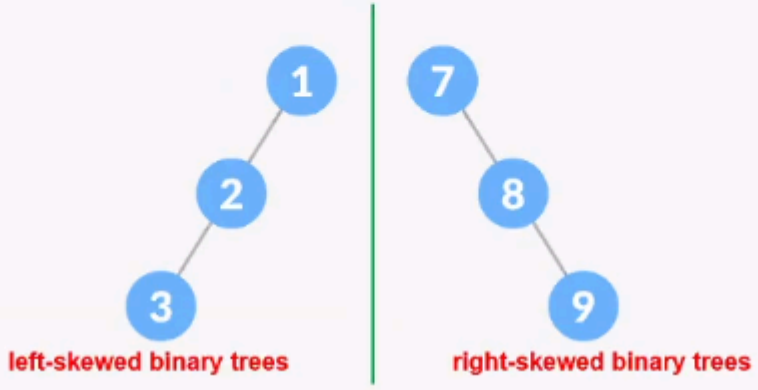
Degenerate or Pathological Tree

A degenerate or pathological tree is the tree having a single child either left or right.



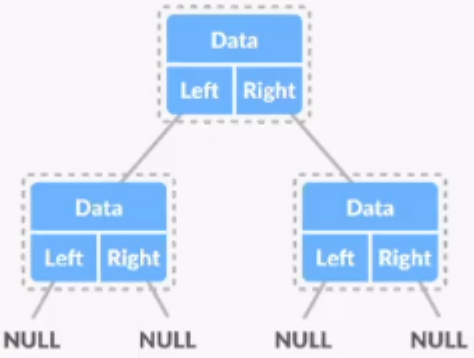
Skewed Binary Tree

A skewed binary tree is a pathological or degenerate form of a binary tree in which all the nodes are aligned either to the left or the right. Therefore, there are two types of skewed binary trees: **left-skewed binary trees**, where each node has only a left child, and **right-skewed binary trees**, where each node has only a right child.

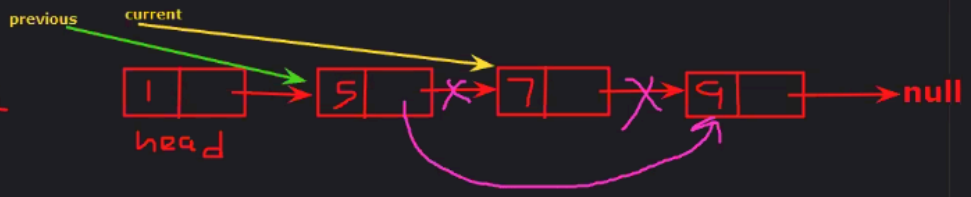


Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.



```java
//Deletion by value
public void deleteAtValue(int value){  no usages
    if(head == null){
        System.out.println("List is empty");
        return;
    }

    if(head.data == value){
        head = head.next;
        return;
    }

    Node current = head;
    Node previous = null;
    while (current != null && current.data != value){
        previous = current;
        current = current.next;
    }

    if(current == null){
        System.out.println("Value not found");
        return;
    }

    previous.next = current.next;  Tab to complete
}
```

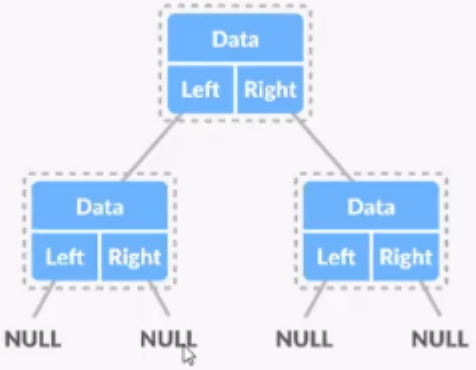


Lahiru Viraj has paused screen sharing

10 - Day

Binary Tree Representation

A node of a binary tree is represented by a structure containing a data part and two pointers to other structures of the same type.

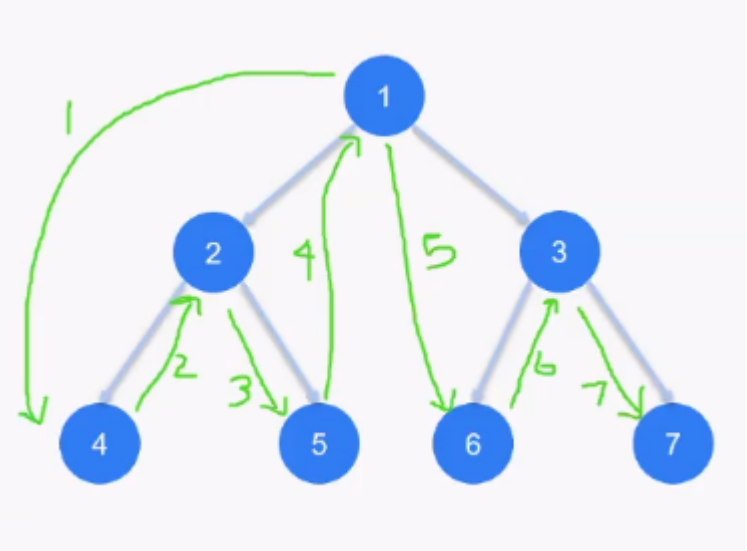


Inorder Traversal

Inorder traversal is a depth-first traversal method used primarily for binary trees.

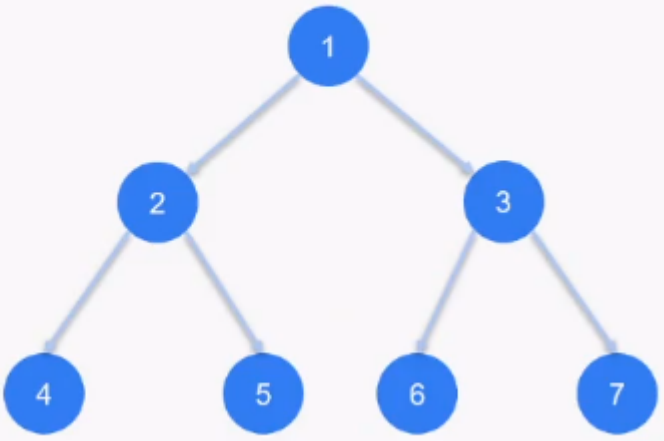
In this method, nodes are visited in the following order:

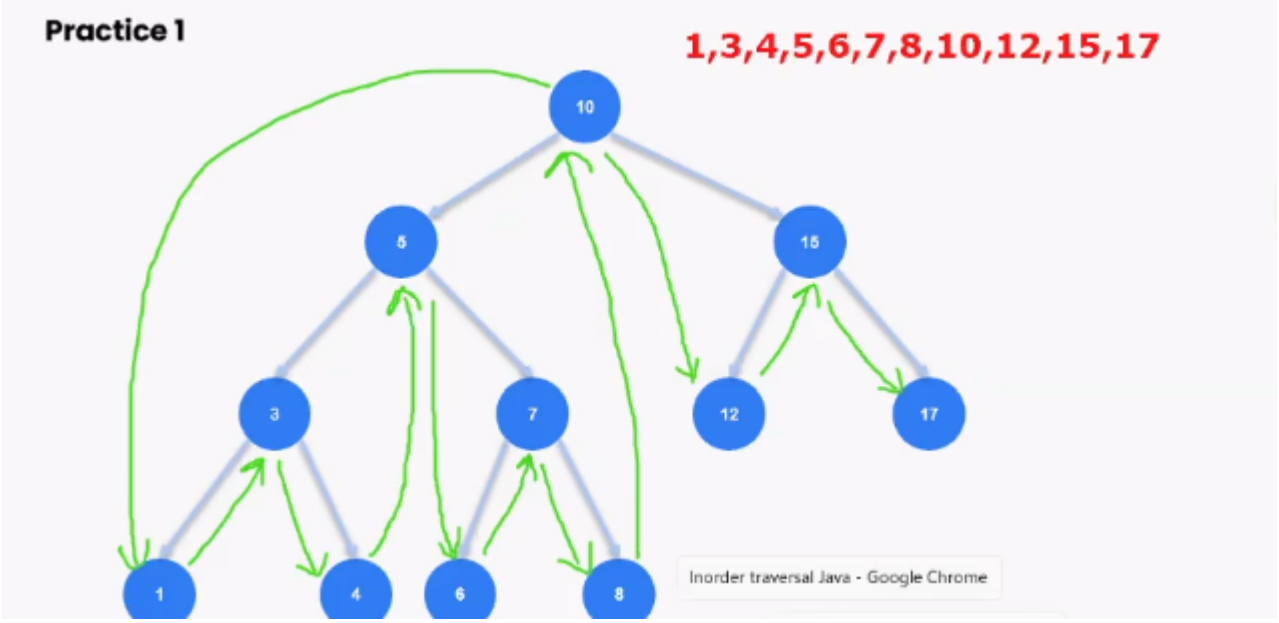
1. Visit the left subtree. **Inorder(left subtree)**
2. Visit the root node.
3. Visit the right subtree. **Inorder(right subtree)**

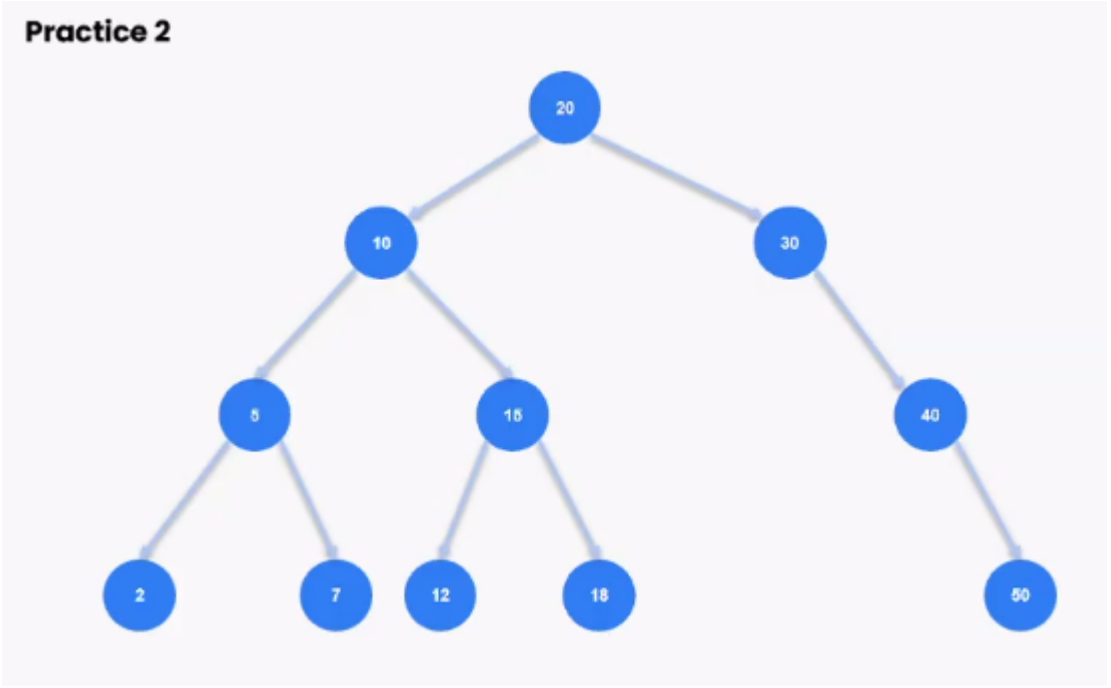


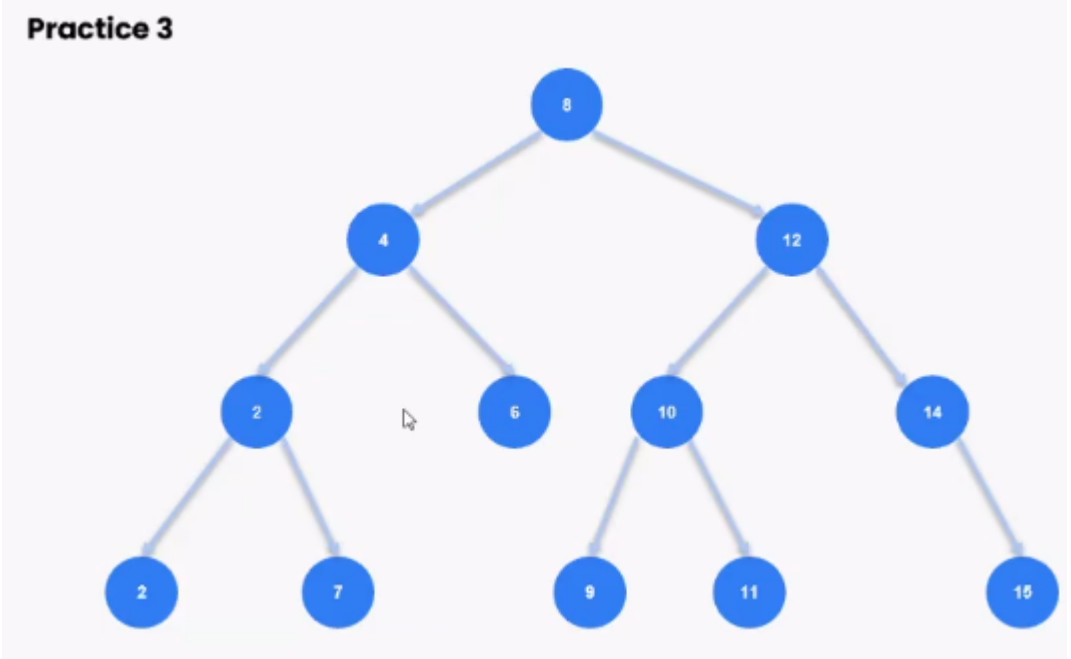
Try to do Inorder Traversal

4,2,5,1,6,3,7

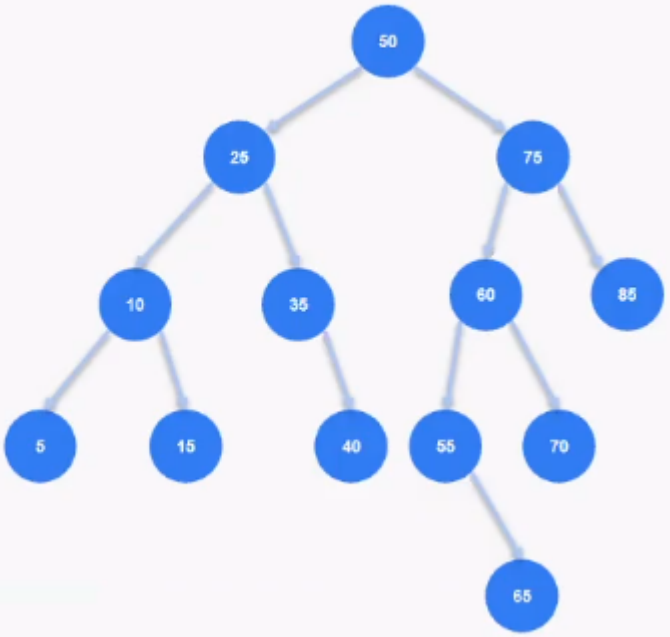



Practice 1
1,3,4,5,6,7,8,10,12,15,17
10
5
15
3
7
12
17
1
4
6
8
Inorder traversal Java - Google Chrome


Practice 2
20
10
30
5
15
40
2
7
12
18
50


Practice 3
8
4
12
2
6
10
14
2
7
9
11
15

Practice 4

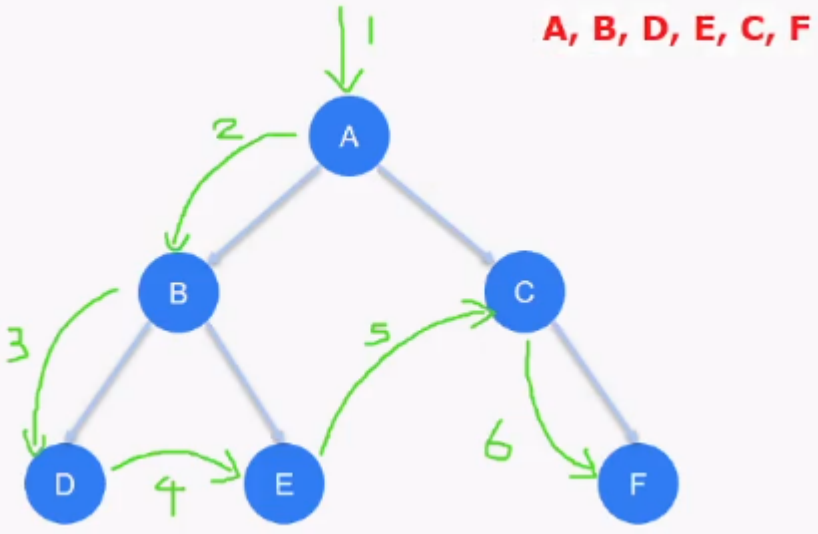


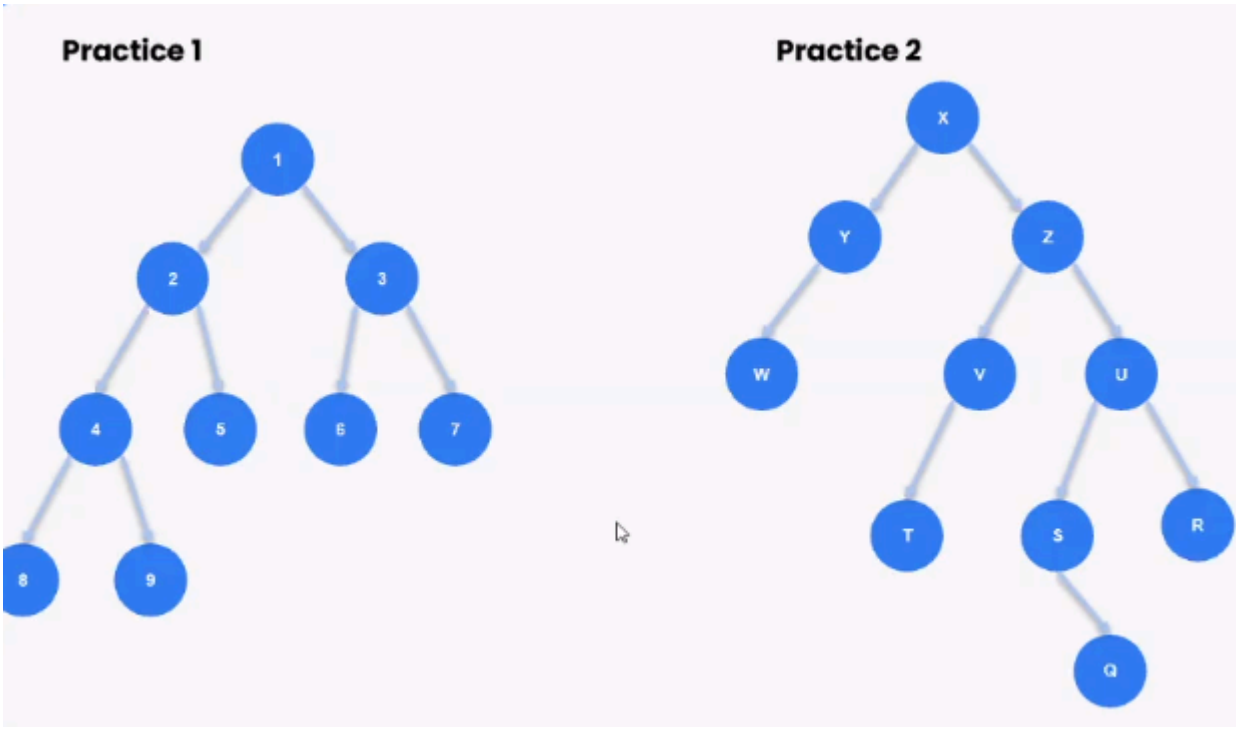
Preorder Traversal

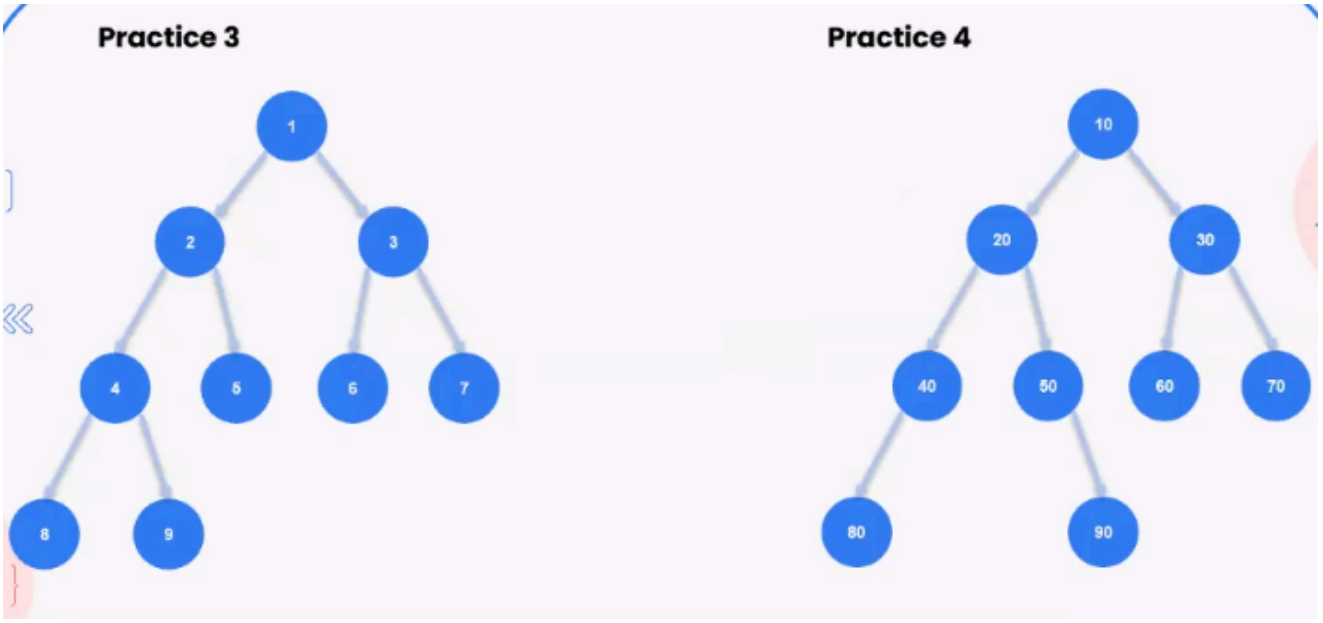
Preorder traversal is a tree traversal method where you visit the root node first, then recursively visit the left subtree, and finally, the right subtree. The steps for preorder traversal can be summarized as:

1. Visit the **root** node.
2. Traverse the **left subtree** in **preorder**.
3. Traverse the **right subtree** in **preorder**.

Try to do Preorder Traversal







12 - Day

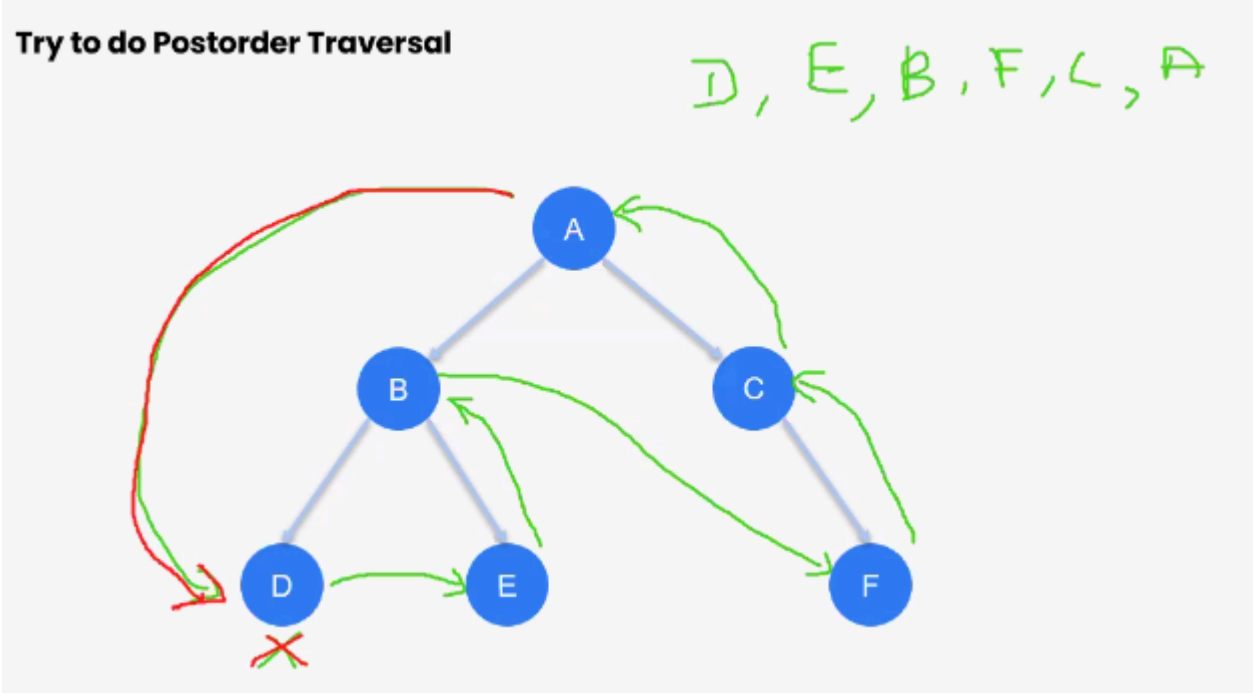
Postorder Traversal

Postorder traversal is a type of depth-first traversal for trees where the nodes are recursively visited in the following order:

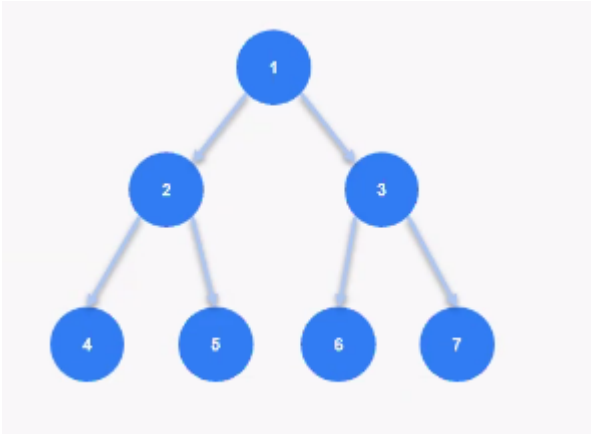
left subtree, right subtree, root node.

This means that each node is visited only after its children have been visited.

1. Traverse the left subtree by recursively applying postorder.
2. Traverse the right subtree by recursively applying postorder.
3. Visit the root node.

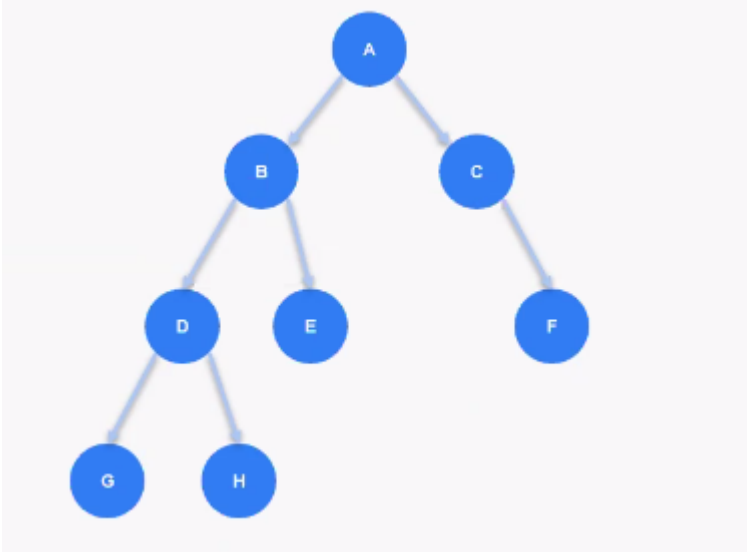


Practice 1



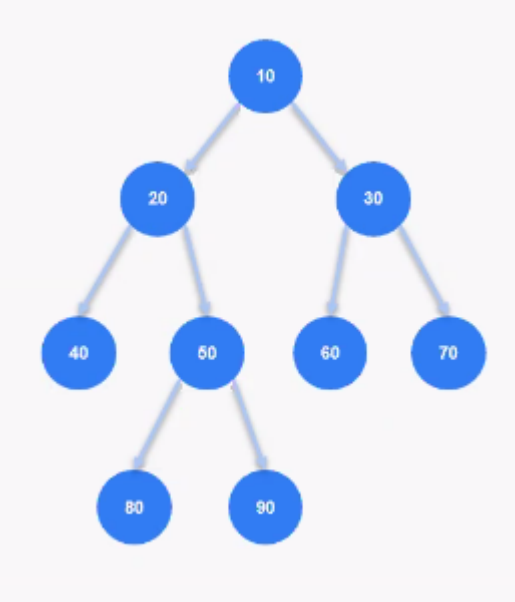
- answer : 4,5,2,6,7,3,1

Practice 2



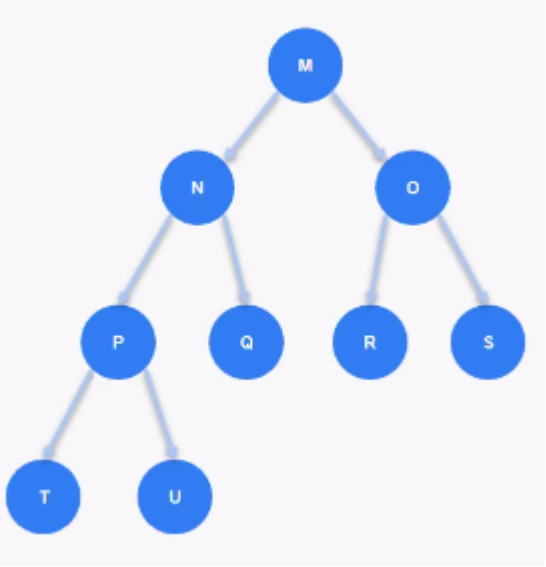
- answer : G,H,D,E,B,F,C,A

Practice 3



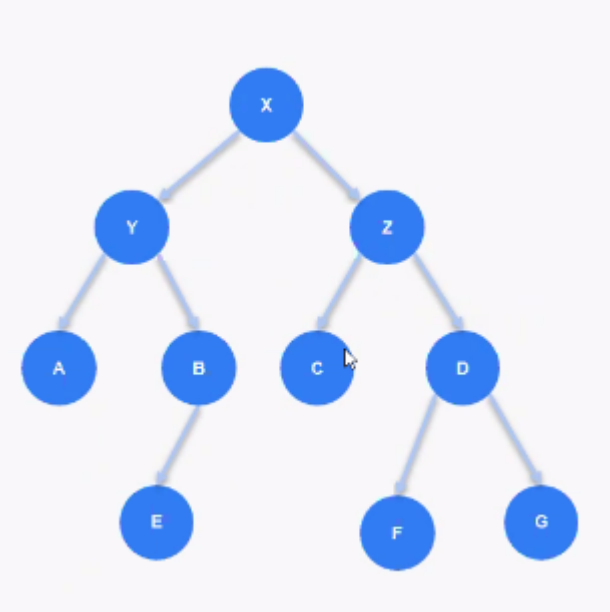
- answer : 40,80,90,50,20,60,70,30,10

Practice 4



- answer : T,U,P,Q,N,R,S,O,M

Practice 5



- answer : A,E,B,Y,C,F,G,D,Z,X

Binary Search Tree(BST)

A binary search tree (**BST**) is a data structure that efficiently maintains a sorted collection of numbers.

- It is termed a binary tree because each node can have at most two children.
- It is referred to as a search tree because it supports fast searching for the presence of a number in **O(log(n))** time.

The distinguishing properties of a binary search tree include:

- All nodes in the left subtree are less than the root node.
- All nodes in the right subtree are greater than the root node.
- Both the left and right subtrees of each node are also BSTs, adhering to the above two properties.

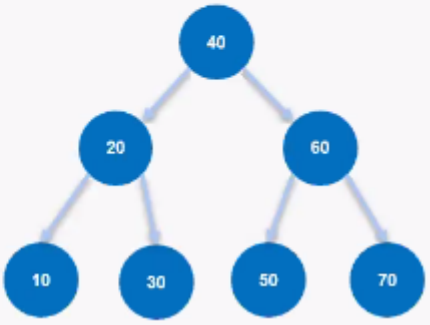
Binary Search Tree(BST)

Node Structure: Each node in the tree contains a key (and possibly associated data) and has at most two children referred to as the left child and the right child.

Ordering Property: For any given node with key **k**:

- All keys in the **left subtree** of **k** are less than **k**.
- All keys in the **right subtree** of **k** are greater than **k**.

No Duplicate Nodes: Each key in the BST is unique.



Basic Operations on a BST

Search:

- Start at the root and compare the key to be searched with the key of the current node.
- If the key is equal, the search is successful.
- If the key is less, search the left subtree.
- If the key is greater, search the right subtree.
- Repeat until the key is found or the subtree becomes null.

Insertion:

- Start at the root and compare the key to be inserted with the key of the current node.
- If the key is less, recursively insert it into the left subtree.
- If the key is greater, recursively insert it into the right subtree.
- If the position is found (null), insert the new node there.

Basic Operations on a BST

Deletion:

There are three cases to consider when deleting a node:

- The node is a leaf (no children): Simply remove the node.
- The node has one child: Remove the node and link its parent to its child.
- The node has two children: Find the in-order predecessor (maximum node in the left subtree) or in-order successor (minimum node in the right subtree) to replace the node, then delete the predecessor or successor node.

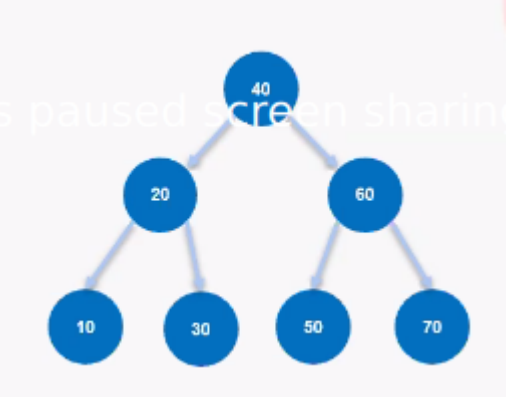
Traversal:

- **In-order (Left, Root, Right):** Produces a sorted list of keys.
- **Pre-order (Root, Left, Right):** Useful for copying the tree.
- **Post-order (Left, Right, Root):** Useful for deleting the tree.

13 - Day

Draw binary tree for below data sets

40,20,60,10,30,50,70



Draw binary tree for below data sets

10,20,30,40,50,60,70

Or

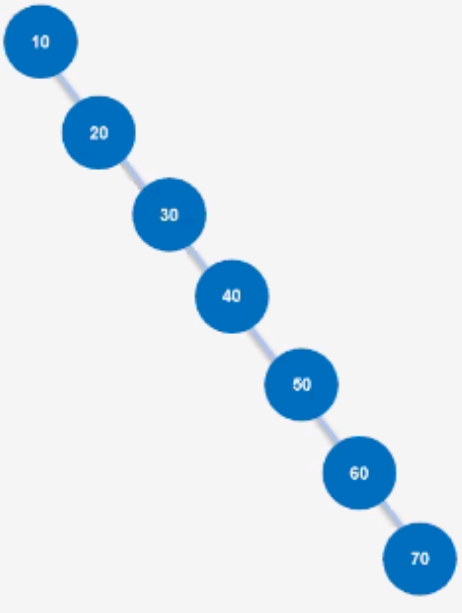
70,60,50,40,30,20,10

Draw binary tree for below data sets

10,20,30,40,50,60,70

Or

70,60,50,40,30,20,10



AVL Tree

An **AVL (Adelson-Velsky and Landis)** tree is a self-balancing binary search tree where the difference in heights between the left and right subtrees of any node is at most one.

This balancing ensures that the tree remains approximately balanced, leading to better performance for insertion, deletion, and lookup operations compared to unbalanced binary search trees.

Properties of AVL Trees

Balance Factor:

For every node in the AVL tree, the difference in heights (balance factor) of the left and right subtrees is -1, 0, or 1. If a node's balance factor becomes less than -1 or greater than 1, the tree needs to be rebalanced.

Height:

The height of an AVL tree with n nodes is **O(log n),** ensuring that operations like search, insert, and delete are efficient.

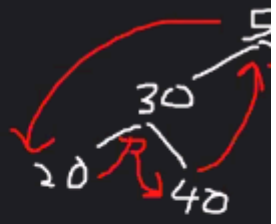
Balance Factor

The balance factor of a node in an AVL tree is defined as the difference between the heights of its left subtree and its right subtree:

Balance Factor = *Height of Left Subtree - Height of Right Subtree*

- **Balance Factor = 0**: The heights of the left and right subtrees are equal.
- **Balance Factor = 1**: The height of the left subtree is greater than the height of the right subtree by 1.
- **Balance Factor = -1**: The height of the right subtree is greater than the height of the left subtree by 1.

```java
public class Main {
    public static void main(String[] args) {
        System.out.println("Inorder :");
        bst.inOrder();

        System.out.println("Preorder :");
        bst.preorder();

        System.out.println("Postorder :");
        bst.postorder();

        // Search examples
        System.out.println("\nSearch 40: " + bst.search( value: 40));
        System.out.println("Search 90: " + bst.search( value: 90));

        // Deletion examples
        System.out.println("\nDeleting 20...");
        bst.delete( value: 20);
        bst.inOrder();

        System.out.println("Deleting 30...");
        bst.delete( value: 30);
        bst.inOrder();

        System.out.println("Deleting 50...");
        bst.delete( value: 50);
        bst.inOrder();
    }
}
```



main

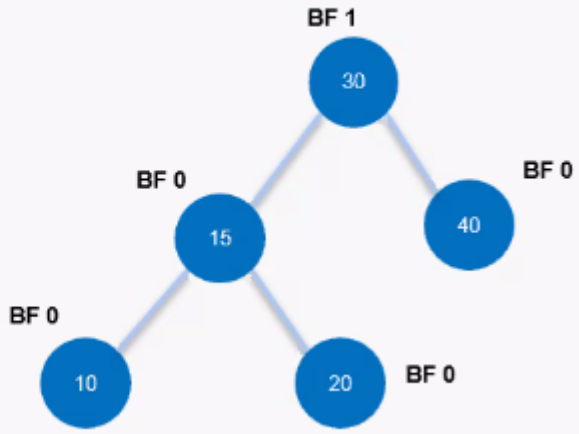
14 - Day

Rotating the subtrees in an AVL Tree

In rotation operation, the positions of the nodes of a subtree are interchanged.

- Left Condition (LL)
- Right Condition (RR)
- Left – Right Condition (LR)
- Right – Left Condition (RL)

40,20,30,15,10,5



15 - Day

Priority Queue

A priority queue is a specialized queue where each element is assigned a priority value. Elements are processed based on their priority, with higher priority elements served before lower priority ones. In cases where elements share the same priority, they are processed according to their order in the queue.

Difference between Priority Queue and Normal Queue

In a queue, the **first-in-first-out** rule is implemented whereas, in a priority queue, the values are removed on the **basis of priority**. The element with the highest priority is removed first.

Implement Priority Queue

Priority queue can be implemented using the following data structures:

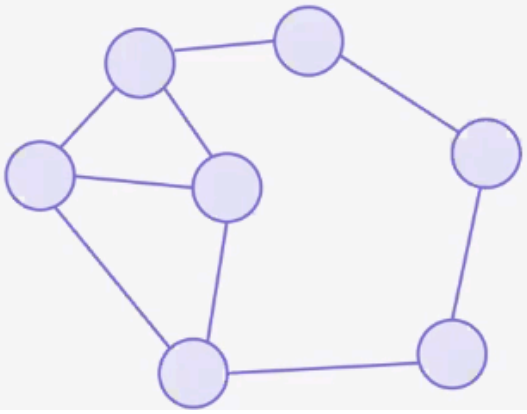
- Arrays
- Linked list
- Binary search tree

Operations	peek	insert	delete
Linked List	O(1)	O(n)	O(1)
Binary Search Tree	O(1)	O(log n)	O(log n)

Graph Data Structure

A Graph in Data Structure is a non-linear data structure that consists of nodes and edges which connects them.

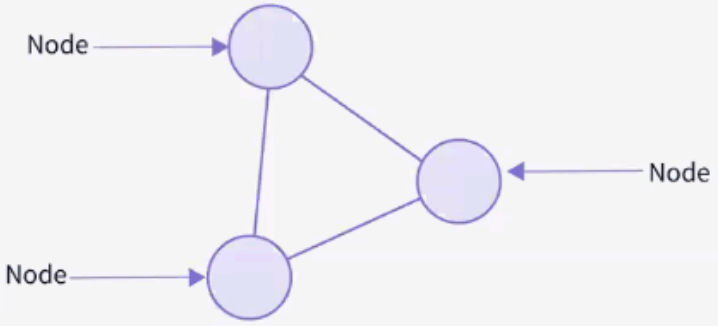
A Graph in Data Structure is a collection of nodes that consists of data and are connected to other nodes of the graph.



Nodes / Vertices

Nodes create complete network in any graph. They are one of the building blocks of a Graph in Data Structure. They connect the edges and create the main network of a graph. They are also called vertices. A node can represent anything such as any location, port, houses, buildings, landmarks, etc.

They basically are anything that you can represent to be connected to other similar things, and you can establish a relation between the them.



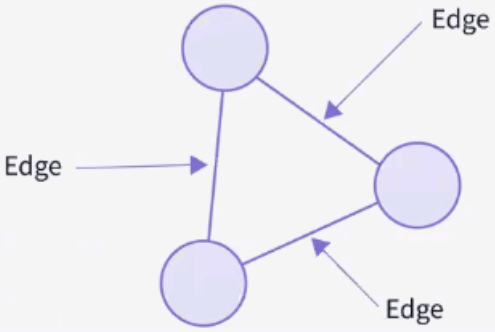
Edges

Edges

Edges basically connects the nodes in a Graph in Data Structure. They represent the relationships between various nodes in a graph. Edges are also called the path in a graph.

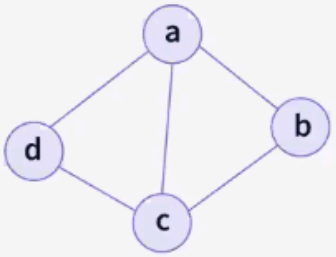
A graph data structure **(V, E)** consists of:

- A collection of vertices **(V)** or **nodes**.
- A collection of edges **(E)** or **path**



Example

The below represents a set of edges and vertices



A graph is a pair of sets (V, E), where V is the set of vertices and E is the set of edges, connecting the pairs of vertices.

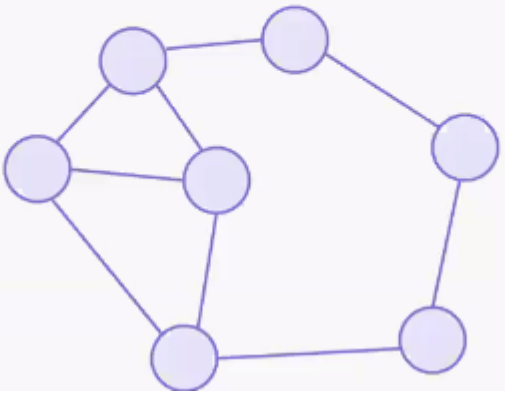
$V = \{a, b, c, d\}$
$E = \{ab, ac, ad, bc, cd\}$

In this graph, **|V| = 4** because there are four nodes (vertices) and, **|E| = 5** because there are five edges (lines).

Graph Data Structure

A Graph in Data Structure is a non-linear data structure that consists of nodes and edges which connects them.

A Graph in Data Structure is a collection of nodes that consists of data and are connected to other nodes of the graph.

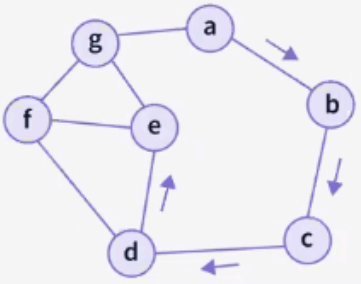


Graph Terminology

Path

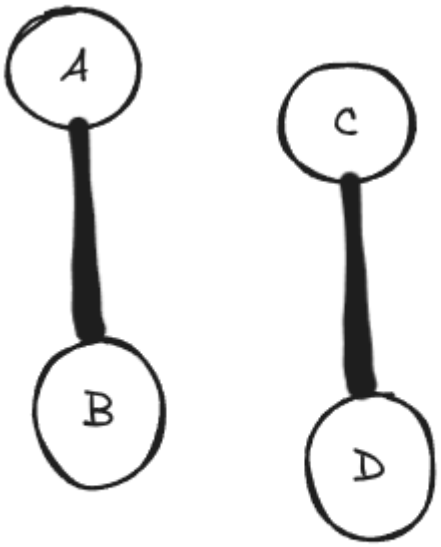
A path in a graph is a finite or infinite set of edges which joins a set of vertices. It can connect to 2 or more nodes. Also, if the path connects all the nodes of a graph in Data Structure, then it is a connected graph, otherwise it is called a disconnected graph.

There may or may not be path to each and every node of graph. In case, there is no path to any node, then that node becomes an isolated node.



In this graph, the path from **'a'** to **'e'** is = **{a,b,c,d,e}**

Types Of Graphs

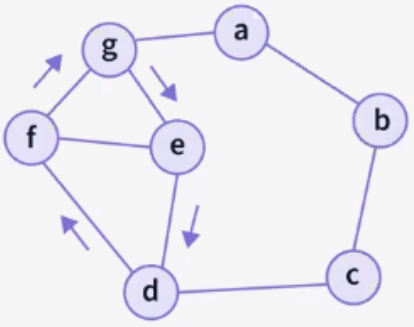


Disconnected Graph

Graph Terminology

Closed Path

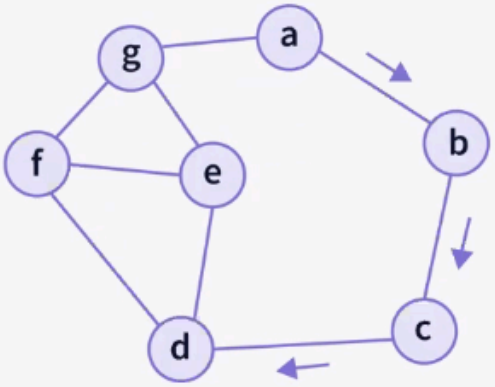
A path is called as closed path if the initial node is same as terminal(end) node. A path will be closed path if : $V_0 = V_n$, where V_0 is the starting node if the graph and V_n is the last node. So, the starting and the terminal nodes are same in a closed graph.



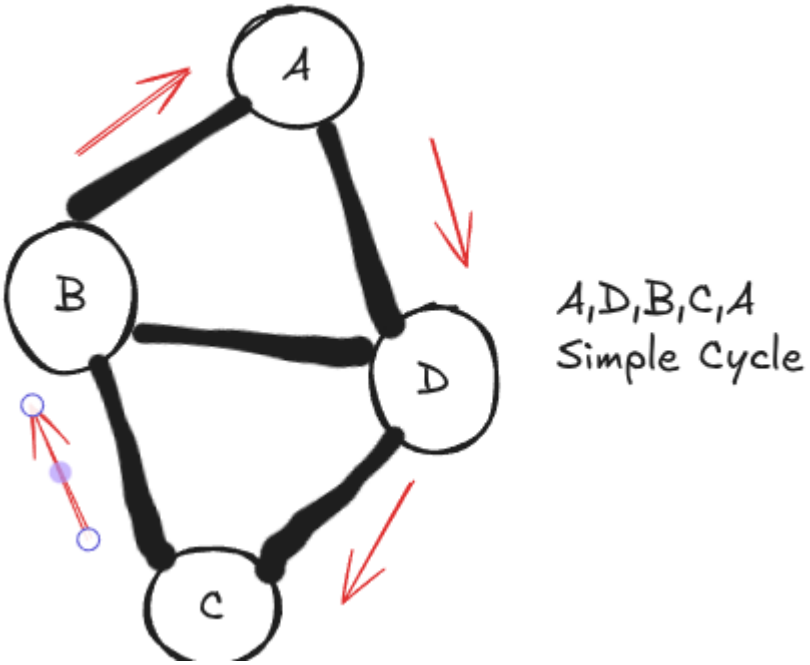
The this graph have a closed path, where the **initial node = {e}** is same as the **final node = {e}**. So, the path becomes = **{e,d,f,g,e}**.

Simple Path

A path that does not repeat any nodes(vertices) is called a simple path. A simple path in a graph exists if all the nodes of the graph are distinct, expect for the first and the last vertex.

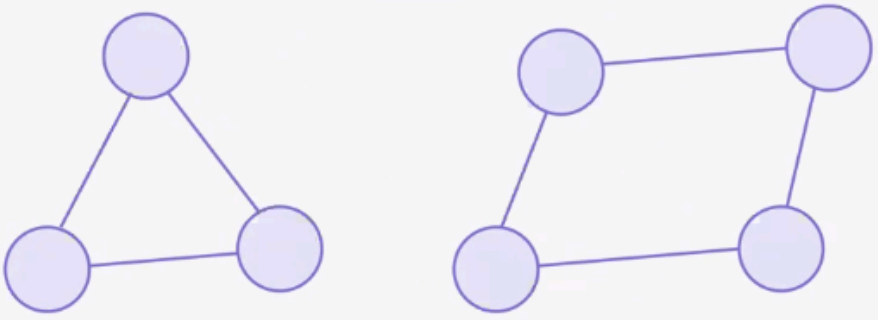


In this graph, **a,b,c,d** is a simple path. Because, this graph do not have any loop or cycle and none of the paths point to themselves.



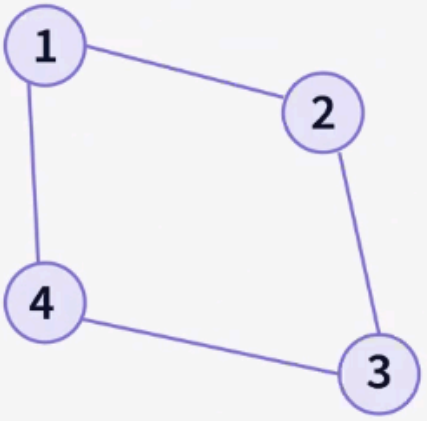
Cycle Graph

A path that does not repeat any nodes(vertices) is called a simple path. A simple path in a graph exists if all the nodes of the graph are distinct, expect for the first and the last vertex.



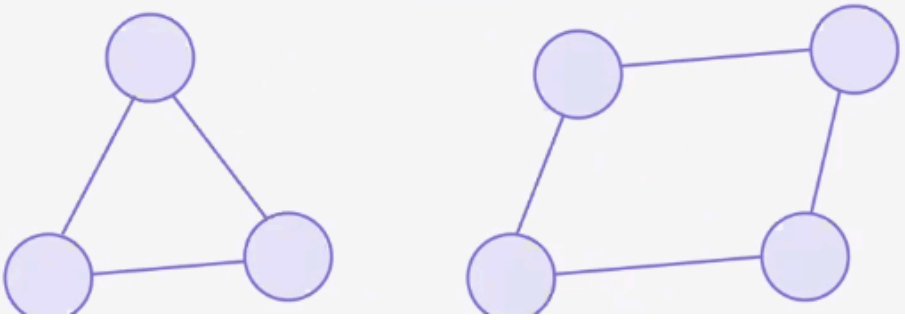
Connected Graph

Connected graph is a graph in which there is an edge or path joining each pair of vertices. So, in a connected graph, it is possible to get from one vertex to any other vertex in the graph through a series of edges.



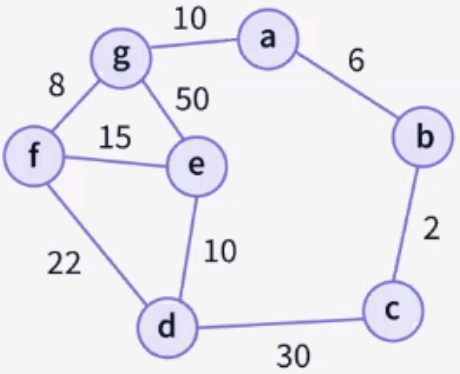
Complete Graph

In a complete graph, there is an edge between every single pair of node in the graph. Here, every vertex has an edge to all other vertices. It is also known as a full graph.



Weighted Graph

In weighted graphs, each edge has a value associated with them (called weight). It refers to a simple graph that has weighted edges. The weights are usually used to compute the shortest path in the graph.

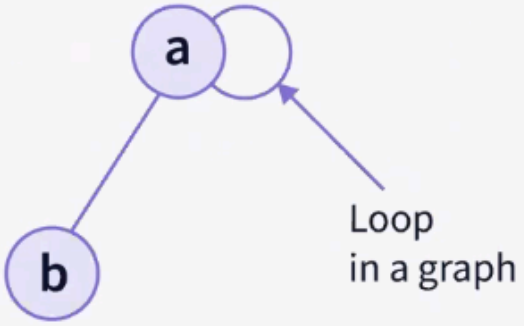


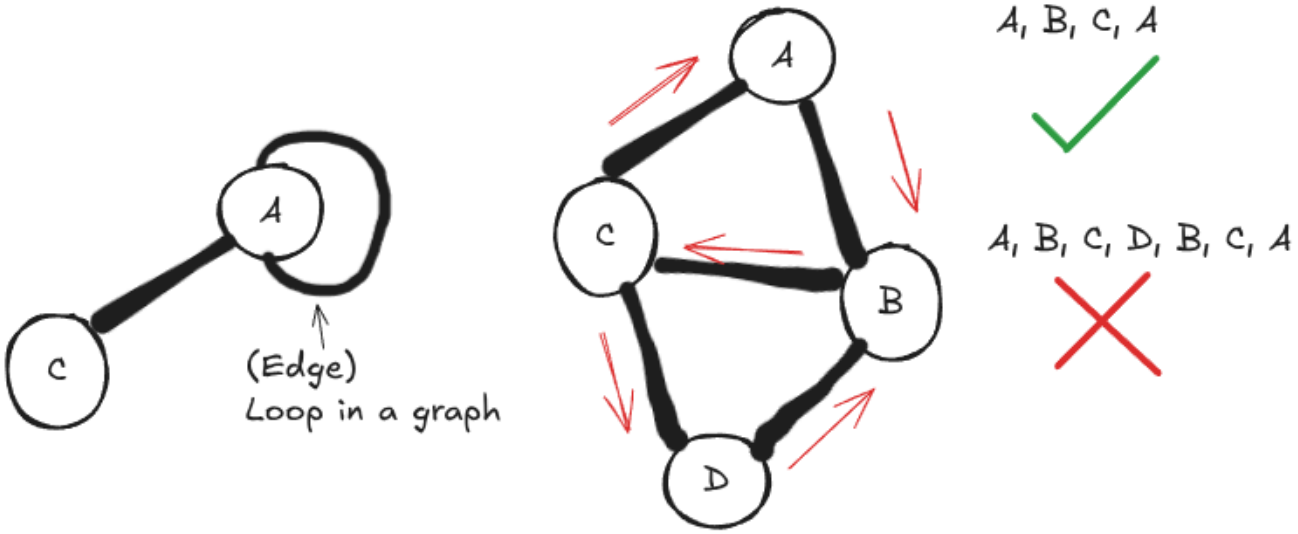
The above graph is a weighted graph, where each edge is associated with a weight. It is not mandatory in a weighted graph that all nodes have distinct weight. Some edges may have same weights.

Loop

A loop (also called a self-loop) is an edge that connects a vertex to itself. It is commonly defined as an edge with both ends as the same vertex.

Although all loops are cycles, not all cycles are loops. Because, cycles do not repeat edges or vertices except for the starting and ending vertex.



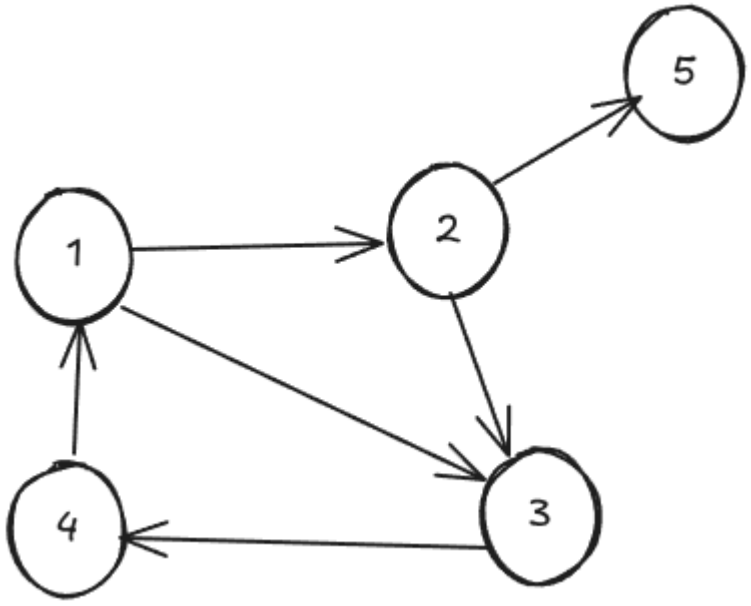


16 - Day

Graphs are classified based on the characteristics of their edges.There are two types of graphs.

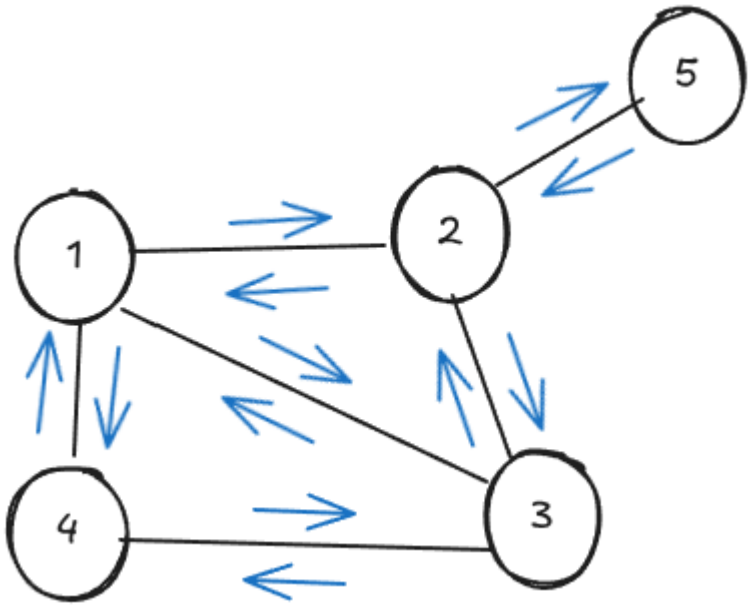
Directed Graphs

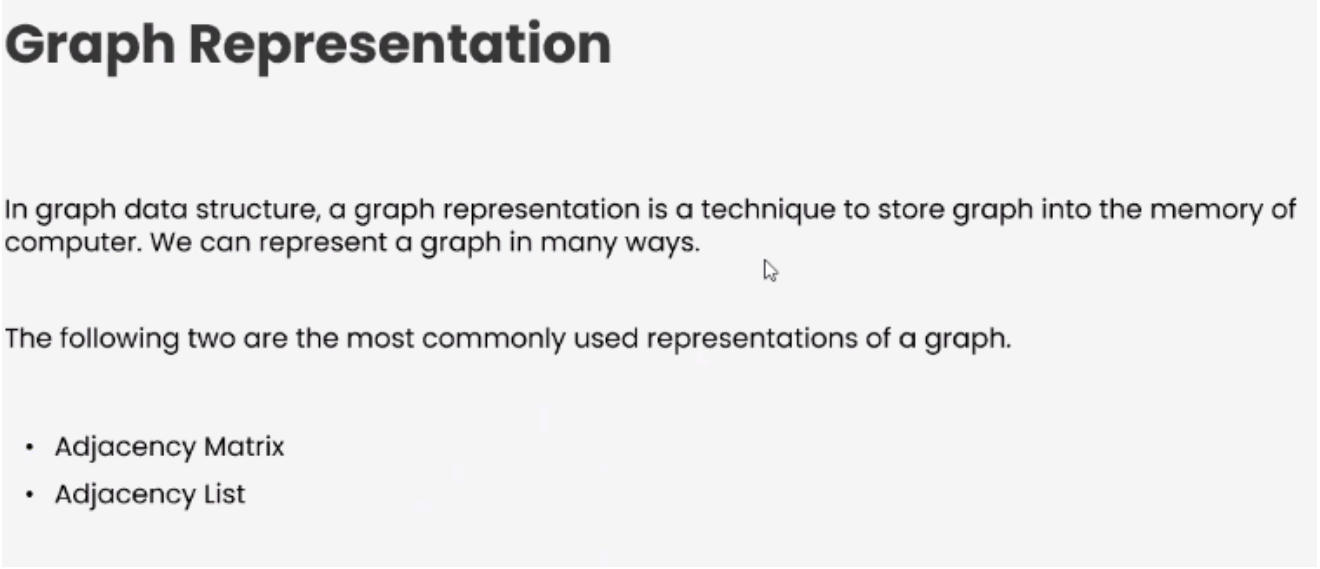
Directed graphs in graph data structure are the graphs where the edges have directions from one node towards the other node.In Directed Graphs, we can only traverse from one node to another if the edge have a direction pointing to that node.



Undirected Graphs

Undirected graphs have edges that do not have a direction. Hence,the graph can be traversed in either direction.





Graph Representation

In graph data structure, a graph representation is a technique to store graph into the memory of computer. We can represent a graph in many ways.

The following two are the most commonly used representations of a graph.

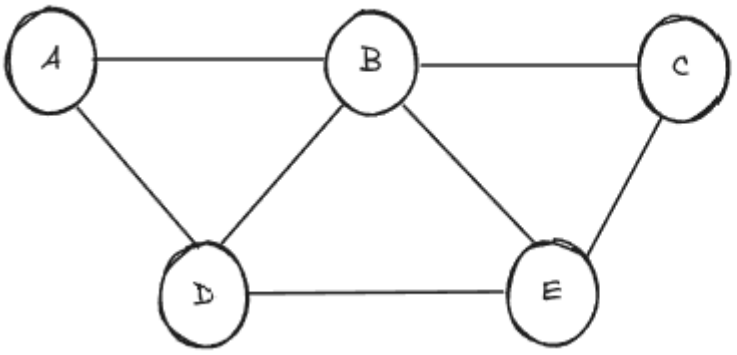
- Adjacency Matrix.
- Adjacency List.

17 - Day

Adjacency Matrix

An adjacency Matric is a 2D array of size V x V where V is the number of nodes in a graph.It is used to represent a `finite graph` ,with 0's and 1's. Since it's size is V x V, it is a squire matrix. The elements of the matrix indicates whether pairs of vertices are adjacent or not in the graph.

Example



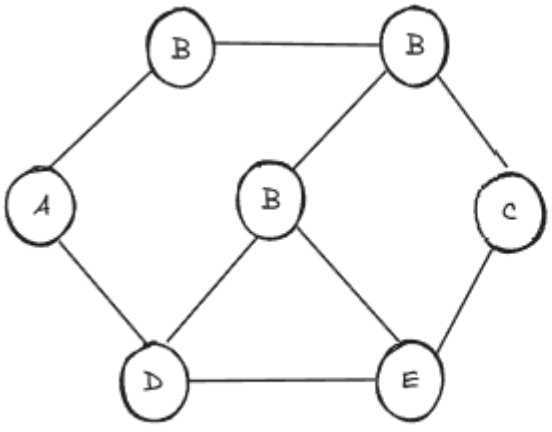
	A	B	C	D	E
A	0	1	0	1	0
B	1	0	1	1	1
C	0	1	0	0	1
D	1	1	0	0	1
E	0	1	1	1	0

Code example

```java
import java.util.ArrayList;
import java.util.Collection;
import java.util.Collections;

public class Graph {  1 usage
    static ArrayList<ArrayList<Integer>> createGraph(int V,int[][] edges){  1 usage

        ArrayList<ArrayList<Integer>> mat = new ArrayList<>();

//        Initialize the matrix with 0
        for(int i = 0;i < V;i++){
            ArrayList<Integer> row = new ArrayList<>(Collections.nCopies(V, o: 0));
            mat.add(row);
        }

//        Add each edge to the adjacency matrix
        for(int[] it : edges){
            int u = it[0];
            int v = it[1];
            mat.get(u).set(v,1);

//              since the graph is undirected
            mat.get(v).set(u,1);
        }
        return mat;
    }
}
```

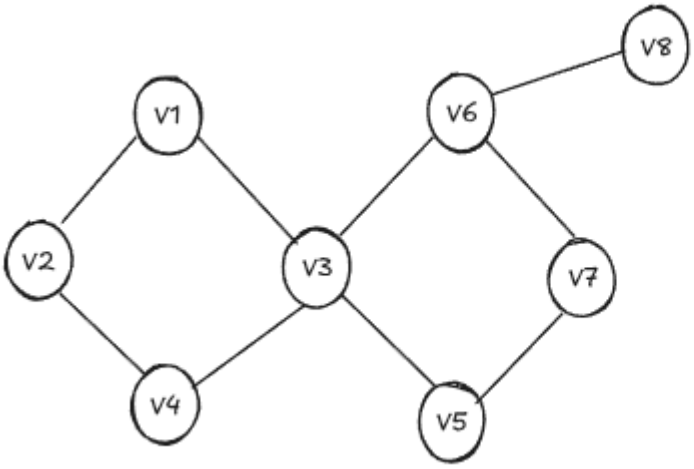
```java
public class Main {
    public static void main(String[] args) {
        int V = 5;

//        List of edges (u, v)
//        A = 0, B = 1, C = 2, D = 3, E = 4
        int[][] edges = {
                {0,1},
                {0,3},
                {1,3},
                {1,2},
                {1,4},
                {2,4},
                {3,4},
        };

//        Build the graph using edges
        ArrayList<ArrayList<Integer>> mat = Graph.createGraph(V, edges);

        System.out.println("Adjacency Matrix Representation");
        for (int i = 0; i < V ; i++){
            for (int j = 0; j < V; j++)
                System.out.print(mat.get(i).get(j) + " ");
            System.out.println();
        }
    }
}
```

Example



	1	2	3	4	5	6	7
1	0	1	0	0	0	1	0
2	1	0	1	0	0	0	0
3	0	1	0	1	1	0	0
4	0	0	1	0	0	1	1
5	0	0	1	0	0	0	1
6	1	0	0	1	0	0	1
7	0	0	0	1	1	1	0

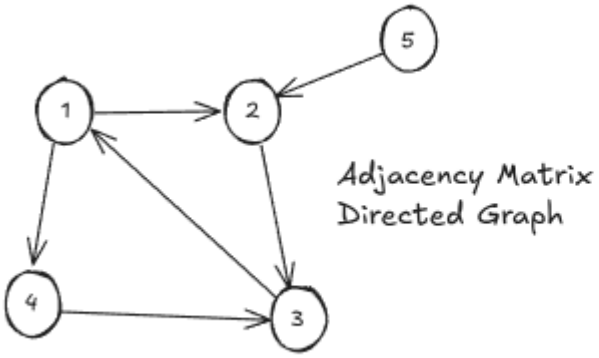
Example



	V1	V2	V3	V4	V5	V6	V7	V8
V1	0	1	1	0	0	0	0	0
V2	1	0	0	1	0	0	0	0
V3	1	0	0	1	1	1	0	0
V4	0	1	1	0	0	0	0	0
V5	0	0	1	0	0	0	1	0
V6	0	0	1	0	0	0	1	1
V7	0	0	0	0	1	1	0	0
V8	0	0	0	0	0	1	0	0

Adjacency Matrix

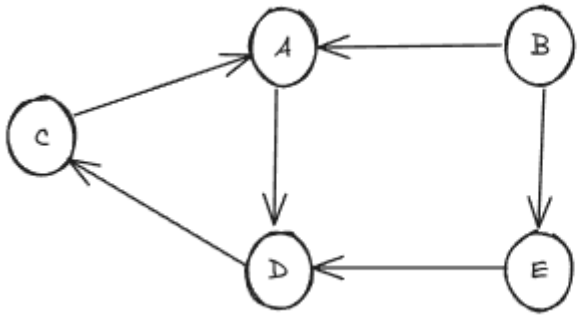
The adjacency Matrix for a directed graph also follows the same conventions, expect for, there is a '1' in the matrix if there is an edge pointing from one node to another, say from node A to node B.But vice versa may not be applicable.



Nodes →

Nodes ↓	1	2	3	4	5
1	0	1	0	1	0
2	0	0	1	0	0
3	1	0	0	0	0
4	0	0	1	0	0
5	0	1	0	0	0

Example



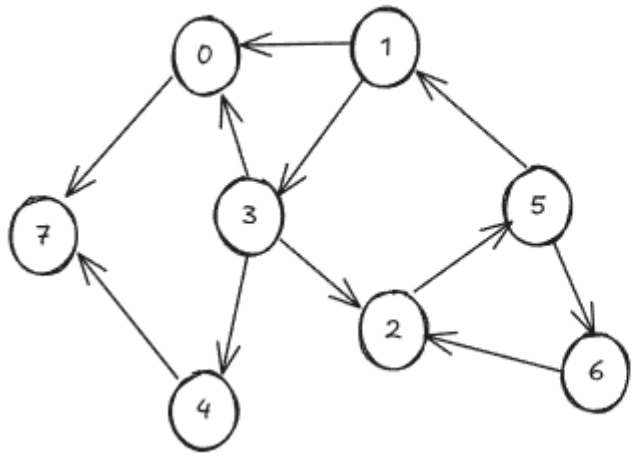
	A	B	C	D	E
A	0	0	0	0	0
B	1	0	0	0	1
C	1	0	0	0	0
D	0	0	1	0	0
E	0	0	0	1	0

Code Example :

```java
public class DirectedGraph {  no usages
    static ArrayList<ArrayList<Integer>> createGraph(int V, int[][] edges) {  no usages
        ArrayList<ArrayList<Integer>> mat = new ArrayList<>();

        // Initialize the matrix with 0
        for (int i = 0; i < V; i++) {
            ArrayList<Integer> row = new ArrayList<>(Collections.nCopies(V,  o: 0));
            mat.add(row);
        }

        // Add each edge to the adjacency matrix
        for (int[] it : edges) {
            int u = it[0];
            int v = it[1];
            mat.get(u).set(v, 1);
        }
        return mat;
    }
}
```

```java
public class Main2 {
    public static void main(String[] args) {
    int V = 5;

//        List of edges (u, v)
//        A = 0, B = 1, C = 2, D = 3, E = 4
    int[][] edges = {
            {2,0},
            {1,0},
            {1,4},
            {4,3},
            {3,2},
    };

//        Build the graph using edges
    ArrayList<ArrayList<Integer>> mat = DirectedGraph.createGraph(V, edges);

    System.out.println("Adjacency Matrix (Directed graph) Representation");
    for (int i = 0; i < V ; i++){
        for (int j = 0; j < V; j++)
            System.out.print(mat.get(i).get(j) + " ");
        System.out.println();
    }
}
}
```

Example

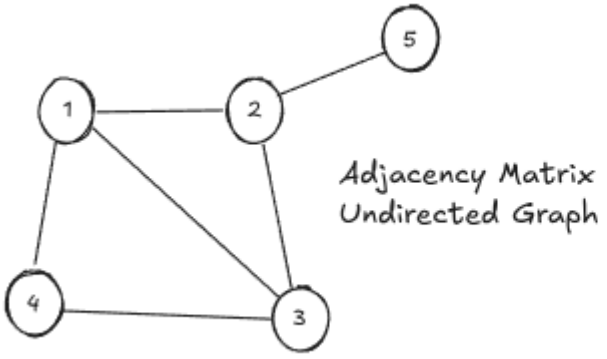


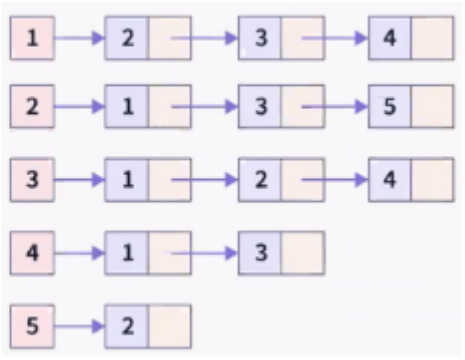
	0	1	2	3	4	5	6	7
0	0	0	0	0	0	0	0	1
1	1	0	0	1	0	0	0	0
2	0	0	0	0	0	1	0	0
3	1	0	1	0	1	0	0	0
4	0	0	0	0	0	0	0	1
5	0	1	0	0	0	0	1	0
6	0	0	1	0	0	0	0	0
7	0	0	0	0	0	0	0	0

Adjacency List

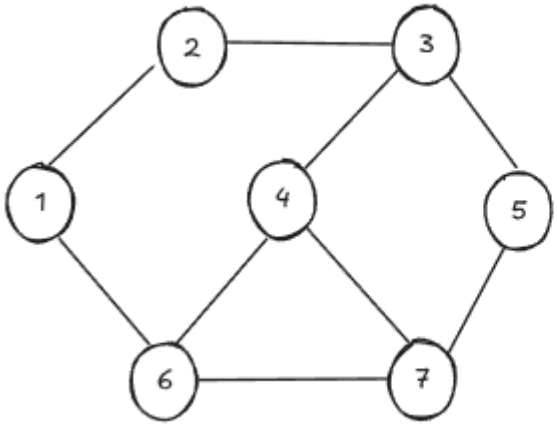
- An adjacency list represents a graph as an array of linked lists.
- The index of the array represents a node.
  - Each element in the linked list represents the nodes that are connected to that node by an edge.

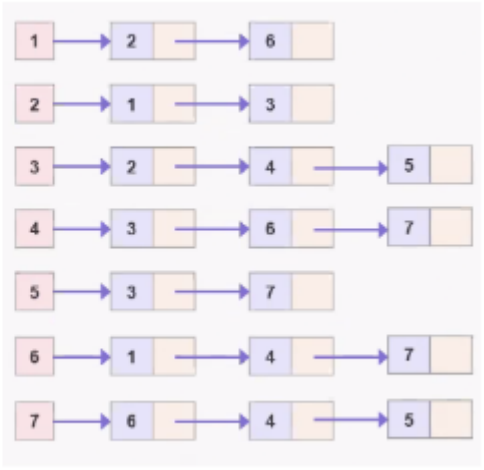
Example





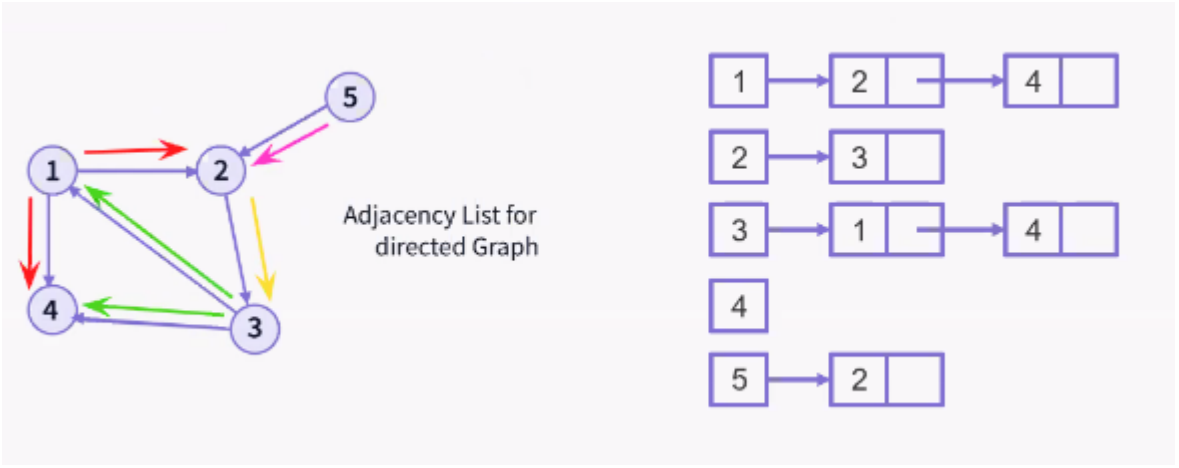
Example



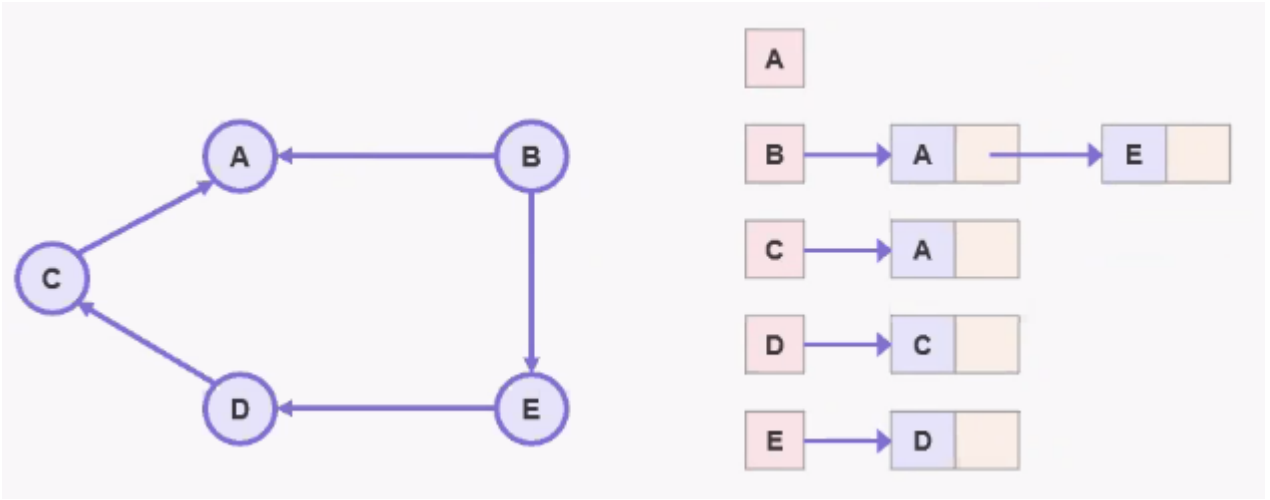


Adjacency List (Directed Graph)

- We us an array of lists (or vector of lists) to represent the graph.
- The size of the array is equal to the number of vertices (here,3).
- Each index in the array represents a vertex.

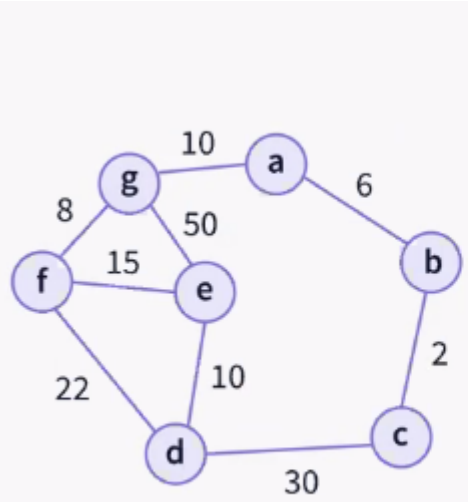


Example



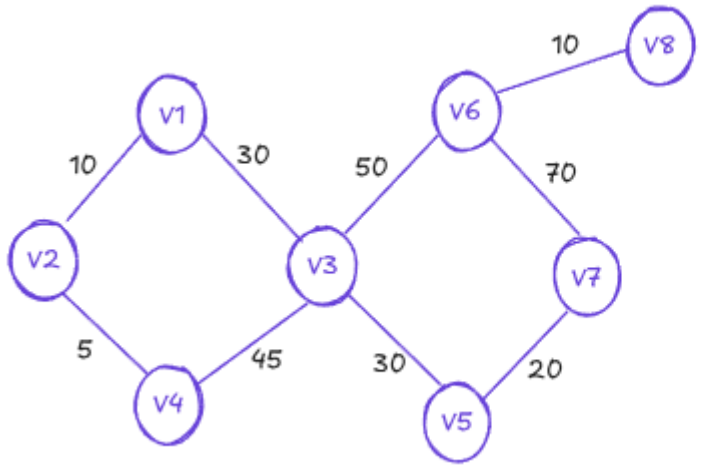
Adjacency Matrix

Weighted Graph



	a	b	c	d	e	f	g
a	0	6	0	0	0	0	10
b	6	0	2	0	0	0	0
c	0	2	0	30	0	0	0
d	0	0	30	0	10	22	0
e	0	0	0	10	0	15	50
f	0	0	0	22	15	0	8
g	10	0	0	0	50	8	0

Example



	V1	V2	V3	V4	V5	V6	V7	V8
V1	0	10	30	0	0	0	0	0
V2	10	0	0	5	0	0	0	0
V3	30	0	0	45	30	50	0	0
V4	0	5	45	0	0	0	0	0
V5	0	0	30	0	0	0	20	0
V6	0	0	50	0	0	0	70	10
V7	0	0	0	0	20	70	0	0
V8	0	0	0	0	0	10	0	0

18 - Day

Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once.

The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

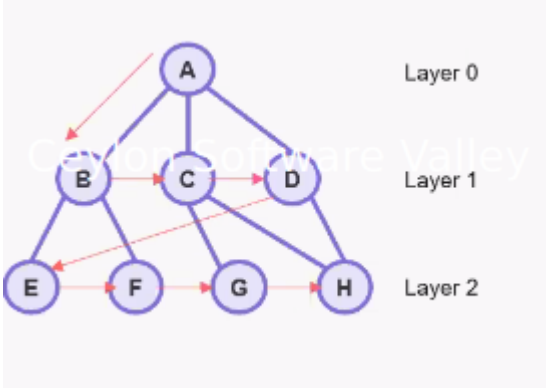
Breadth First Search (BFS)

There are many ways to traverse graphs. BFS is the most commonly used approach.

BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.

As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1. First move horizontally and visit all the nodes of the current layer.
2. Move to the next layer.



Solution :

```
public class Breadth_first {  2 usages

    private Map<Character, List<Character>> adjList;  6 usages

    public Breadth_first() {  1 usage
        adjList = new HashMap<>();

        // Build the graph (based on the image)
        adjList.put('A', Arrays.asList('B', 'C', 'D'));
        adjList.put('B', Arrays.asList('E', 'F'));
        adjList.put('C', Arrays.asList('G', 'H'));
        adjList.put('D', Arrays.asList('H'));
    }

    public void bfs(char start) {  1 usage
        Queue<Character> queue = new LinkedList<>();
        Set<Character> visited = new HashSet<>();

        queue.add(start);
        visited.add(start);

        System.out.print("BFS Traversal: ");

        while (!queue.isEmpty()) {
            char current = queue.poll();
            System.out.print(current + " ");

            for (char neighbor : adjList.getOrDefault(current, new ArrayList<>())) {
                if (!visited.contains(neighbor)) {
```

19 - Day

Depth First Search (DFS)

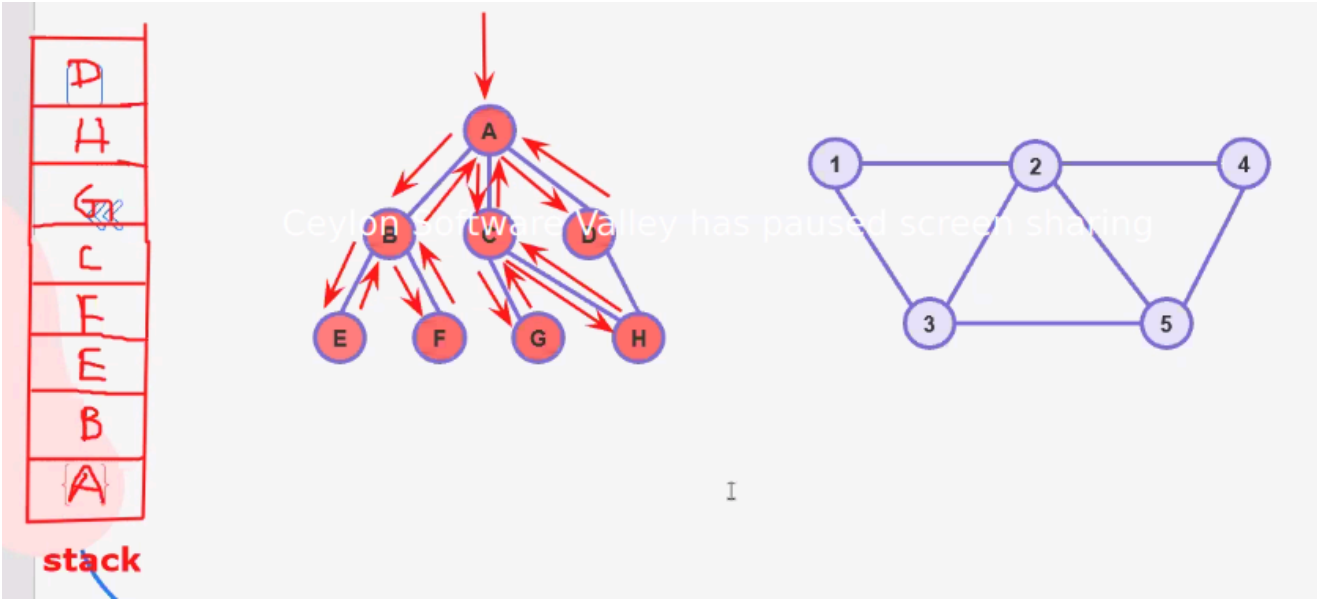
The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

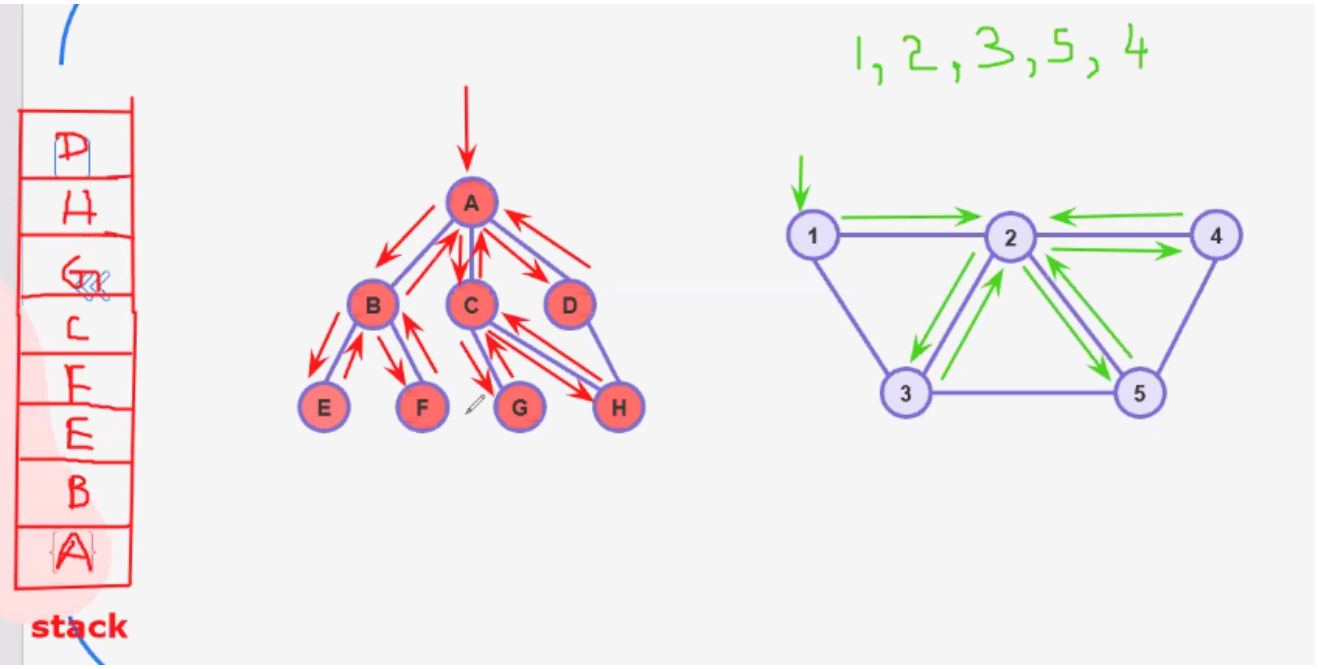
The word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:

- Pick a starting node and push all its adjacent nodes into a stack.
- Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.

Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.





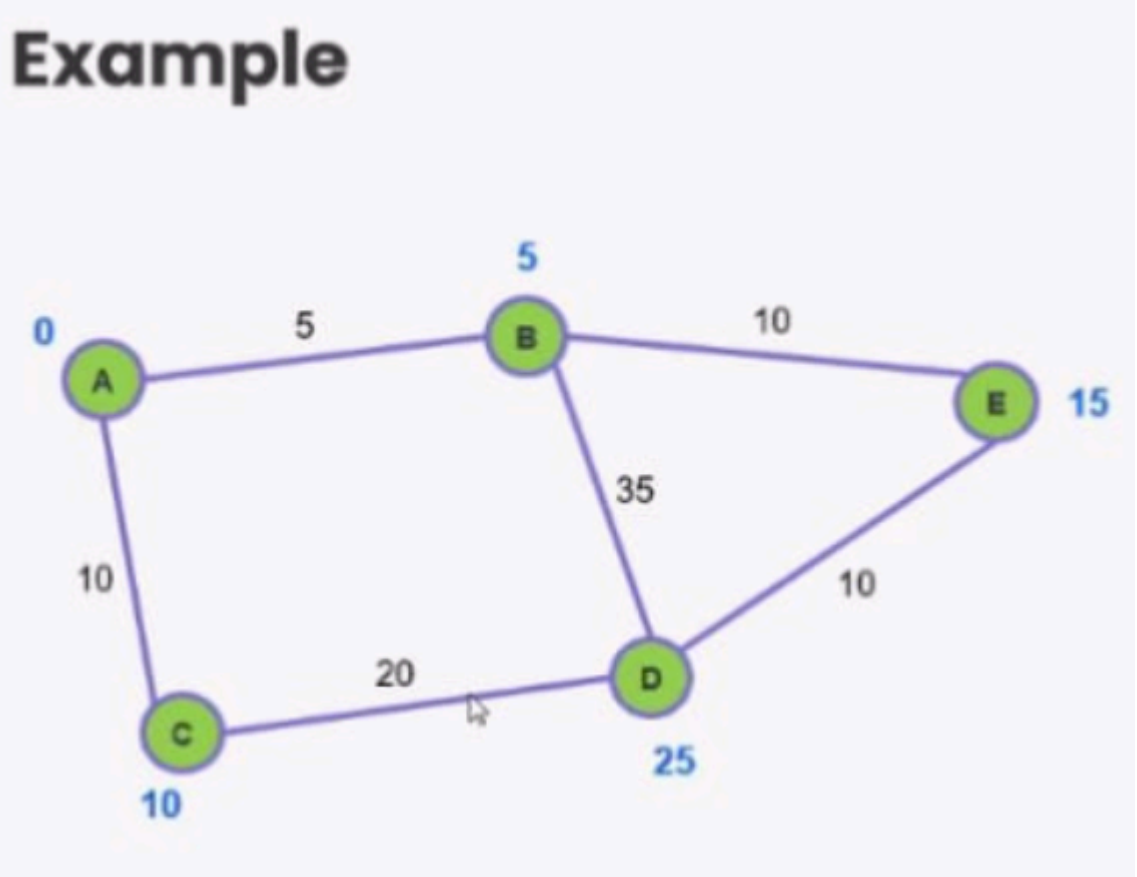
20 - Day

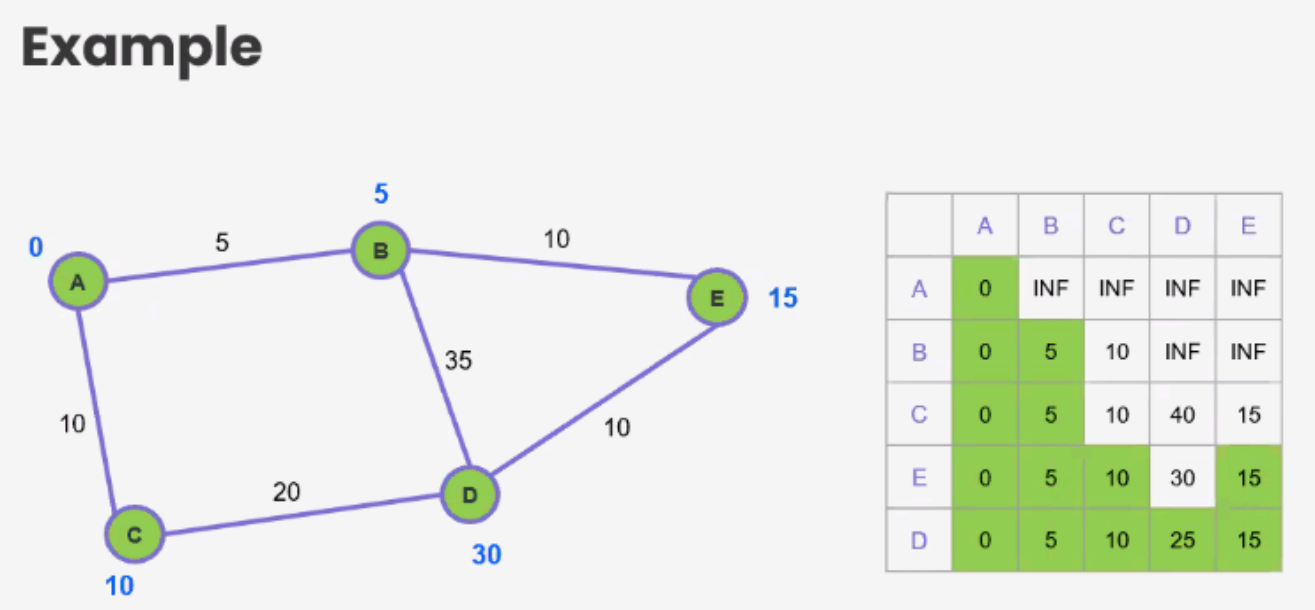
Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from one vertex to all other vertices.
It does so by repeatedly selecting the nearest unvisited vertex and calculating the distance to all the unvisited neighboring vertices.
Dijkstra's algorithm is often considered to be the most straightforward algorithm for solving the shortest path problem.
Dijkstra's algorithm is used for solving single- source shortest path problems for diected or undirected paths. Single - source means that one verte is chosen to be the start, and the algorithm will find the shortest path from that vertex to all other vertices.

Algorithm Works

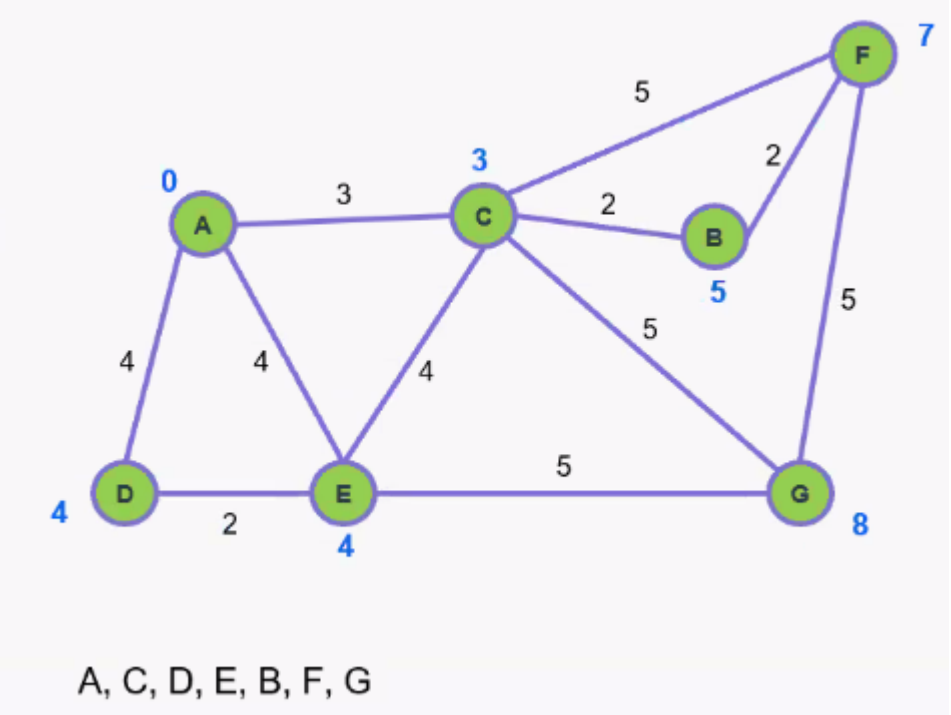
- Set Initial distances for all vertices: 0 for the source vertex, and infinity for all the other.
- Choose the unvisited vertex with the shortest distance from the start to be the current vertex. So the algorithm will always start with the source as the current vertex.
- For each of the current vertex's unvisited neighbor vertices, calculate the distance from the source and update the distance if the new, calculated, distance is lower.
- We are now done with the current vertex, so we mark it s visited. A visited vertex is not checked again.
- Go back to step 2 to choose a new current vertex, and keep repeating these steps until all vertices are visited.
- In the end we are left with the shortest path from the source vertex to every other vertex in the graph.



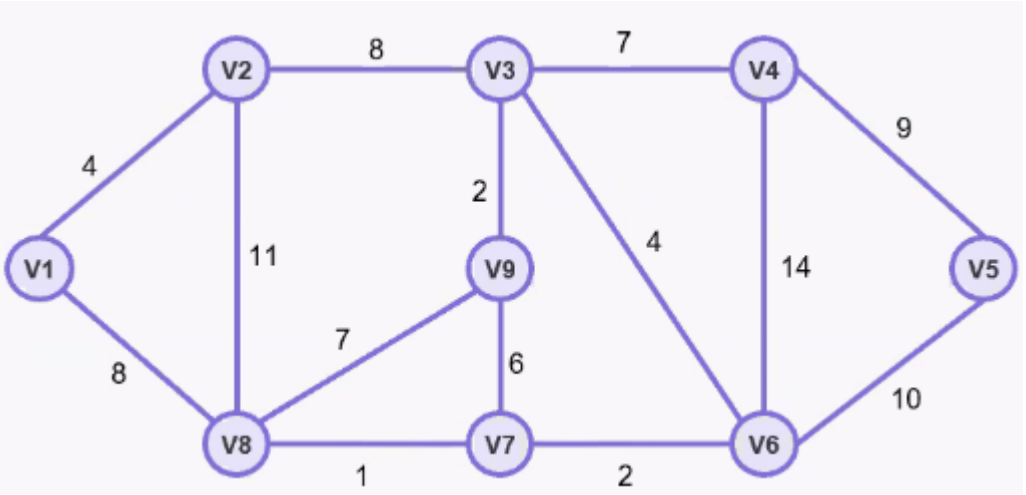


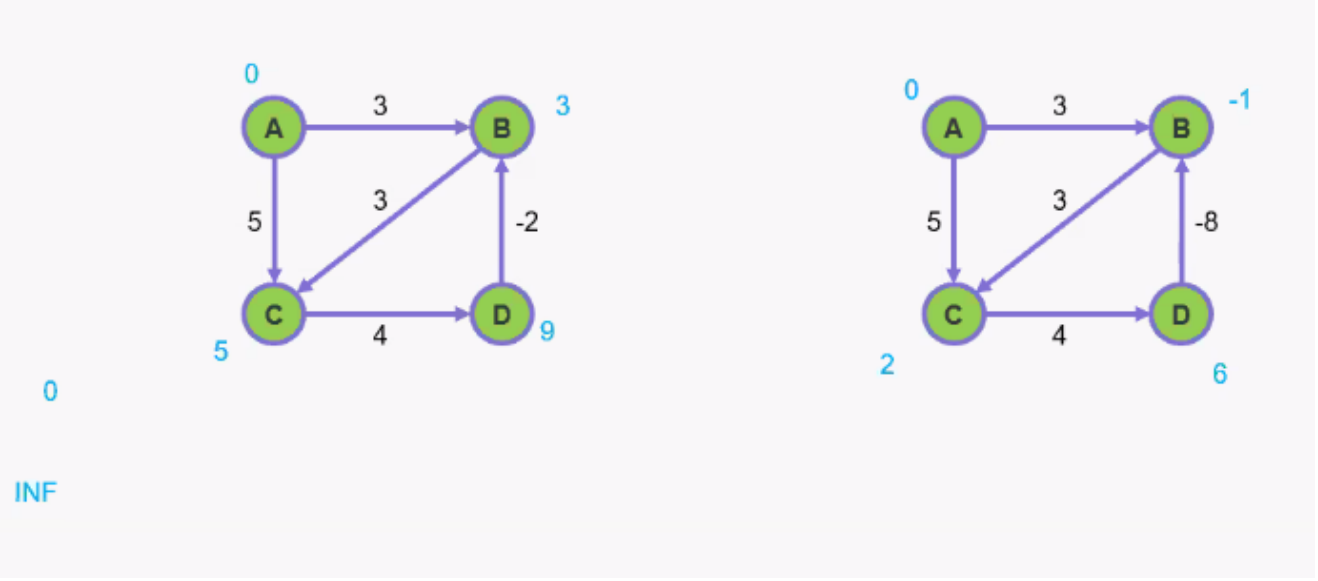
	A	B	C	D	E
A	0	INF	INF	INF	INF
B	0	5	10	INF	INF
C	0	5	10	40	15
E	0	5	10	30	15
D	0	5	10	25	15

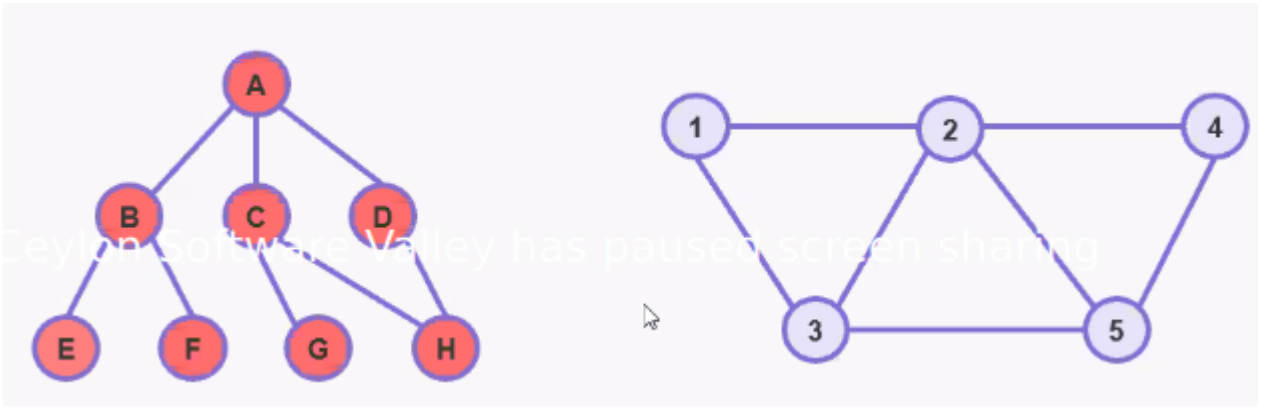
Example

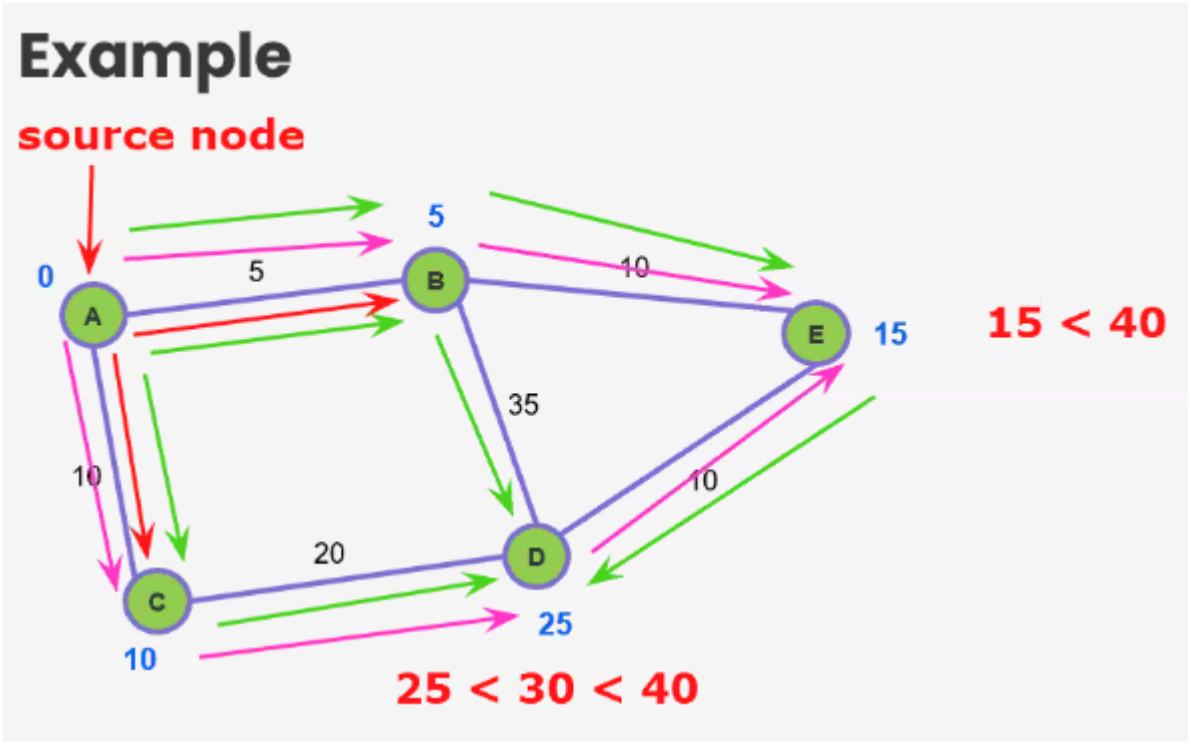

A
B
C
D
E
F
G
0
3
5
7
4
2
4
8
A, C, D, E, B, F, G

Example

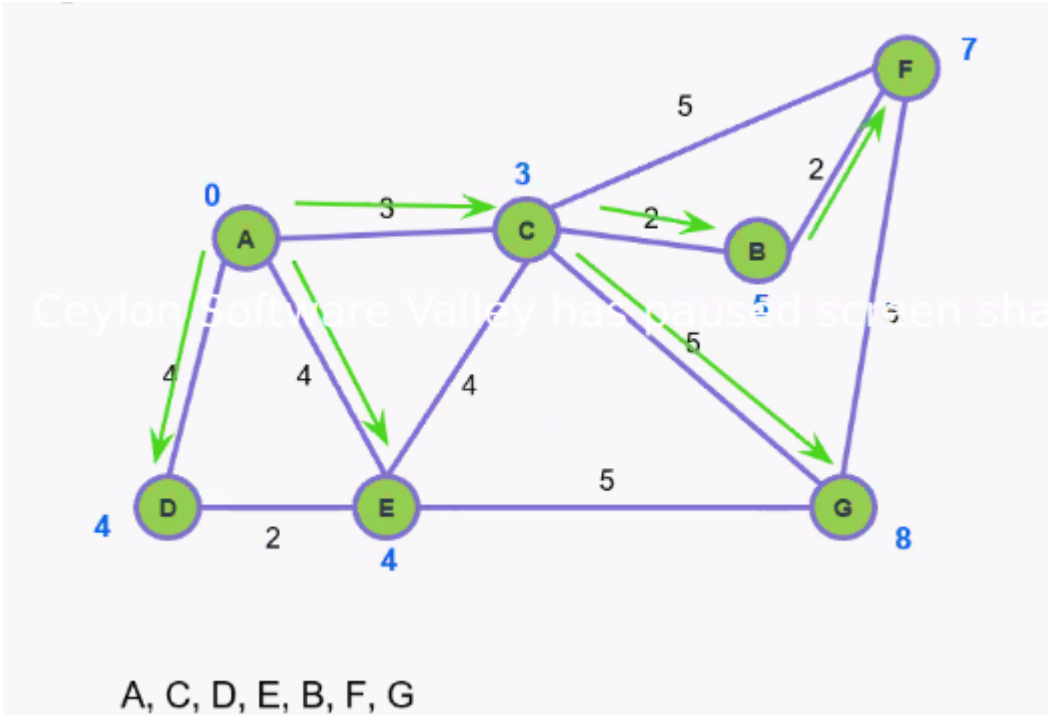

V1
V2
V3
V4
V5
V6
V7
V8
V9
4
8
11
7
2
4
9
14
6
10
1
2


A
B
C
D
0
3
5
-2
4
5
9
INF
A
B
C
D
0
-1
5
3
-8
4
2
6

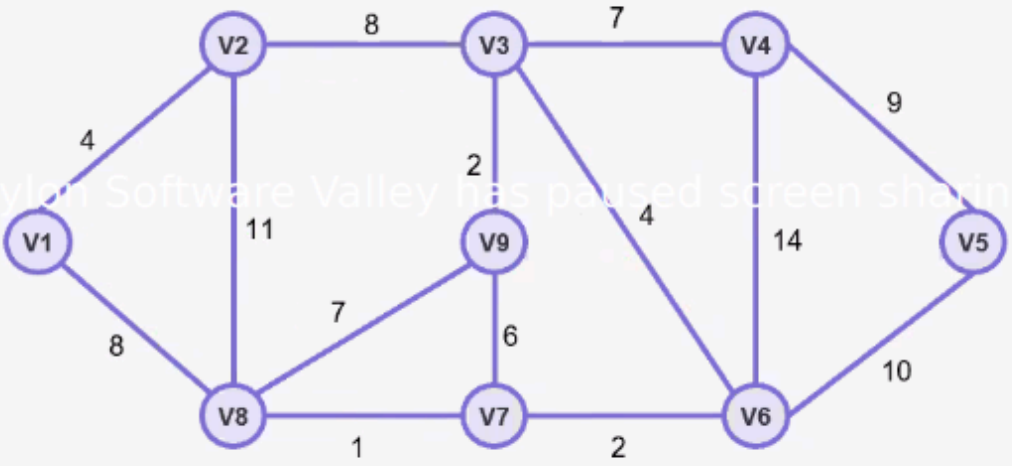

A
B
C
D
E
F
G
H
1
2
3
4
5


Example
source node
0
A
5
B
5
10
E
15
15 < 40
35
10
10
20
D
C
10
25
25 < 30 < 40

Example


0
A
3
C
3
5
F
7
2
2
B
5
5
4
4
4
5
D
E
G
4
2
4
5
8
A, C, D, E, B, F, G

Example



```java
public class Graph {  2 usages
    static final int V = 9; // number of vertices  7 usages

    // find vertex with minimum distance not yet processed
    static int minDistance(int dist[], boolean visited[]) {  1 usage
        int min = Integer.MAX_VALUE, minIndex = -1;
        for (int v = 0; v < V; v++) {
            if (!visited[v] && dist[v] <= min) {
                min = dist[v];
                minIndex = v;
            }
        }
        return minIndex;
    }

    // print shortest distances from source
    static void printSolution(int dist[]) {  1 usage
        System.out.println("Vertex\tDistance from V1");
        for (int i = 0; i < V; i++)
            System.out.println("V" + (i + 1) + "\t\t" + dist[i]);
    }

    // Dijkstra algorithm
    static void dijkstra(int graph[][], int src) {  1 usage
        int dist[] = new int[V];
        boolean visited[] = new boolean[V];
```

```java
public class Main {
    public static void main(String[] args) {
        // adjacency matrix for the given graph (0 = no edge)
        Graph g = new Graph();

        int graph[][] = new int[][] {
                //V1 V2 V3 V4 V5 V6 V7 V8 V9
                {0, 4, 0, 0, 0, 0, 0, 8, 0}, // V1
                {4, 0, 8, 0, 0, 0, 0,11, 0}, // V2
                {0, 8, 0, 7, 0, 4, 0, 0, 2}, // V3
                {0, 0, 7, 0, 9,14, 0, 0, 0}, // V4
                {0, 0, 0, 9, 0,10, 0, 0, 0}, // V5
                {0, 0, 4,14,10, 0, 2, 0, 0}, // V6
                {0, 0, 0, 0, 0, 2, 0, 1, 6}, // V7
                {8,11, 0, 0, 0, 0, 1, 0, 7}, // V8
                {0, 0, 2, 0, 0, 0, 6, 7, 0}  // V9
        };

        g.dijkstra(graph, src: 0); // source is V1 (index 0)
    }
}
```

Bellman Ford's Algorithm

It is similar to Dijkstra's algorithm but it can work with graphs in which edges can have negative weights.

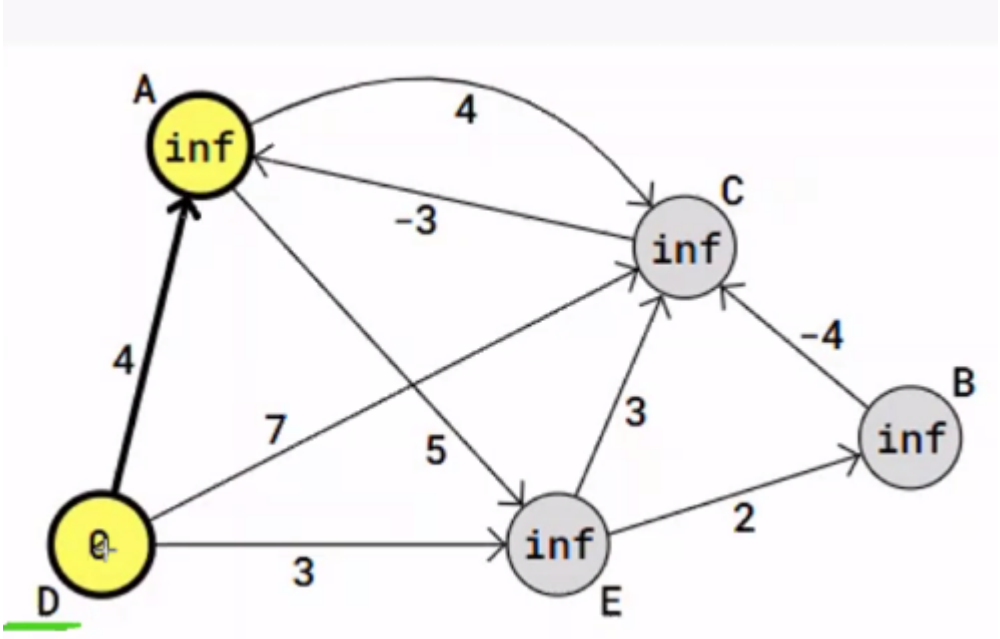
The Bellman-Ford algorithm is best suited to find the shortest paths in a directed graph, with one or more negative edge weights, from the source vertex to all other vertices.

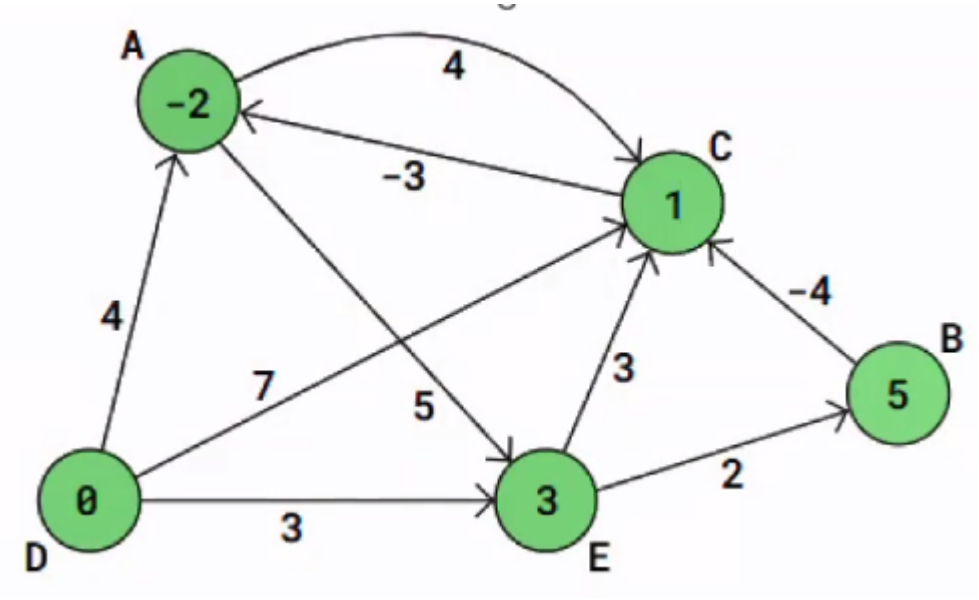
It does so by repeatedly checking all the edges in the graph for shorter paths, as many times as there are vertices in the graph (minus 1).

Relaxation of Edges for Bellman-Ford

It states that for the graph having **N vertices**. all the edges should be relaxed **N-1** times to compute the single source shortest path.

In order to detect whether a negative cycle exists or not, relax all the edge one more time and if the shortest distance for any node reduces then we can say that a negative cycle exists. In short if we relax the edges **N times**, and there is any change in the shortest distance of any node between the **N-11th** and **Nth** relaxation than a negative cycle exists, otherwise not exist.





21 - Day

Huffman Coding

Huffman Coding is a technique that is used for compressing data to reduce its size without losing any of its details. It was first developed by **David Huffman** and was named after him.

Huffman Coding is generally used to compress the data which consists of the frequently repeating characters.

Huffman Coding is a famous **Greedy algorithm**. It is said to be a Greedy Algorithm because the size of code assigned to a character depends on the frequency of the character.

The character with higher frequency gets the short-length variable code and vice-versa for characters with lower frequency. It uses a variable-length encoding which means that it assigns a variable-length code to all the characters in the given stream of data.

Prefix Rule

This rule basically means that the code which is assigned to any character should not be the prefix of any other code.

If this rule is not followed, then some ambiguities can occur during the decoding of the Huffman tree formed.


```
a - 0
b - 1
c - 01
```

Now assume that the generated bitstream is 001, while decoding the code it can be written as follows:

```
0 0 1 = aab
0 01 = ac
```

How does Huffman Coding work?

There are mainly two major steps involved in obtaining the Huffman Code for each unique character:

1. Firstly, build a Huffman Tree from the given stream of data(only taking the unique characters).
2. Secondly, we need to traverse the Huffman Tree which is formed and assign codes to characters and decode the given string using these huffman codes.

A - 4
B - 7
C - 3
D - 2
E - 4

How to Build a Huffman Tree from Input Characters?

Firstly, we need to calculate the frequency of each character in the given string.

ABBCDBCCDAABBEEEBEAB

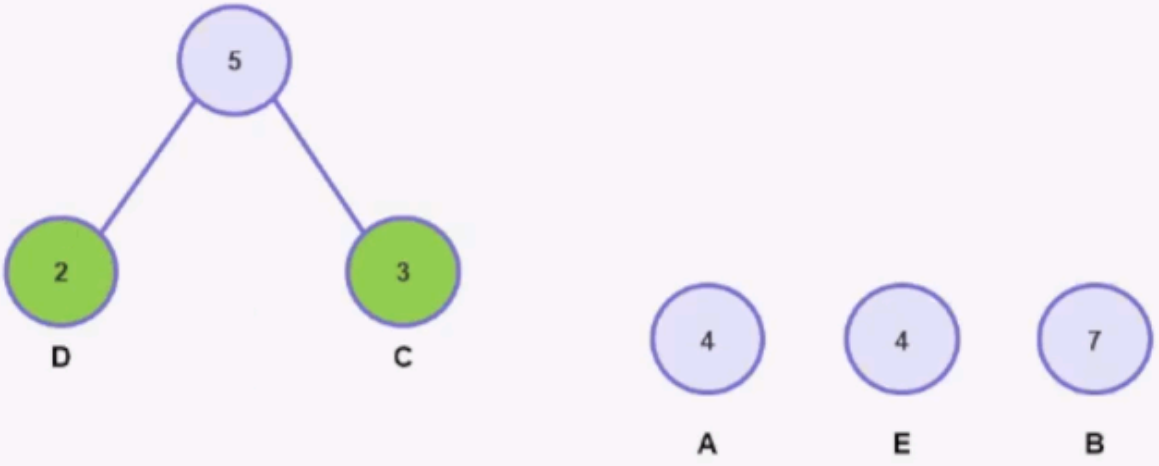
Character	Frequency/count	Binary Code
A	4	
B	7	
C	3	
D	2	
E	4	

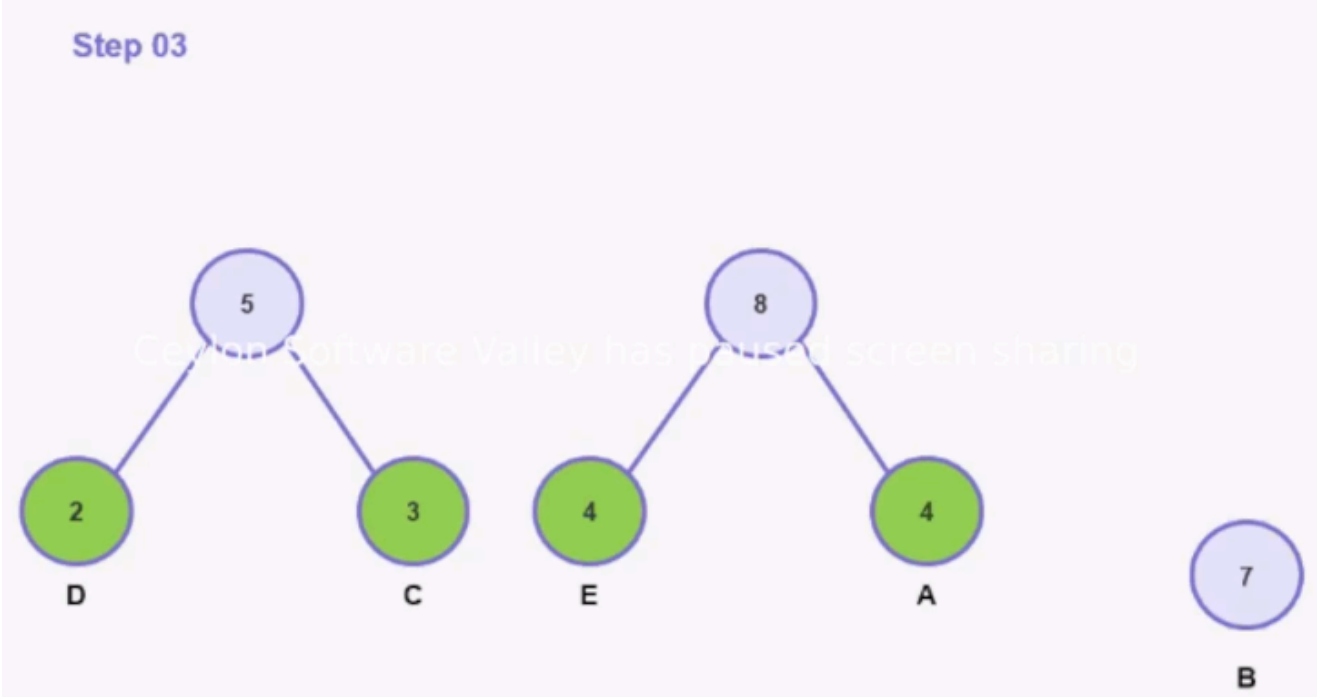
How to Build a Huffman Tree from Input Characters?

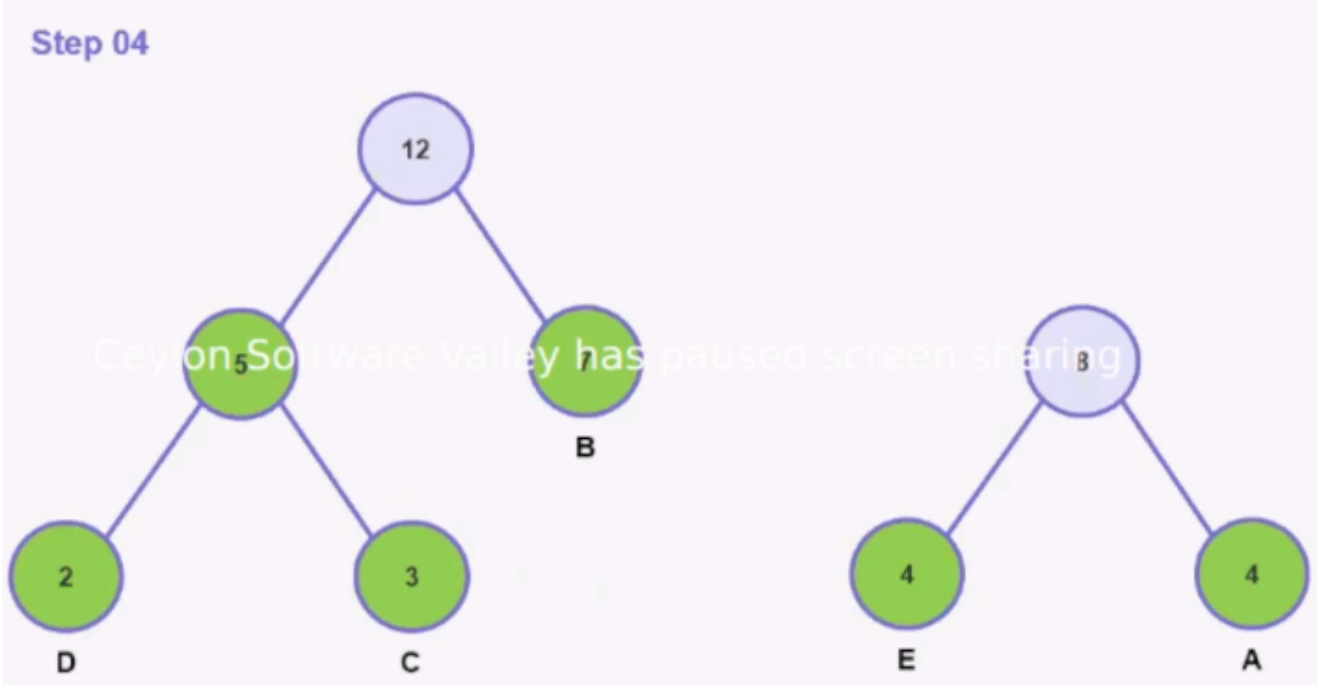
1. Sort the characters in ascending order of frequency. These are stored in a priority queue Q/ min-heap.
2. Create a leaf node for every unique character and its frequency from the given stream of data.
3. Extract the two minimum frequency nodes from the nodes and the sum of these frequencies is made the new root of the tree.
4. While extracting the nodes with the minimum frequency from the min-heap:
   - Make the first extracted node its left child and the other extracted node as its right child.
   - Add this node to the min-heap.
   - Since the minimum frequency should always be on the left side of the root.
5. Repeat steps 3 and 4 until the heap contains only one node(i.e, all characters are included in the tree). The remaining node is the root node and the tree is complete

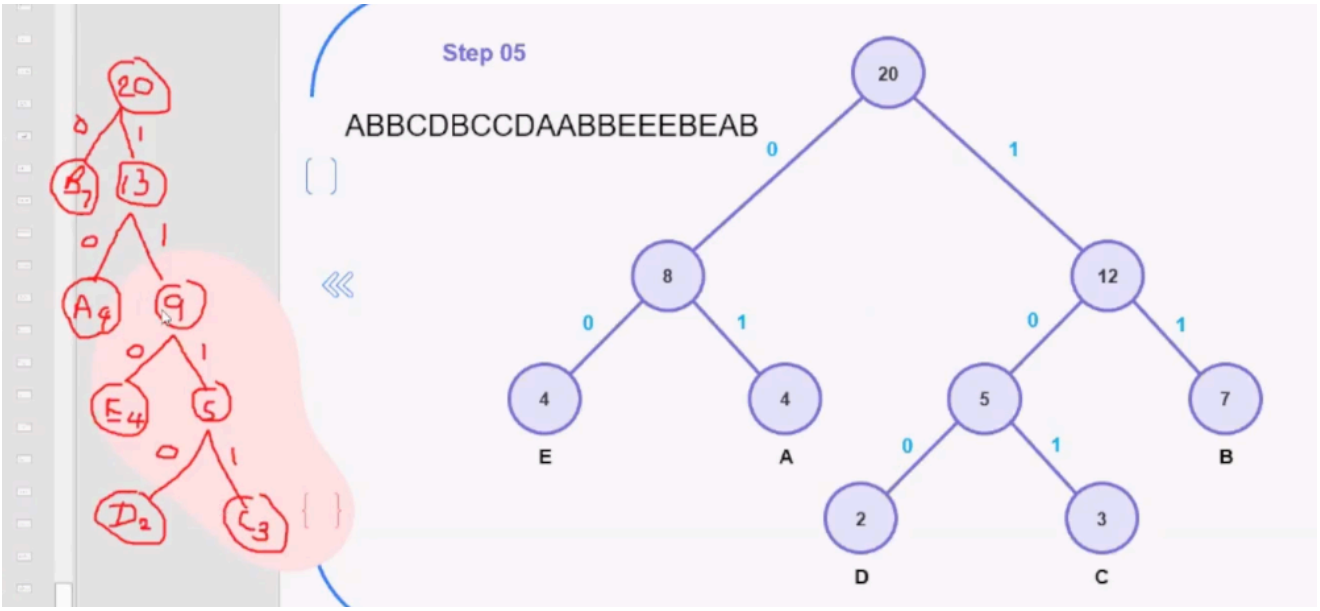
Step 01

Step 02




Step 03
5
2
3
D
C
8
4
4
E
A
7
B


Step 04
12
5
7
B
2
3
D
C
8
4
4
E
A


Step 05
ABBCDBCCDAABBEEEBEAB
20
0
1
8
12
0
1
0
1
4
4
5
7
E
A
0
1
B
2
3
D
C

ABBCDBCCDAABBEEEBEAB -> 160bit (20 bytes)

Character	Frequency/count	Binary Code
A	4	01
B	7	11
C	3	101
D	2	100
E	4	00

01111110110011101101100010111110000001100011 1

2. Remaining Iteration

The same process goes on for the remaining iterations.

After each iteration, the largest element among the unsorted elements is placed at the end.

In each iteration, the comparison takes place up to the last unsorted element.

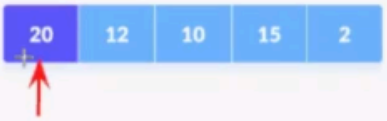
The array is sorted when all the unsorted elements are placed at their correct positions.

Selection Sort Algorithm

Selection sort is a sorting algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

Working of Selection Sort

1. Set the first element as minimum.

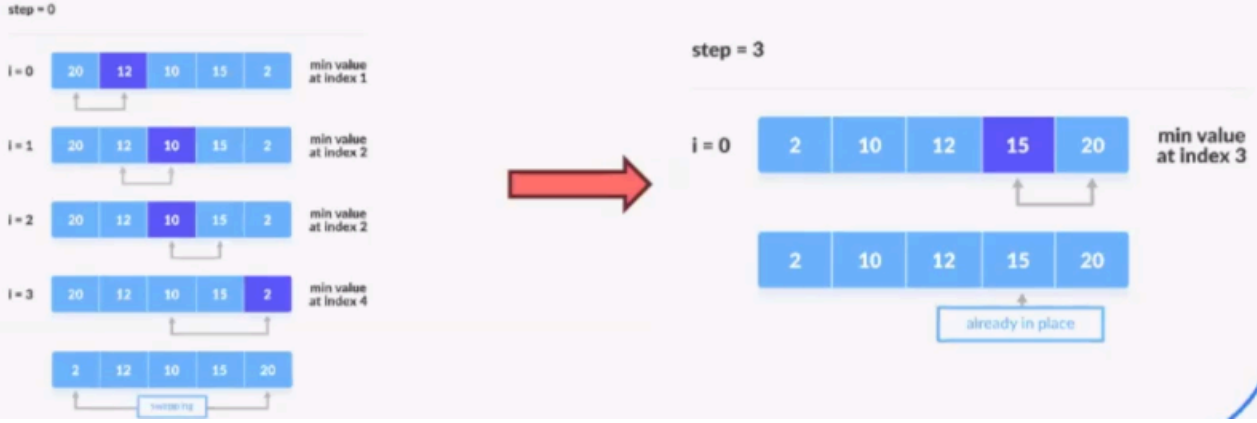


2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.

Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.

3. After each iteration, minimum is placed in the front of the unsorted list.

4. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions



🧠 Why Is It Called “Selection”?

Because in each pass you **select** the smallest value from the remaining unsorted section.

22 - Day

Insertion Sort Algorithm

Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.

Insertion sort works similarly as we sort cards in our hand in a card game.

We assume that the first card is already sorted then, we select an unsorted card. If the unsorted card is greater than the card in hand, it is placed on the right otherwise, to the left. In the same way, other unsorted cards are taken and put in their right place.

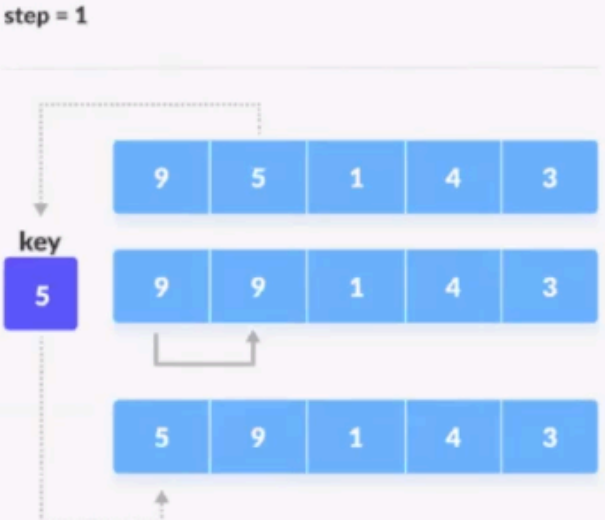
A similar approach is used by insertion sort.

Working of Insertion Sort

Suppose we need to sort the following array.

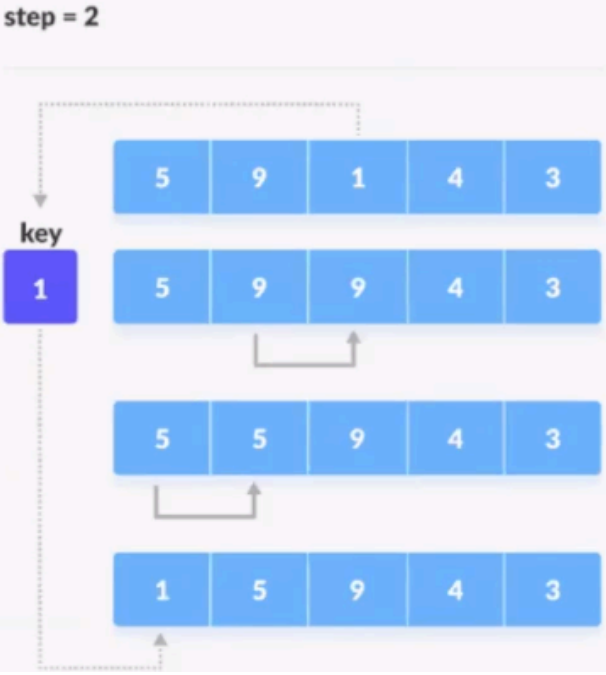
01. The first element in the array is assumed to be sorted. Take the second element and store it separately in key.

Compare key with the first element. If the first element is greater than key, then key is placed in front of the first element.

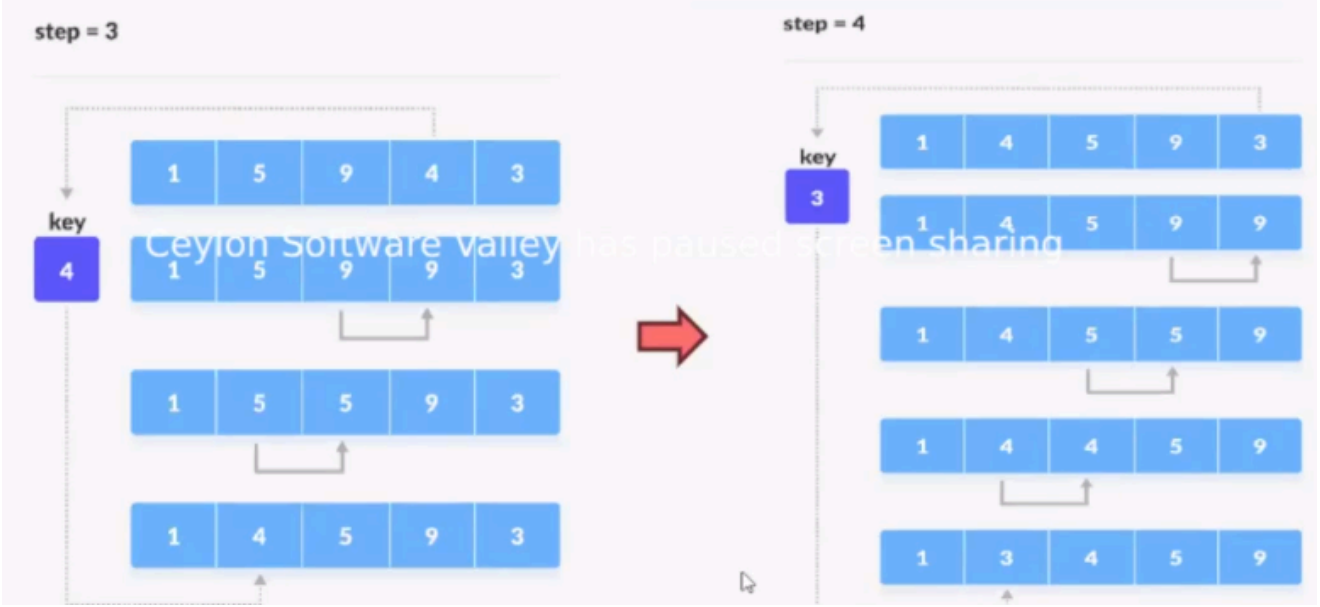


02. Now, the first two elements are sorted.

Take the third element and compare it with the elements on the left of it. Placed it just behind the element smaller than it. If there is no element smaller than it, then place it at the beginning of the array.



03. Similarly, place every unsorted element at its correct position



Merge Sort Algorithm

Merge Sort is one of the most popular sorting algorithms that is based on the **principle of Divide and Conquer Algorithm.**

Here, a problem is divided into multiple sub-problems. Each sub-problem is solved individually. Finally, sub-problems are combined to form the final solution.

Divide and Conquer Strategy

Using the **Divide and Conquer** technique, we divide a problem into subproblems. When the solution to each subproblem is ready, we 'combine' the results from the subproblems to solve the main problem.

Suppose we had to sort an array A. A subproblem would be to sort a sub-section of this array starting at index p and ending at index r, denoted as A[p..r].

Divide

If q is the half-way point between p and r, then we can split the subarray A[p..r] into two arrays A[p..q] and A[q+1, r].

Conquer

In the conquer step, we try to sort both the subarrays A[p..q] and A[q+1, r]. If we haven't yet reached the base case, we again divide both these subarrays and try to sort them.

Combine

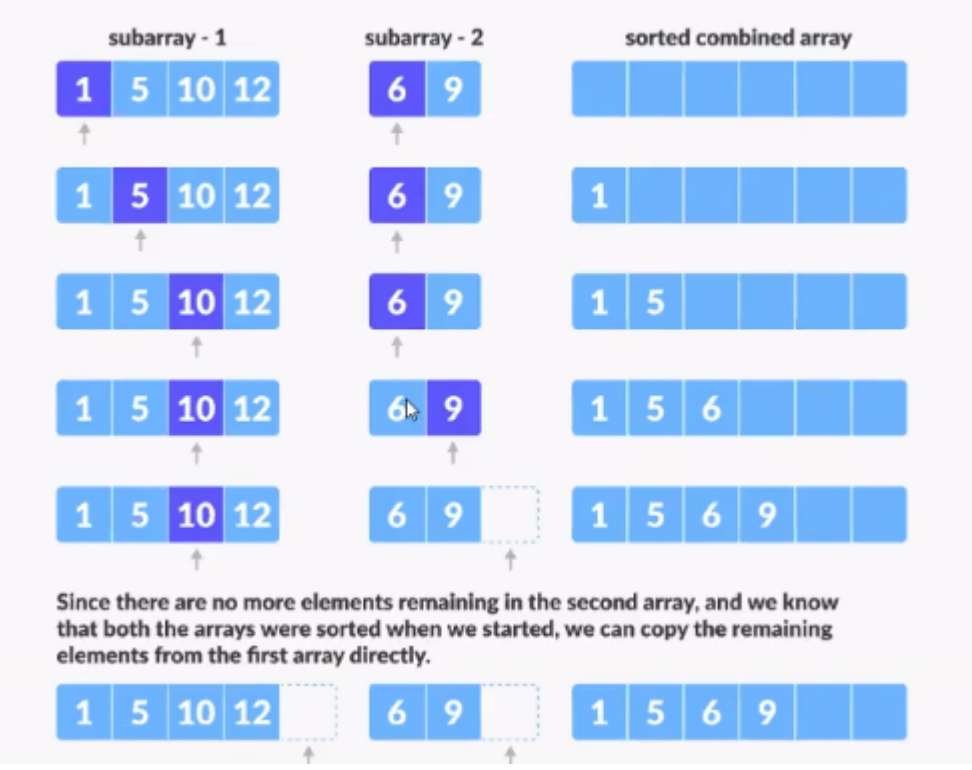
When the conquer step reaches the base step and we get two sorted subarrays A[p..q] and A[q+1, r] for array A[p..r], we combine the results by creating a sorted array A[p..r] from two sorted subarrays A[p..q] and A[q+1, r].

The merge Step of Merge Sort

Every recursive algorithm is dependent on a base case and the ability to combine the results from base cases. Merge sort is no different. The most important part of the merge sort algorithm is, you guessed it, merge step.

The merge step is the solution to the simple problem of merging two sorted lists(arrays) to build one large sorted list(array).

The algorithm maintains three pointers, one for each of the two arrays and one for maintaining the current index of the final sorted array.



The merge function works as follows:

1.Create copies of the subarrays L <- A[p..q] and M <- A[q+1..r].

2.Create three pointers i, j and k

1. i maintains current index of L, starting at 1
2. j maintains current index of M, starting at 1
3. k maintains the current index of A[p..q], starting at p.

3.Until we reach the end of either L or M, pick the larger among the elements from L and M and place them in the correct position at A[p..q]

4.When we run out of elements in either L or M, pick up the remaining elements and put in A[p..q]

23 - Day

Merge( ) Function Explained Step-By-Step

A lot is happening in this function, so let's take an example to see how this would work.

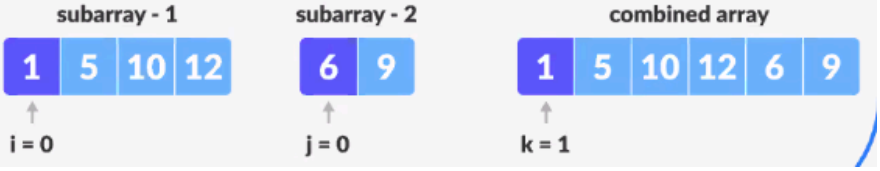
The array contains two sorted subarrays A[0..3] and A[4..5]. Let us see how the merge function will merge the two arrays.

Step 1:

Create duplicate copies of sub-arrays to be sorted

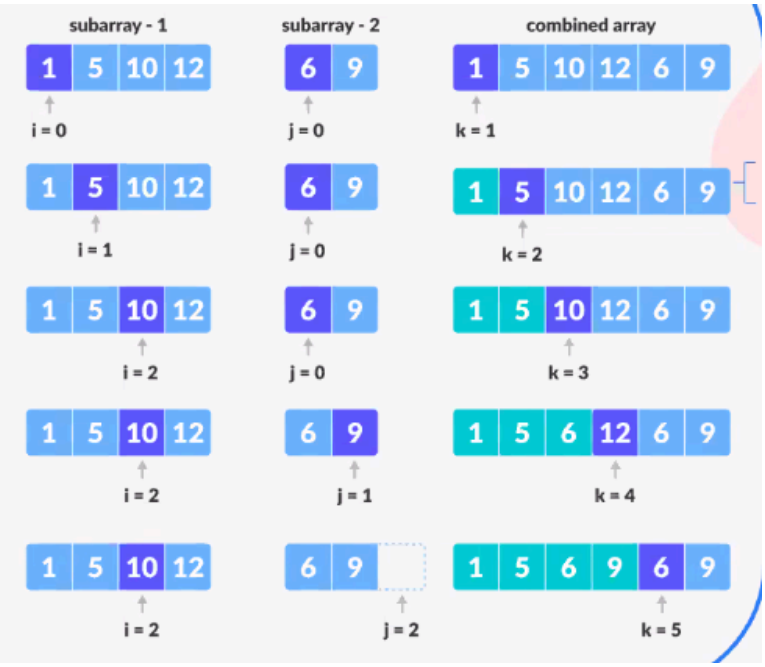
Step 2:

Maintain current index of sub-arrays and main array



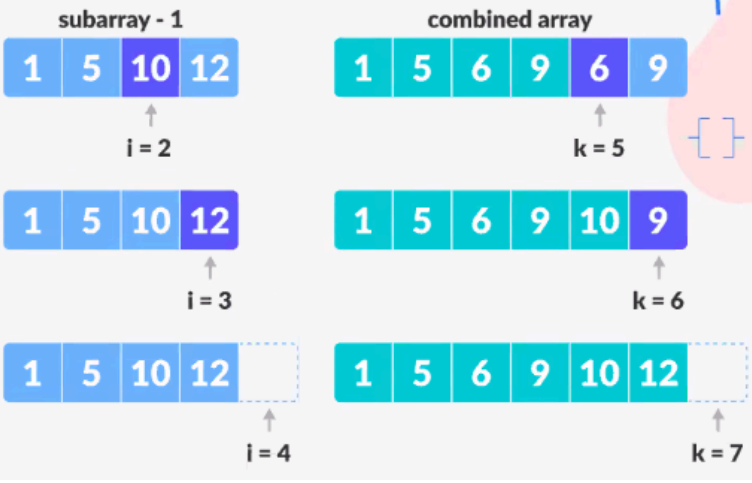
Step 3:

Until we reach the end of either L or M, pick larger among elements L and M and place them in the correct position at A[p..r]



Step 4:

When we run out of elements in either L or M, pick up the remaining elements and put in A[p..r]



24 - Day

Quicksort Algorithm

Quicksort is a sorting algorithm based on the **divide and conquer approach** where,

1. An array is divided into subarrays by selecting a **pivot element** (element selected from the array).

While dividing the array, the pivot element should be positioned in such a way that elements less than pivot are kept on the left side and elements greater than pivot are on the right side of the pivot.

2. The left and right subarrays are also divided using the same approach. This process continues until each subarray contains a single element.

3. At this point, elements are already sorted. Finally, elements are combined to form a sorted array.

Working of Quicksort Algorithm

1. Select the Pivot Element

There are different variations of quicksort where the pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

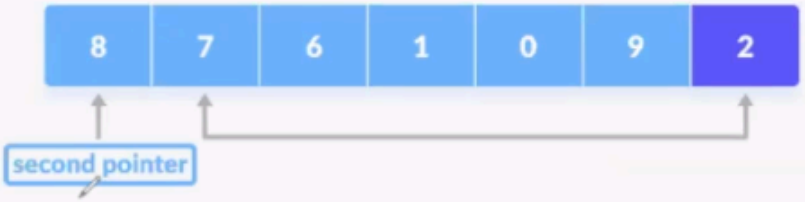
2. Rearrange the Array

Now the elements of the array are rearranged so that elements that are smaller than the pivot are put on the left and the elements greater than the pivot are put on the right.

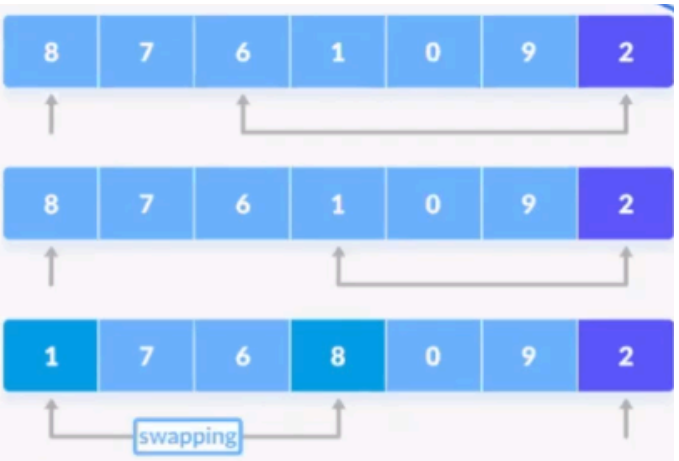
Here's how we rearrange the array:

1. A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.

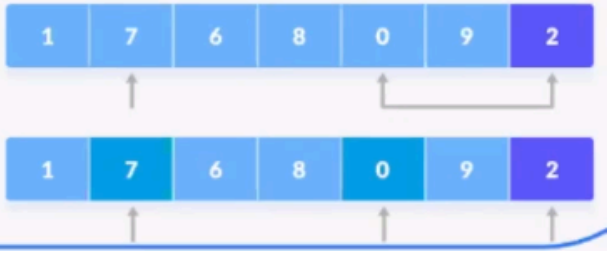
2. If the element is greater than the pivot element, a second pointer is set for that element.



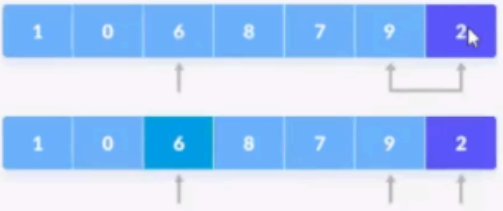
3. Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



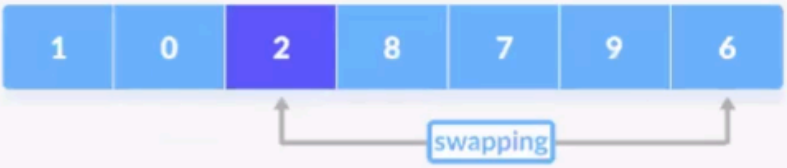
4. Again, the process is repeated to set the next greater element as the second pointer. And, swap it with another smaller element.



5. The process goes on until the second last element is reached.



6. Finally, the pivot element is swapped with the second pointer.



3. Divide Subarrays

Pivot elements are again chosen for the left and the right sub-parts separately. And, **step 2** is repeated.

The subarrays are divided until each subarray is formed of a single element. At this point, the array is already sorted.

Linear Search

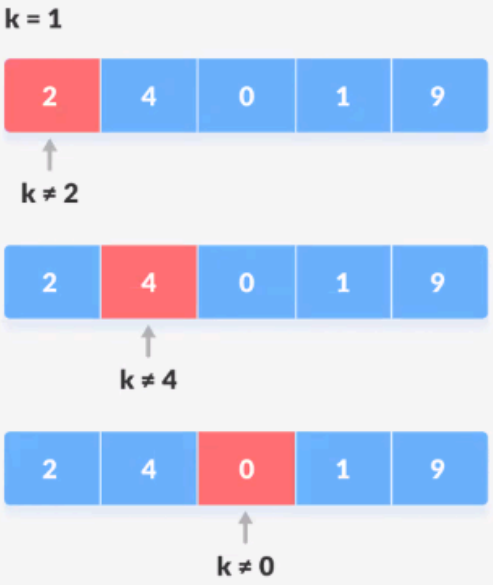
Linear Search

Linear search is a sequential searching algorithm where we start from one end and check every element of the list until the desired element is found. It is the simplest searching algorithm.

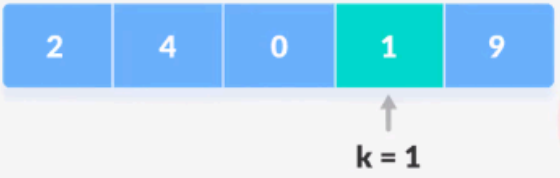
How Linear Search Works?

The following steps are followed to search for an element k = 1 in the list below.

1. Start from the first element, compare k with each element x.



2. If x == k, return the index.



3. Else, return not found.

Binary Search

Binary Search is a searching algorithm for finding an element's position in a sorted array.

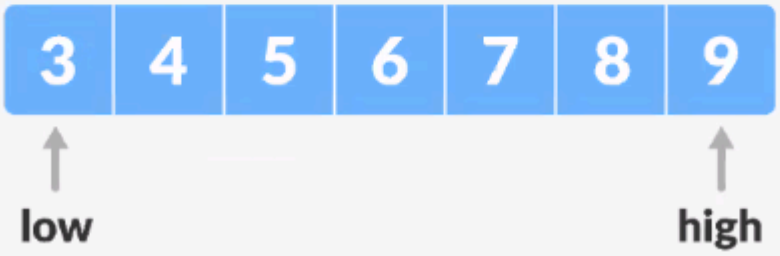
In this approach, the element is always searched in the middle of a portion of an array.

Note:
Binary search can be implemented only on a sorted list of items. If the elements are not sorted already, we need to sort them first.

1. The array in which searching is to be performed is:

Let $x = 4$ be the element to be searched.

2. Set two pointers low and high at the lowest and the highest positions respectively.



4. Find the middle position mid of the array ie. mid = (low + high)/2 and arr[mid] = 6.

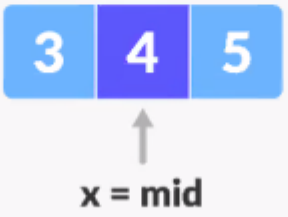
5. If x == arr[mid], then return mid. Else, compare the element to be searched with arr[mid].

6. If x > arr[mid], compare x with the middle element of the elements on the right side of arr[mid]. This is done by setting low to low = mid + 1.

7. Else, compare x with the middle element of the elements on the left side of arr[mid]. This is done by setting high to high = mid - 1.

7. Repeat steps 3 to 6 until low meets high.

8. x = 4 is found.



Binary Search Working

Binary Search Algorithm can be implemented in two ways which are discussed below.

1.Iterative Method

2.Recursive Method

The recursive method follows the divide and conquer approach.

The general steps for both methods are discussed below.

DSA