BINARY TREE OPERATIONS

Binary Tree Traversal

Going through a Tree by visiting every node, one node at a time, is called **traversal**.

Since Arrays and Linked Lists are linear data structures, there is only one obvious way to traverse these: start at the first element, or node, and continue to visit the next until you have visited them all.

But since a Tree can branch out in different directions (non-linear), there are different ways of traversing Trees.

There are two main categories of Tree traversal methods:

Breadth First Search (BFS) is when the nodes on the same level are visited before going to the next level in the tree. This means that the tree is explored in a more sideways direction.

Depth First Search (DFS) is when the traversal moves down the tree all the way to the leaf nodes, exploring the tree branch by branch in a downwards direction.

There are three different types traversals:

- pre-order
- in-order
- post-order

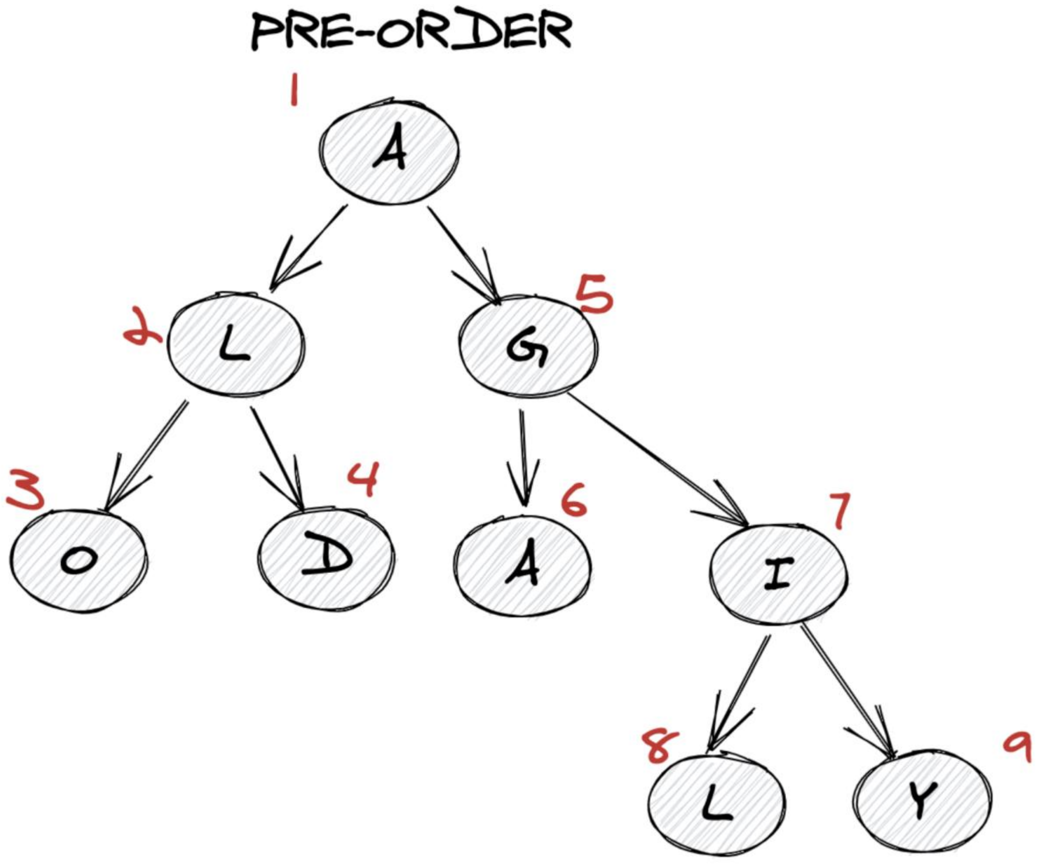
Pre-order Traversal of Binary Trees

Pre-order Traversal is done by visiting the root node first, then recursively do a pre-order traversal of the left subtree, followed by a recursive pre-order traversal of the right subtree. It's used for creating a copy of the tree, prefix notation of an expression tree, etc.

This traversal is "pre" order because the node is visited "before" the recursive pre-order traversal of the left and right subtrees.

Preorder traversal

1. Visit root node
2. Then go to all the nodes in the left subtree
3. Visit all the nodes in the right subtree



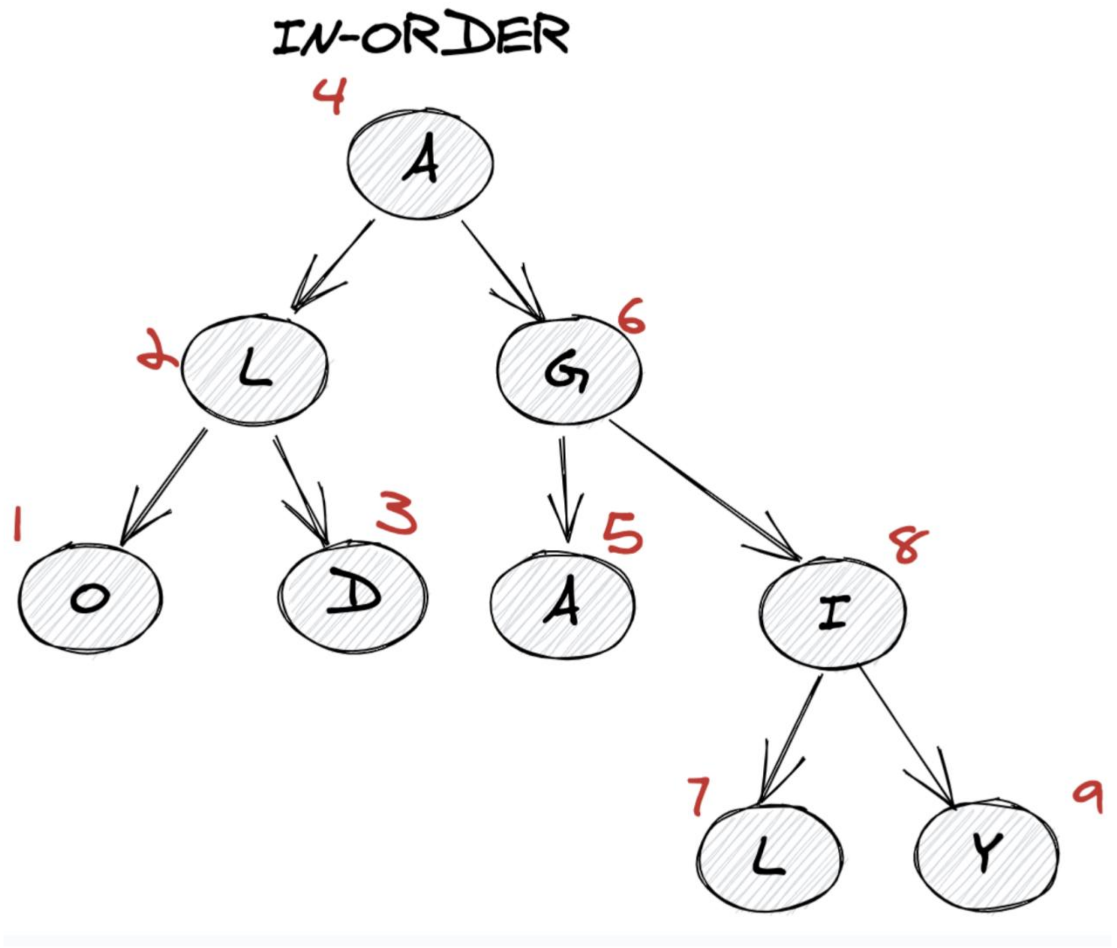
In-order Traversal of Binary Trees

In-order Traversal does a recursive In-order Traversal of the left subtree, visits the root node, and finally, does a recursive In-order Traversal of the right subtree. This traversal is mainly used for Binary Search Trees where it returns values in ascending order.

What makes this traversal "in" order, is that the node is visited in between the recursive function calls. The node is visited after the In-order Traversal of the left subtree, and before the In-order Traversal of the right subtree.

Inorder traversal

1. First, visit all the nodes in the left subtree
2. Then the root node
3. Visit all the nodes in the right subtree



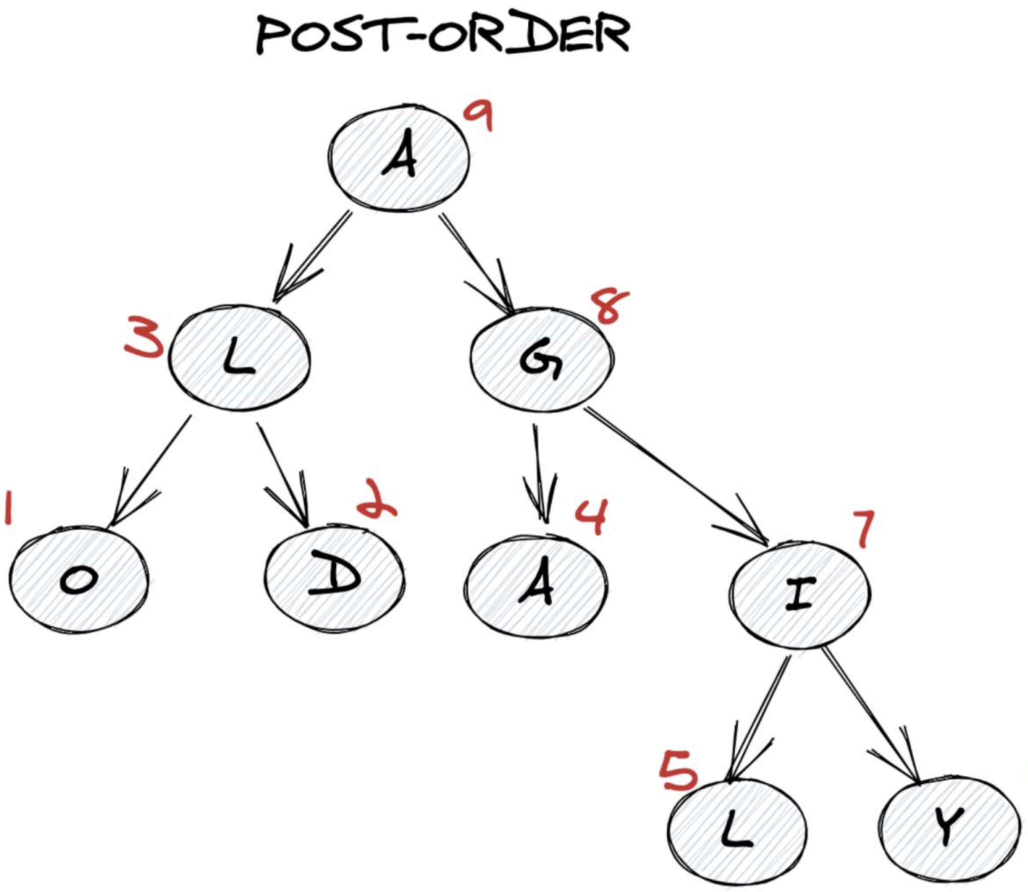
Post-order Traversal of Binary Trees

Post-order Traversal works by recursively doing a Post-order Traversal of the left subtree and the right subtree, followed by a visit to the root node. It is used for deleting a tree, post-fix notation of an expression tree, etc.

What makes this traversal "post" is that visiting a node is done "after" the left and right child nodes are called recursively.

Postorder traversal

1. Visit all the nodes in the left subtree
2. Visit all the nodes in the right subtree
3. Visit the root node



APPLICATIONS OF BINARY TREES

File Systems

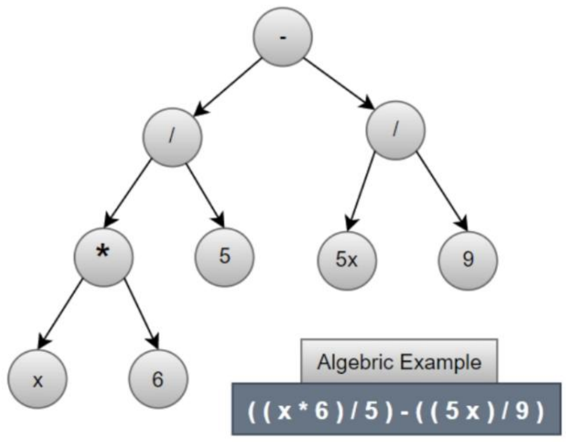
Binary trees are used in file systems to organise and store files. Each node in the tree represents a file or a directory, and the left child of a node represents a subdirectory or a file that is smaller than the node, while the right child represents a subdirectory or a file that is larger than the node.

Search Engines

Binary trees are used in search engines to organise and index web pages. Each node in the tree represents a web page, and the left child of a node represents a web page ranked lower than the node, while the right child represents a web page ranked higher than the node.

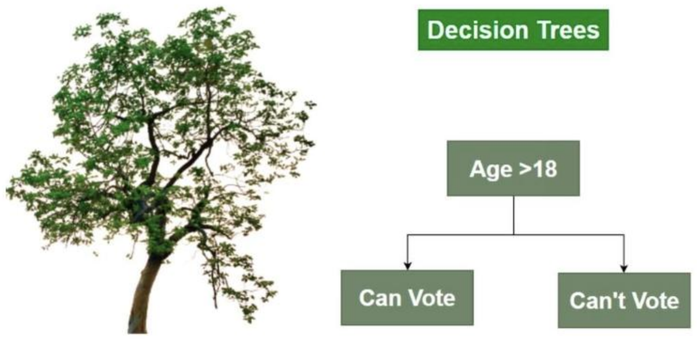
Expression Trees

Are you a maths lover? Because binary trees can also be used to represent maths expressions, such as equations. These are known as expression trees. Expression trees are basically used in evaluating and simplifying expressions and generating code.



Decision Trees

The decision tree is used in machine learning and data mining to model decisions and their consequences. In the decision tree, the inner node represents the test on an attribute, while the leaf node represents the result of the decision. Decision trees are used in various applications, such as fraud detection, customer segmentation, and medical diagnosis.

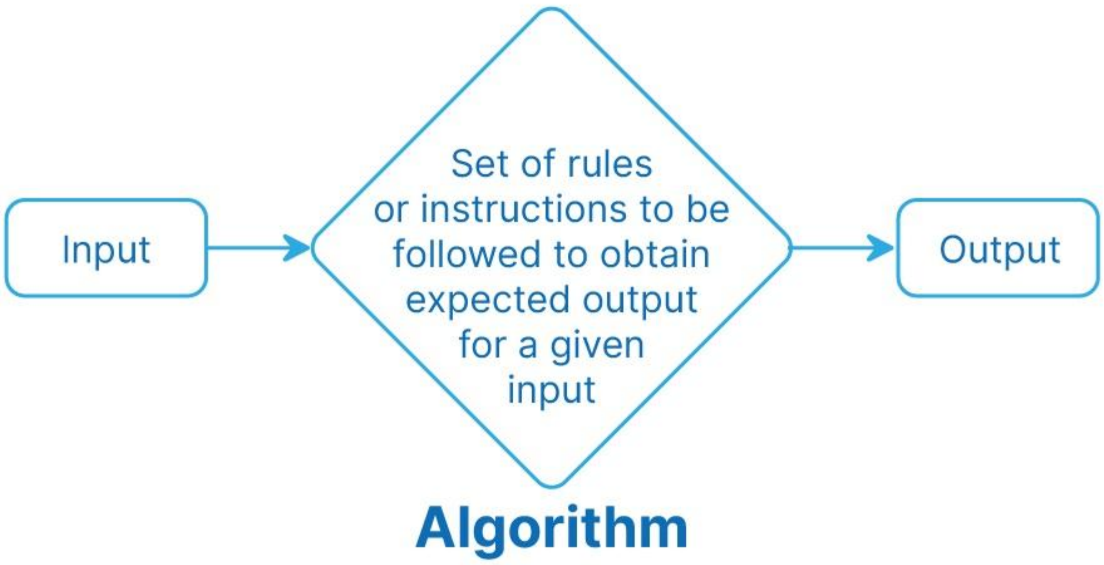


ALGORITHMS

CONCEPT OF ALGORITHMS

What is an Algorithm?

An algorithm is a set of commands that must be followed for a computer to perform calculations or other problem-solving operations.According to its formal definition, an **algorithm** is a finite set of instructions carried out in a specific order to perform a particular task. It is not the entire program or code; it is simple logic to a problem represented as an informal description in the form of a flowchart or pseudocode.


Input
Set of rules or instructions to be followed to obtain expected output for a given input
Output
Algorithm

Problem: A problem can be defined as a real-world problem or real-world instance problem for which you need to develop a program or set of instructions. An algorithm is a set of instructions.

Algorithm: An algorithm is defined as a step-by-step process that will be designed for a problem.

Input: After designing an algorithm, the algorithm is given the necessary and desired inputs.

Processing unit: The input will be passed to the processing unit, producing the desired output.

Output: The outcome or result of the program is referred to as the output.

How do Algorithms Work?

Algorithms are step-by-step procedures designed to solve specific problems and perform tasks efficiently in the realm of computer science and mathematics. These powerful sets of instructions form the backbone of modern technology and govern everything from web searches to artificial intelligence. Here's how algorithms work:

- **Input**: Algorithms take input data, which can be in various formats, such as numbers, text, or images.

- **Processing**: The algorithm processes the input data through a series of logical and mathematical operations, manipulating and transforming it as needed.

- **Output**: After the processing is complete, the algorithm produces an output, which could be a result, a decision, or some other meaningful information.

- **Efficiency**: A key aspect of algorithms is their efficiency, aiming to accomplish tasks quickly and with minimal resources.

- **Optimization**: Algorithm designers constantly seek ways to optimize their algorithms, making them faster and more reliable.

- **Implementation**: Algorithms are implemented in various programming languages, enabling computers to execute them and produce desired outcomes.

What is the Need for Algorithms?

You require algorithms for the following reasons:

Scalability

It aids in your understanding of scalability. When you have a sizable real-world problem, you must break it down into small steps to analyze it quickly.

Performance

The real world is challenging to break down into smaller steps. If a problem can be easily divided into smaller steps, it indicates that the problem is feasible.

After understanding what is an algorithm, why you need an algorithm, you will look at how to write one using an example.

Use of the Algorithms

Algorithms are essential to many disciplines because they offer organized, practical answers to challenging issues. Here are a few significant applications of algorithms in various fields:

1. Data Analysis and Machine Learning

Algorithms are used in data analysis and machine learning to find patterns in big datasets and forecast outcomes. Thanks to machine learning methods like support vector machines, decision trees, and neural networks, computers can learn from data and improve over time. These techniques are essential for applications such as recommendation systems, natural language processing, and picture recognition.

2. Cryptography & Security

Cryptography algorithms safeguard data by using encryption and decryption techniques, guaranteeing safe data storage and communication. Algorithms such as SHA-256, AES, and RSA are commonly employed in data integrity maintenance, user authentication, and sensitive information security. These algorithms comprise the foundation of cybersecurity measures used in secure communications, data encryption, and online transactions.

3. Information Retrieval and Search Engines

Search engines can efficiently index and retrieve pertinent information thanks to search algorithms such as PageRank and Hummingbird. By prioritizing web pages according to their significance and relevancy, these algorithms assist users in locating the most relevant information available online. Effective search algorithms are necessary to manage the enormous volume of online information.

4. Optimization problems

Optimization methods are utilized to select the optimal answer from various options. In various industries, including banking, engineering, logistics, and artificial intelligence, sophisticated issues are resolved using methods like gradient descent, linear programming, and genetic algorithms. These algorithms increase productivity, reduce expenses, and optimize resources for operations and decision-making.

5. Genomics and medical diagnostics

Due to their ability to analyze medical images, forecast disease outbreaks, and recognize genetic changes, algorithms are indispensable in medical diagnostics. Personalized medicine has been transformed by machine learning algorithms, in particular, which enable the creation of customized treatment regimens based on each patient's unique genetic profile. Additionally, algorithms help to speed up the sequencing and interpretation of genomic data, improving biotechnology and genomics research.

Characteristics of an Algorithm

An algorithm is a methodical process used to solve a task or solve a problem. Several important factors impact an algorithm's effectiveness:

1. Finiteness

An algorithm must always have a finite number of steps before it ends. When the operation is finished, it must have a defined endpoint or output and not enter an endless loop.

2. Definiteness

An algorithm needs to have exact definitions for each step. Clear and straightforward directions ensure that every step is understood and can be taken easily.

3. Input

An algorithm requires one or more inputs. The values that are first supplied to the algorithm before its processing are known as inputs. These inputs come from a predetermined range of acceptable values.

4. Output

One or more outputs must be produced by an algorithm. The output is the outcome of the algorithm after every step has been completed. The relationship between the input and the result should be clear.

5. Effectiveness

An algorithm's stages must be sufficiently straightforward to be carried out in a finite time utilizing fundamental operations. With the resources at hand, every operation in the algorithm should be doable and practicable.

6. Generality

Rather than being limited to a single particular case, an algorithm should be able to solve a group of issues. It should offer a generic fix that manages a variety of inputs inside a predetermined range or domain.

Qualities of a Good Algorithm

- **Efficiency**: A good algorithm should perform its task quickly and use minimal resources.
- **Correctness**: It must produce the correct and accurate output for all valid inputs.
- **Clarity**: The algorithm should be easy to understand and comprehend, making it maintainable and modifiable.
- **Scalability**: It should handle larger data sets and problem sizes without a significant decrease in performance.
- **Reliability**: The algorithm should consistently deliver correct results under different conditions and environments.
- **Optimality**: Striving for the most efficient solution within the given problem constraints.
- **Robustness**: Capable of handling unexpected inputs or errors gracefully without crashing.

- **Adaptability**: Ideally, it can be applied to a range of related problems with minimal adjustments.
- **Simplicity**: Keeping the algorithm as simple as possible while meeting its requirements, avoiding unnecessary complexity.

Advantage and Disadvantages of Algorithms

Advantages of Algorithms:

- **Efficiency**: Algorithms streamline processes, leading to faster and more optimized solutions.
- **Reproducibility**: They yield consistent results when provided with the same inputs.
- **Problem-solving**: Algorithms offer systematic approaches to tackle complex problems effectively.
- **Scalability**: Many algorithms can handle larger datasets and scale with increasing input sizes.
- **Automation**: They enable automation of tasks, reducing the need for manual intervention.

Disadvantages of Algorithms:

- **Complexity**: Developing sophisticated algorithms can be challenging and time-consuming.
- **Limitations**: Some problems may not have efficient algorithms, leading to suboptimal solutions.
- **Resource Intensive**: Certain algorithms may require significant computational resources.
- **Inaccuracy**: Inappropriate algorithm design or implementation can result in incorrect outputs.
- **Maintenance**: As technology evolves, algorithms may require updates to stay relevant and effective.

HOW TO WRITE AN ALGORITHMS

as we all know, an algorithms is a basic tool and very useful for understanding a problem through it's step by step analysis . However there are no well-defined standards for writting algorithms;- althought it depends on the problem and available resources.

the following are the steps:-

a. determine the input and output of the problem.

b. find the correct data structure to present the problem.

c. try to reduce the problem to a variation of a well-known one.

d. decide whether you look for a recursive or imperative one or mixed algorithm.

e. write an algorithm.

example:- Algorithm for multiplication of two numbers.

solution:-

step 1: Start

step2: Enter the first number.

step 3: Enter the second number.

step 4: Multiply two numbers and store the result.

step 5: Display the result.

step 6: Stop.

TASK:

Write the following algorithms

1. algorithm to find the largest number between two numbers.

2. write an algorithm to find the area of a squeare.

3. write an algothms to find simple interest.

4. write an algorithm to find compound interest.

TYPES OF ALGORITHMS.

There are many algorithms, but for your level we are going to learn two fundamental types :-

i. Recursive Algorithms.

ii. Iterative Algorithms.

RECURSIVE ALGORITHMS

This is an algorithm that calls itself repeatedly with a smaller value as inputs generated after solving the current inputs until the problem is solved.

Recursion is a programming technique where a function calls itself to solve smaller instances of a problem until a base case is met, at which point the recursive calls stop.

Key Concepts:

- **Base Case:** The condition under which the recursion ends. Without a base case, the recursion would continue indefinitely, leading to a stack overflow.

- **Recursive Case:** The part of the function where it calls itself with a modified argument, moving towards the base case.

Analogy: Think of recursion as standing between two mirrors: each reflection is a smaller version of the original image, continuing until it becomes too small to see (the base case).

Why Use Recursion? Recursion is often used to simplify the code for problems that have a natural recursive structure, like tree traversals, the Fibonacci sequence, and factorial calculations.

Implementation of Recursive Algorithms

Algorithm: Recursive Factorial Calculation

Input:

- A non-negative integer n.

Output:

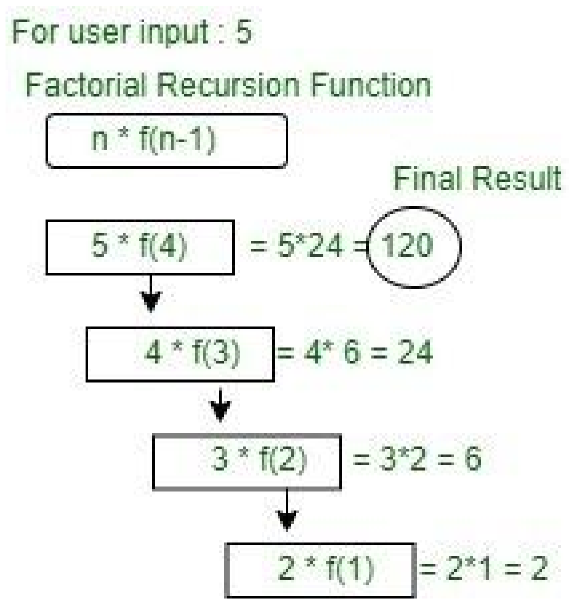
- The factorial of n, denoted as n!.

Steps:

1. **Base Case:**
    - If n is 0 or 1, then:
        - Return 1.
        - (Reason: By definition, 0! = 1 and 1! = 1).
2. **Recursive Case:**
    - If n is greater than 1, then:
        - Calculate the factorial of (n - 1) by calling the algorithm recursively.
        - Multiply n by the result of the recursive call.
        - Return the product.
3. **End of Algorithm.**

Example:

- If n = 5:
    - The algorithm calculates `5 * (4 * (3 * (2 * (1 * 1)))) = 120`.



Advantages and Disadvantages of Recursive Algorithms

Advantages:

- it reduces the number of lines of code in a program.
- it is clear and straight for the programmer to understand.
- recursive algorithms simply solve complex problems.

Disadvantages:

- **Performance Overhead:** Recursive calls can lead to significant overhead due to function call stack management, especially for deep recursions.

- **Stack Overflow:** If the recursion depth is too high (e.g., no proper base case or a large input), it can cause a stack overflow.
- **Difficult to Debug:** Recursive functions can be harder to trace and debug compared to iterative solutions, especially for complex problems.

TASK:

1. Write a recursive algorithm to find the sum of n-integers.
2. differentiate btn algorithm and program.
3. explain which effect will be arisen when the base case is not defined in a recursive function?

ITERATIVE ALGORITHMS

Definition: Iterative algorithms repeatedly execute a set of instructions until a specific condition is met. Unlike recursion, which solves a problem by breaking it down into smaller sub-problems, iteration solves a problem by repeatedly applying the same operations.

Key Concepts:

- **Loop:** The primary structure used in iterative algorithms, such as for, while, or do-while loops. A loop allows you to execute a block of code multiple times.
- **Condition:** A boolean expression that determines when the loop should stop. It's checked before each iteration.
- **Iteration:** Each execution of the loop's body is called an iteration.

Why Use Iteration? Iteration is often more efficient than recursion in terms of memory usage because it doesn't involve the overhead of multiple function calls. It's ideal for problems where you need to repeat an action a known number of times or until a condition is met.

Implementation of Iterative Algorithms

Basic Structure:

1. **Initialize variables:** Set up any variables that control the loop or store results.
2. **Loop condition:** Define the condition under which the loop will continue executing.
3. **Loop body:** Write the code that will be executed during each iteration.

4. **Update:** Modify the loop control variable to ensure that the loop progresses toward termination.
5. **Exit the loop:** When the loop condition is no longer true, the loop stops, and the algorithm proceeds to the next step.

Examples of Iterative Algorithms

Example 1: Factorial Calculation (Iterative Approach)

Problem: Compute the factorial of a number n, denoted as n! = n * (n-1) * (n-2) * ... * 1.

Algorithm:

Input:

A non-negative integer n.

Output:

The factorial of n.

Steps:

1. Initialize result to 1.
2. For each integer i from 1 to n:
   - Multiply result by i.
3. After the loop ends, result holds the value of n!.

4. Return result.

Example:

1. If n = 5, the loop calculates 1 * 1 * 2 * 3 * 4 * 5 = 120.

Advantages and Disadvantages of Iterative Algorithms

Advantages:

- **Efficiency:** Iterative algorithms generally use less memory than recursive algorithms because they don't involve the overhead of multiple function calls.
- **Simplicity:** They are easier to understand and debug for problems that involve simple repetition.
- **Avoidance of Stack Overflow:** Since they don't involve deep recursion, iterative algorithms are less likely to cause stack overflow.

Disadvantages:

- **Code Complexity:** For problems that naturally fit a recursive pattern, iterative solutions can be more complex and harder to write.
- **Less Intuitive:** Some problems, like tree traversals, are less intuitive when solved iteratively, leading to more complicated code.

TASK:

1. a.) design an algorithm to find the sum of the digits made from a positive number.

b.)create a c++ program to implement the algorithms.

2. what are the advantages of using iterative over recursive algorithms?explain.

SEARCHING ALGORITHMS

Searching algorithms are essential tools in computer science used to locate specific items within a collection of data. These algorithms are designed to efficiently navigate through data structures to find the desired information, making them fundamental in various applications such as databases, web search engines, and more.

What is Searching?

Searching is the fundamental process of locating a specific element or item within a collection of data. This collection of data can take various forms, such as arrays, lists, trees, or other structured representations. The primary objective of searching is to determine whether the desired element exists within the data, and if so, to identify its precise location or retrieve it. It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, and more.

Searching terminologies:

Target Element:

In searching, there is always a specific target element or item that you want to find within the data collection. This target could be a value, a record, a key, or any other data entity of interest.

Search Space:

The search space refers to the entire collection of data within which you are looking for the target element. Depending on the data structure used, the search space may vary in size and organization.

Complexity:

Searching can have different levels of complexity depending on the data structure and the algorithm used. The complexity is often measured in terms of time and space requirements.

Applications of Searching:

Searching algorithms have numerous applications across various fields. Here are some common applications:

- **Information Retrieval**: Search engines like Google, Bing, and Yahoo use sophisticated searching algorithms to retrieve relevant information from vast amounts of data on the web.

- **Database Systems**: Searching is fundamental in database systems for retrieving specific data records based on user queries, improving efficiency in data retrieval.

- **E-commerce**: Searching is crucial in e-commerce platforms for users to find products quickly based on their preferences, specifications, or keywords.

- **Networking**: In networking, searching algorithms are used for routing packets efficiently through networks, finding optimal paths, and managing network resources.

- **Artificial Intelligence**: Searching algorithms play a vital role in AI applications, such as problem-solving, game playing (e.g., chess), and decision-making processes

- **Pattern Recognition**: Searching algorithms are used in pattern matching tasks, such as image recognition, speech recognition, and handwriting recognition.

Linear Search Algorithm

The linear search algorithm is defined as a sequential search algorithm that starts at one end and goes through each element of a list until the desired element is found; otherwise, the search continues till the end of the dataset. In this article, we will learn about the basics of the linear search algorithm, its applications, advantages, disadvantages, and more to provide a deep understanding of linear search.

What is Linear Search Algorithm?

Linear search is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as sequential search.

Algorithm for Linear Search Algorithm:

The algorithm for linear search can be broken down into the following steps:

a. **Start**: Begin at the first element of the collection of elements.

b. **Compare**: Compare the current element with the desired element.

c. **Found**: If the current element is equal to the desired element, return true or index to the current element.

d. **Move**: Otherwise, move to the next element in the collection.

e. **Repeat**: Repeat steps 2-4 until we have reached the end of collection.

f. **Not found**: If the end of the collection is reached without finding the desired element, return that the desired element is not in the array.

How Does Linear Search Algorithm Work?

In Linear Search Algorithm,

Every element is considered as a potential match for the key and checked for the same.

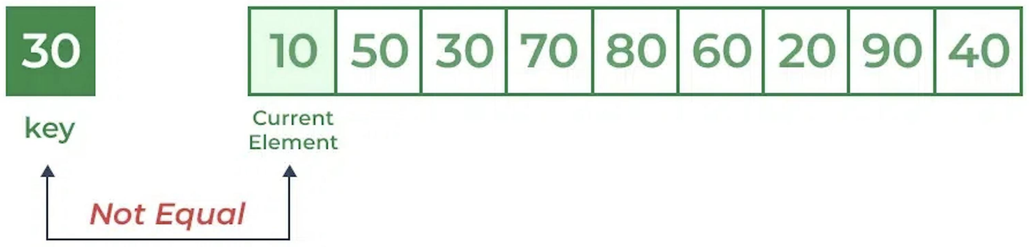
If any element is found equal to the key, the search is successful and the index of that element is returned.

If no element is found equal to the key, the search yields "No match found".

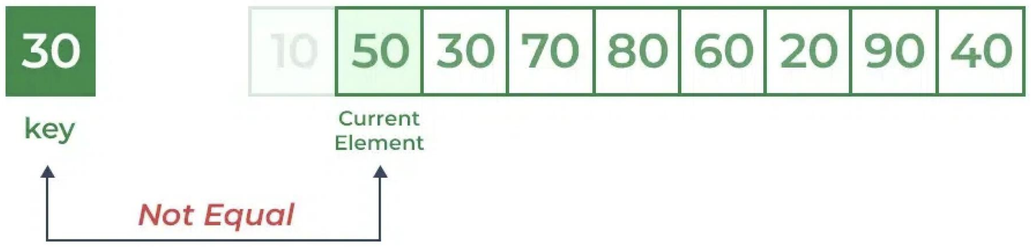
For example: Consider the array arr[] = {10, 50, 30, 70, 80, 20, 90, 40} and key = 30

Step 1: Start from the first element (index 0) and compare key with each element (arr[i]).

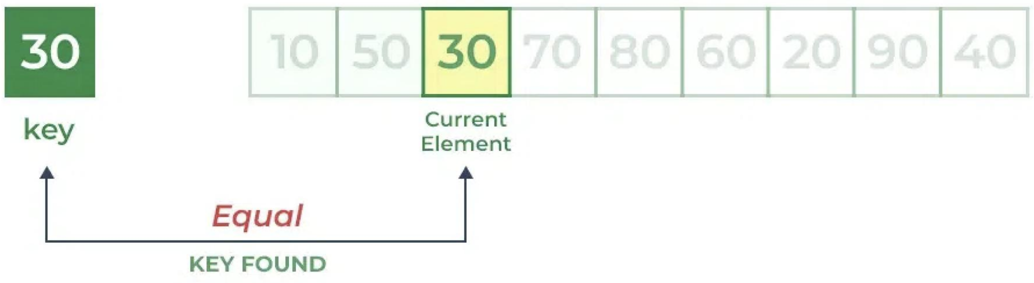
Comparing key with first element arr[0]. SInce not equal, the iterator moves to the next element as a potential match.



Comparing key with next element arr[1]. SInce not equal, the iterator moves to the next element as a potential match.



Step 2: Now when comparing arr[2] with key, the value matches. So the Linear Search Algorithm will yield a successful message and return the index of the element when key is found (here 2).



Time and Space Complexity of Linear Search Algorithm:

Time Complexity:

Best Case: In the best case, the key might be present at the first index. So the best case complexity is O(1)

Worst Case: In the worst case, the key might be present at the last index i.e., opposite to the end from which the search has started in the list. So the worst-case complexity is O(N) where N is the size of the list.

Average Case: O(N)

Auxiliary Space: O(1) as except for the variable to iterate through the list, no other variable is used.

Applications of Linear Search Algorithm:

- Unsorted Lists: When we have an unsorted array or list, linear search is most commonly used to find any element in the collection.

- Small Data Sets: Linear Search is preferred over binary search when we have small data sets with

- Searching Linked Lists: In linked list implementations, linear search is commonly used to find elements within the list. Each node is checked sequentially until the desired element is found.

- Simple Implementation: Linear Search is much easier to understand and implement as compared to Binary Search or Ternary Search.

Advantages of Linear Search Algorithm:

- Linear search can be used irrespective of whether the array is sorted or not. It can be used on arrays of any data type.

- Does not require any additional memory.

- It is a well-suited algorithm for small datasets.

Disadvantages of Linear Search Algorithm:

- Linear search has a time complexity of O(N), which in turn makes it slow for large datasets.

- Not suitable for large arrays.

When to use Linear Search Algorithm?

- When we are dealing with a small dataset.

- When you are searching for a dataset stored in contiguous memory.