Unit I

Introduction to Data Structures

A data structure is a particular way of organising data in a computer so that it can be used effectively. The idea is to reduce the space and time complexities of different tasks. The choice of a good data structure makes it possible to perform a variety of critical operations effectively. An efficient data structure also uses minimum memory space and execution time to process the structure. A data structure is not only used for organising the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or software system that has been developed. So we must have good knowledge of data structures.

Need Of Data Structure:

The structure of the data and the synthesis of the algorithm are relative to each other. Data presentation must be easy to understand so the developer, as well as the user, can make an efficient implementation of the operation.

Data structures provide an easy way of organising, retrieving, managing, and storing data. Here is a list of the needs for data.

- Data structure modification is easy.
- It requires less time.
- Save storage memory space.
- Data representation is easy.
- Easy access to the large database

Classification/Types of Data Structures:

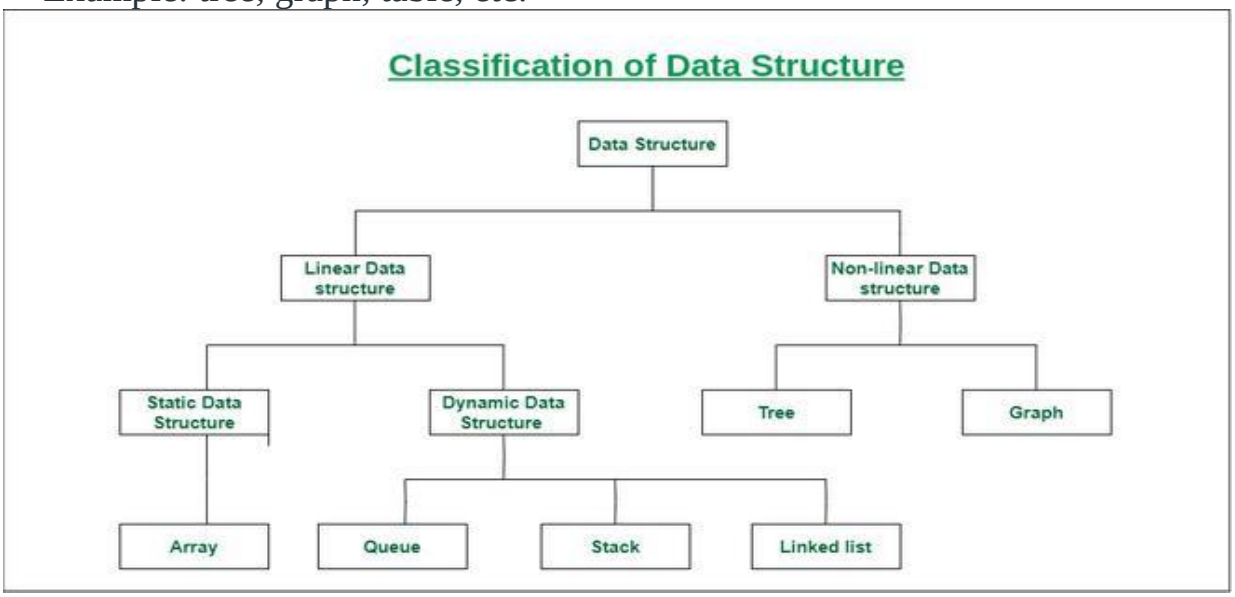
1. Linear Data Structure
2. Non-Linear Data Structure.

Linear Data Structure:

- Elements are arranged in one dimension ,also known as linear dimension.
- Example: lists, stack, queue, etc.

Non-Linear Data Structure

- Elements are arranged in one-many, many-one and many-many dimensions.
- Example: tree, graph, table, etc.

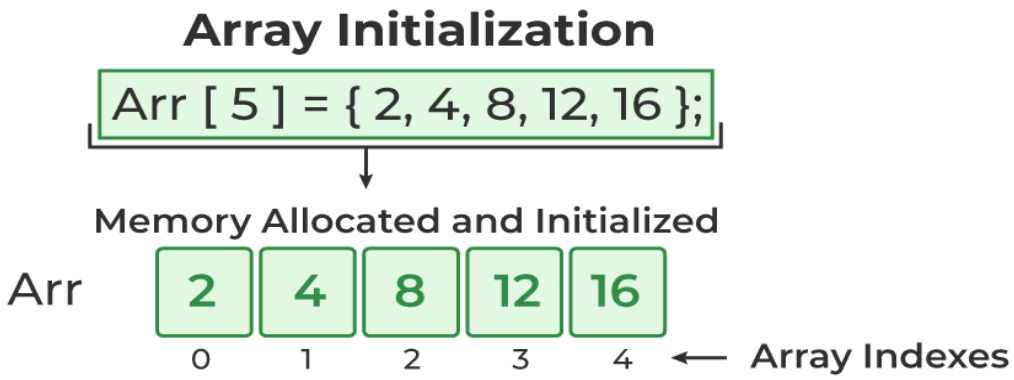


Please refer **DSA Tutorial** for complete tutorial on Data Strictures with guide on every topic, practice problems and top interview questions.

Most Popular Data Structures:

1. Array:

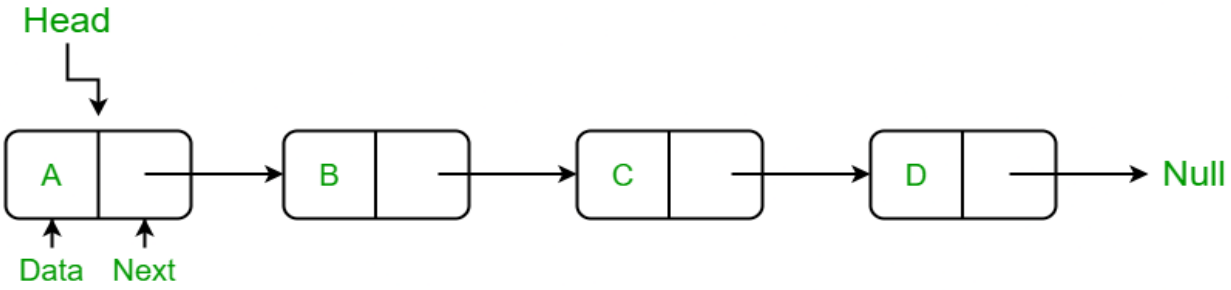
An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



Array Data Structure

2. Linked Lists:

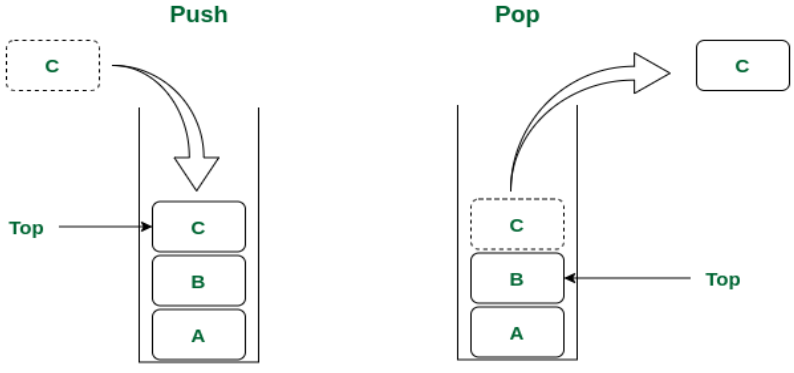
Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



Linked Data Structure

3. Stack:

Stack is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO(Last In First Out) or FILO(First In Last Out). In stack, all insertion and deletion are permitted at only one end of the list.



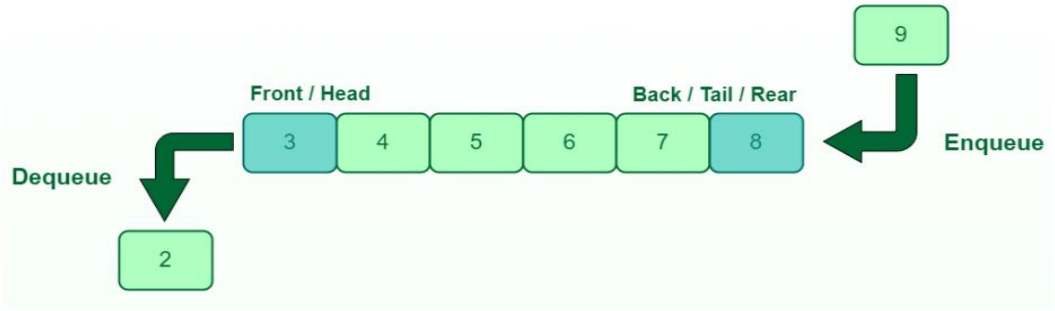
Stack Data Structure

Stack Operations:

- **push():** When this operation is performed, an element is inserted into the stack.
- **pop():** When this operation is performed, an element is removed from the top of the stack and is returned.
- **top():** This operation will return the last inserted element that is at the top without removing it.
- **size():** This operation will return the size of the stack i.e. the total number of elements present in the stack.
- **isEmpty():** This operation indicates whether the stack is empty or not.

4. Queue:

Like Stack, Queue is a linear structure which follows a particular order in which the operations are performed. The order is First In First Out (FIFO). In the queue, items are inserted at one end and deleted from the other end. A good example of the queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.



Queue Data Structure

Queue Data Structure

Queue Operations:

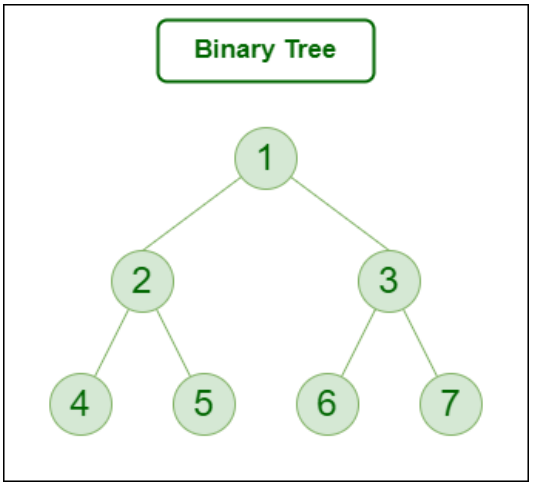
- **Enqueue():** Adds (or stores) an element to the end of the queue..
- **Dequeue():** Removal of elements from the queue.
- **Peek() or front():** Acquires the data element available at the front node of the queue without deleting it.
- **rear():** This operation returns the element at the rear end without removing it.
- **isFull():** Validates if the queue is full.
- **isNull():** Checks if the queue is empty.

5. Binary Tree:

Unlike Arrays, Linked Lists, Stack and queues, which are linear data structures, trees are hierarchical data structures. A binary tree is a tree data structure in which each node has at most two children, which are referred to as the left child and the right child. It is implemented mainly using Links.

A Binary Tree is represented by a pointer to the topmost node in the tree. If the tree is empty, then the value of root is NULL. A Binary Tree node contains the following parts.

1. Data
2. Pointer to left child
3. Pointer to the right child



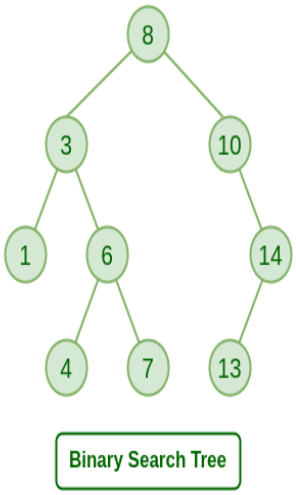
Binary Tree Data Structure

6. Binary Search Tree:

A Binary Search Tree is a Binary Tree following the additional properties:

- The left part of the root node contains keys less than the root node key.
- The right part of the root node contains keys greater than the root node key.
- There is no duplicate key present in the binary tree.

A Binary tree having the following properties is known as Binary search tree (BST).

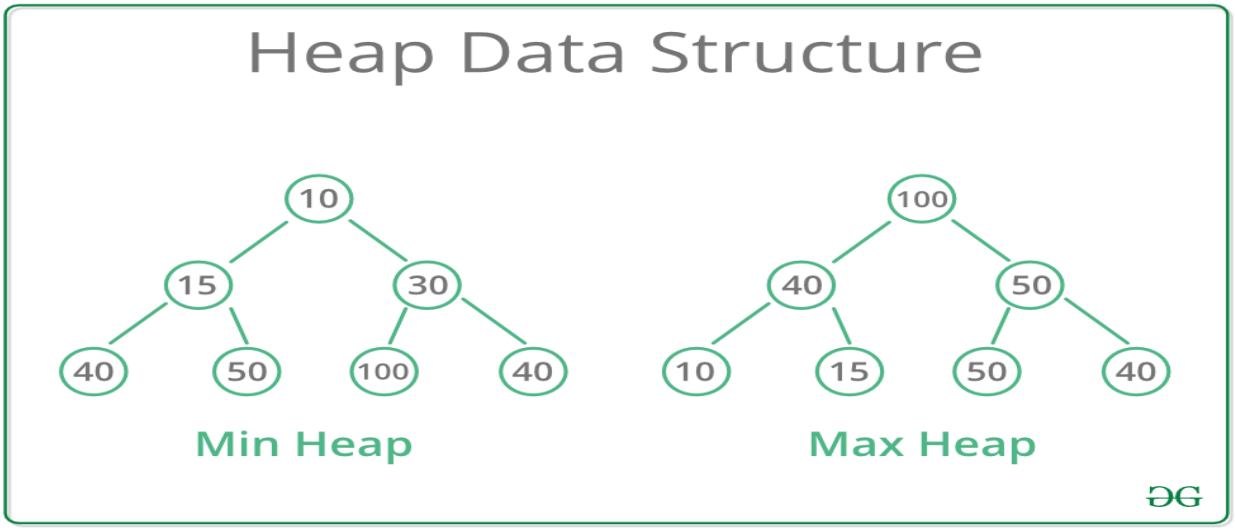


Binary Search Tree Data Structure

7. Heap:

A Heap is a special Tree-based data structure in which the tree is a complete binary tree. Generally, Heaps can be of two types:

- **Max-Heap:** In a Max-Heap the key present at the root node must be greatest among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.
- **Min-Heap:** In a Min-Heap the key present at the root node must be minimum among the keys present at all of its children. The same property must be recursively true for all sub-trees in that Binary Tree.

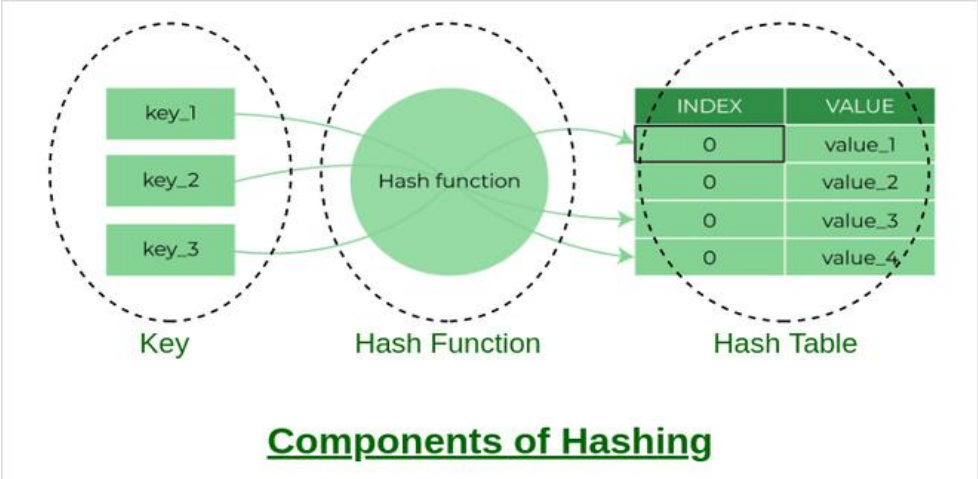


Max and Min Heap

8. Hash Table Data Structure:

Hashing is an important Data Structure which is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements. The efficiency of mapping depends on the efficiency of the hash function used.

Let a hash function H(x) maps the value x at the index x%10 in an Array. For example, if the list of values is [11, 12, 13, 14, 15] it will be stored at positions {1, 2, 3, 4, 5} in the array or Hash table respectively.

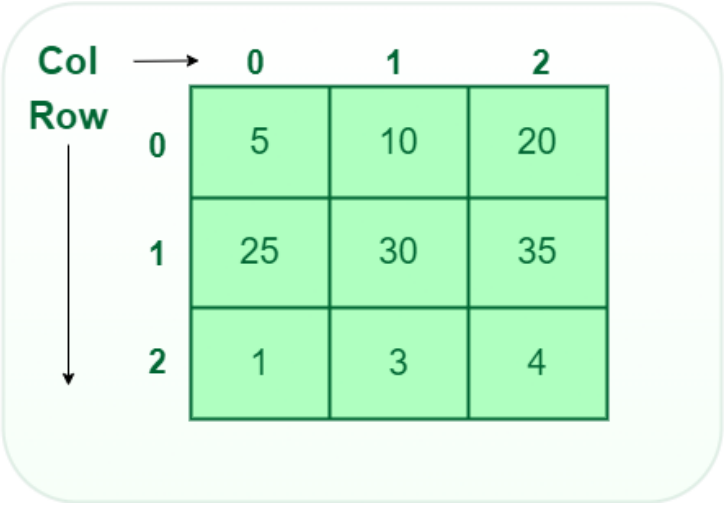


Hash Data Structure

9. Matrix:

A matrix represents a collection of numbers arranged in an order of rows and columns. It is necessary to enclose the elements of a matrix in parentheses or brackets.

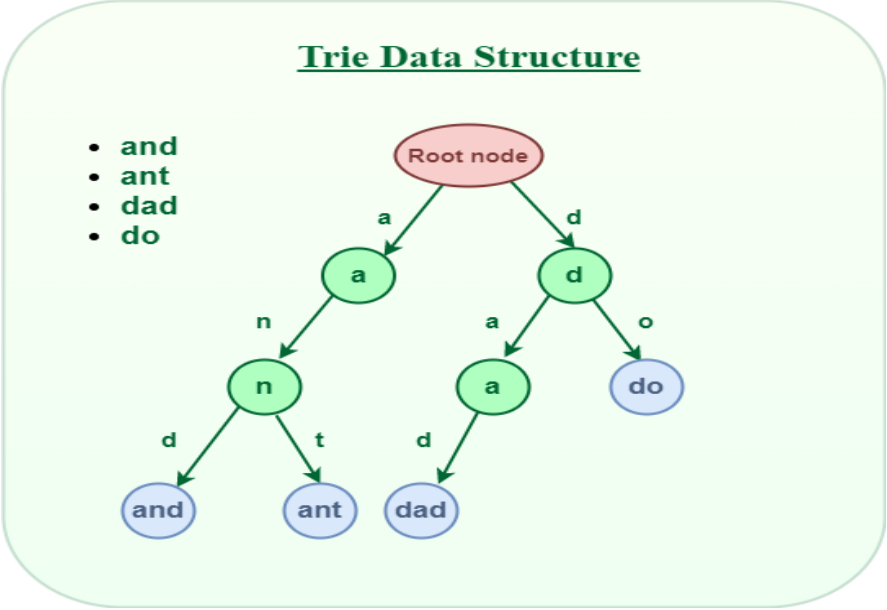
A matrix with 9 elements is shown below.



Matrix

10. Trie:

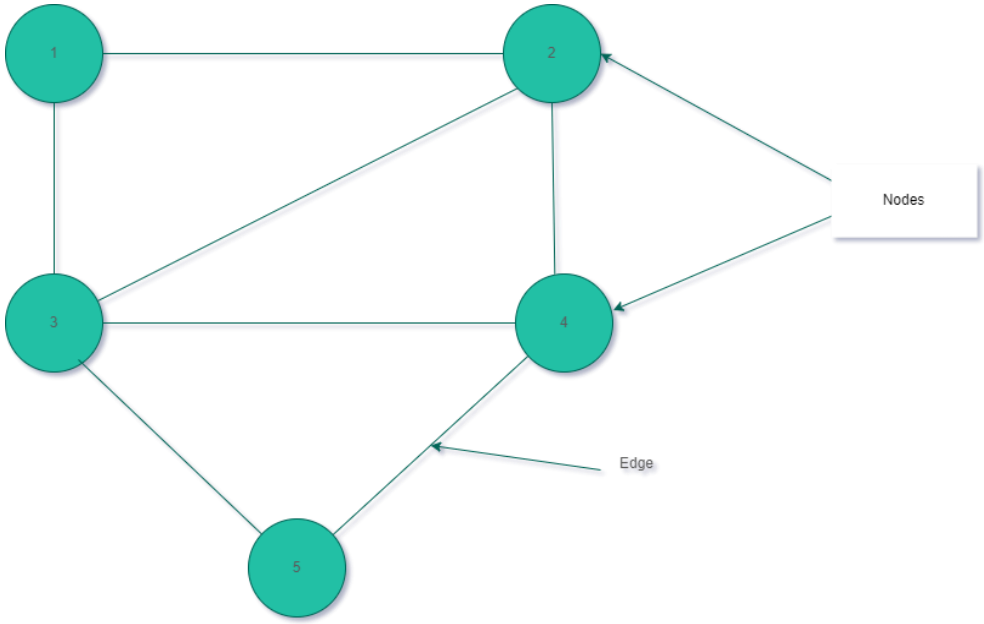
Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length). If we store keys in the binary search tree, a well-balanced BST will need time proportional to M * log N, where M is maximum string length and N is the number of keys in the tree. Using Trie, we can search the key in O(M) time. However, the penalty is on Trie storage requirements.



Trie Data Structure

11. Graph:

Graph is a data structure that consists of a collection of nodes (vertices) connected by edges. Graphs are used to represent relationships between objects and are widely used in computer science, mathematics, and other fields. Graphs can be used to model a wide variety of real-world systems, such as social networks, transportation networks, and computer networks.



Graph Data Structure

Applications of Data Structures:

Data structures are used in various fields such as:

- Operating system
- Graphics
- Computer Design
- Blockchain
- Genetics
- Image Processing
- Simulation,
- etc.

Abstract Data Types

An **Abstract Data Type (ADT)** is a conceptual model that defines a set of operations and behaviors for a data structure, **without specifying how these operations are implemented** or how data is organized in memory. The definition of ADT only mentions what **operations are to be performed** but not **how** these operations will be implemented. It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations. **It is called "abstract" because it provides an implementation-independent view.** The process of providing only the essentials and hiding the details is known as abstraction.

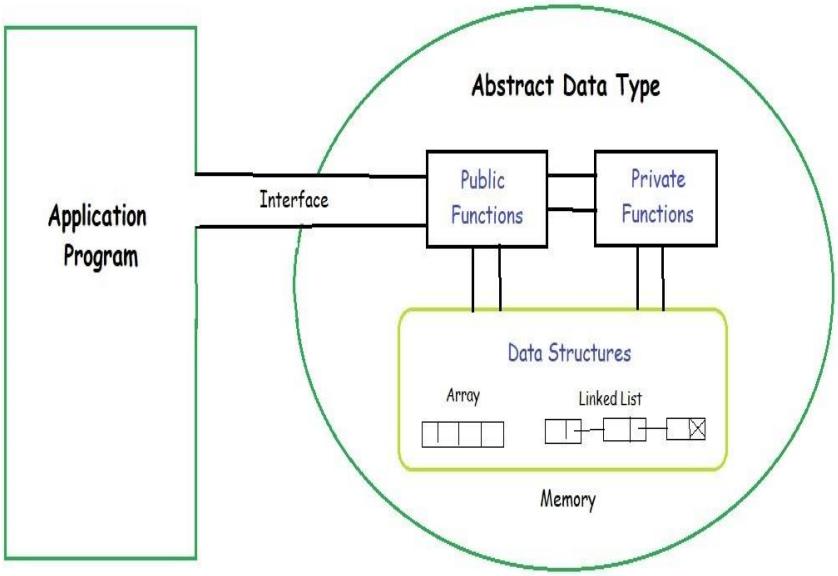
Features of ADT

Abstract data types (ADTs) are a way of encapsulating data and operations on that data into a single unit. Some of the key features of ADTs include:

- **Abstraction:** The user does not need to know the implementation of the data structure only essentials are provided.
- **Better Conceptualization:** ADT gives us a better conceptualization of the real world.
- **Robust:** The program is robust and has the ability to catch errors.

- **Encapsulation**: ADTs hide the internal details of the data and provide a public interface for users to interact with the data. This allows for easier maintenance and modification of the data structure.
- **Data Abstraction**: ADTs provide a level of abstraction from the implementation details of the data. Users only need to know the operations that can be performed on the data, not how those operations are implemented.
- **Data Structure Independence**: ADTs can be implemented using different data structures, such as arrays or linked lists, without affecting the functionality of the ADT.
- **Information Hiding:** ADTs can protect the integrity of the data by allowing access only to authorized users and operations. This helps prevent errors and misuse of the data.
- **Modularity**: ADTs can be combined with other ADTs to form larger, more complex data structures. This allows for greater flexibility and modularity in programming.

This image demonstrates how an Abstract Data Type (ADT) hides internal data structures (like arrays, linked lists) using public and private functions, exposing only a defined interface to the application program.



Why Use ADTs?

The key reasons to use ADTs in Java are listed below:

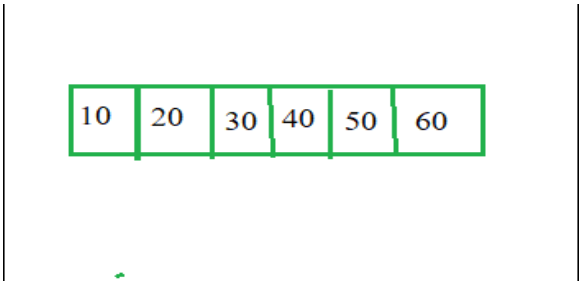
- **Encapsulation:** Hides complex implementation details behind a clean interface.
- **Reusability**: Allows different internal implementations (e.g., array or linked list) without changing external usage.
- **Modularity:** Simplifies maintenance and updates by separating logic.
- **Security:** Protects data by preventing direct access, minimizing bugs and unintended changes.

Examples of ADTs

Now, let's understand three common ADT's: List ADT, Stack ADT, and Queue ADT.

1. List ADT

The List ADT (Abstract Data Type) is a sequential collection of elements that supports a set of operations **without specifying the internal implementation**. It provides an ordered way to store, access, and modify data.



Vies of list

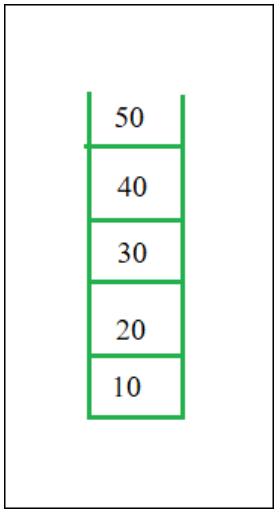
Operations:

The List ADT need to store the required data in the sequence and should have the following operations**:**

- **get():** Return an element from the list at any given position.
- **insert():** Insert an element at any position in the list.
- **remove():** Remove the first occurrence of any element from a non-empty list.
- **removeAt():** Remove the element at a specified location from a non-empty list.
- **replace():** Replace an element at any position with another element.
- **size():** Return the number of elements in the list.
- **isEmpty():** Return true if the list is empty; otherwise, return false.
- **isFull():** Return true if the list is full, otherwise, return false. Only applicable in fixed-size implementations (e.g., array-based lists).

2. Stack ADT

The Stack ADT is a linear data structure that follows the LIFO (Last In, First Out) principle. It allows elements to be added and removed only from one end, called the top of the stack.



View of stack

Operations:

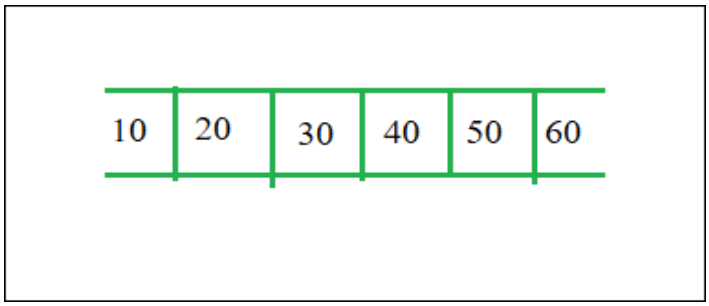
In Stack ADT, the order of insertion and deletion should be according to the FILO or LIFO Principle. Elements are inserted and removed from the same end, called the top of the stack. It should also support the following operations:

- **push():** Insert an element at one end of the stack called the top.
- **pop():** Remove and return the element at the top of the stack, if it is not empty.
- **peek():** Return the element at the top of the stack without removing it, if the stack is not empty.
- **size():** Return the number of elements in the stack.
- **isEmpty():** Return true if the stack is empty; otherwise, return false.

- **isFull():** Return true if the stack is full; otherwise, return false. Only relevant for fixed-capacity stacks (e.g., array-based).

3. Queue ADT

The Queue ADT is a linear data structure that follows the FIFO (First In, First Out) principle. It allows elements to be inserted at one end (rear) and removed from the other end (front).



View of Queue

Operations:

The Queue ADT follows a design similar to the Stack ADT, but the order of insertion and deletion changes to FIFO. Elements are inserted at one end (called the rear) and removed from the other end (called the front). It should support the following operations:

- **enqueue():** Insert an element at the end of the queue.
- **dequeue():** Remove and return the first element of the queue, if the queue is not empty.
- **peek():** Return the element of the queue without removing it, if the queue is not empty.
- **size():** Return the number of elements in the queue.
- **isEmpty():** Return true if the queue is empty; otherwise, return false.

Advantages and Disadvantages of ADT

Abstract data types (ADTs) have several advantages and disadvantages that should be considered when deciding to use them in software development. Here are some of the main advantages and disadvantages of using ADTs:

Advantage:

The advantages are listed below:

- **Encapsulation**: ADTs provide a way to encapsulate data and operations into a single unit, making it easier to manage and modify the data structure.
- **Abstraction**: ADTs allow users to work with data structures without having to know the implementation details, which can simplify programming and reduce errors.
- **Data Structure Independence**: ADTs can be implemented using different data structures, which can make it easier to adapt to changing needs and requirements.
- **Information Hiding**: ADTs can protect the integrity of data by controlling access and preventing unauthorized modifications.
- **Modularity**: ADTs can be combined with other ADTs to form more complex data structures, which can increase flexibility and modularity in programming.

Disadvantages:

The disadvantages are listed below:

- **Overhead**: Implementing ADTs can add overhead in terms of memory and processing, which can affect performance.
- **Complexity**: ADTs can be complex to implement, especially for large and complex data structures.
- **Learning** Curve: Using ADTs requires knowledge of their implementation and usage, which can take time and effort to learn.
- **Limited Flexibility:** Some ADTs may be limited in their functionality or may not be suitable for all types of data structures.

- **Cost**: Implementing ADTs may require additional resources and investment, which can increase the cost of development.

Time and Space Complexity

Many times there are more than one ways to solve a problem with different algorithms and we need a way to compare multiple ways. Also, there are situations where we would like to know how much time and resources an algorithm might take when implemented. To measure performance of algorithms, we typically use time and space complexity analysis. The idea is to measure order of growths in terms of input size.

- Independent of the machine and its configuration, on which the algorithm is running on.
- Shows a direct correlation with the number of inputs.
- Can distinguish two algorithms clearly without ambiguity.

Time Complexity:

The time complexity of an algorithm quantifies the amount of time taken by an algorithm to run as a function of the length of the input. Note that the time to run is a function of the length of the input and not the actual execution time of the machine on which the algorithm is running on.

The valid algorithm takes a finite amount of time for execution. The time required by the algorithm to solve given problem is called ***time complexity*** of the algorithm. Time complexity is very useful measure in algorithm analysis.

It is the time needed for the completion of an algorithm. To estimate the time complexity, we need to consider the cost of each fundamental instruction and the number of times the instruction is executed.

Example 1: Addition of two scalar variables.

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <- A + B
return C
```

The addition of two scalar numbers requires one addition operation. the time complexity of this algorithm is constant, so T(n) = O(1) .

In order to calculate time complexity on an algorithm, it is assumed that a **constant time c** is taken to execute one operation, and then the total operations for an input length on **N** are calculated. Consider an example to understand the process of calculation: Suppose a problem is to find whether a pair **(X, Y)** exists in an array, A of **N** elements whose sum is **Z**. The simplest idea is to consider every pair and check if it satisfies the given condition or not.

The pseudo-code is as follows:

```
int a[n];
for(int i = 0;i < n;i++)
  cin >> a[i]

for(int i = 0;i < n;i++)
  for(int j = 0;j < n;j++)
    if(i!=j && a[i]+a[j] == z)
      return true
return false
```

Below is the implementation of the above approach:

```
// C++ program for the above approach
#include <bits/stdc++.h>
```

```
using namespace std;

// Function to find a pair in the given
// array whose sum is equal to z
bool findPair(int a[], int n, int z)
{
    // Iterate through all the pairs
    for (int i = 0; i < n; i++)
        for (int j = 0; j < n; j++)

            // Check if the sum of the pair
            // (a[i], a[j]) is equal to z
            if (i != j && a[i] + a[j] == z)
                return true;

    return false;
}

// Driver Code
int main()
{
    // Given Input
    int a[] = { 1, -2, 1, 0, 5 };
    int z = 0;
    int n = sizeof(a) / sizeof(a[0]);

    // Function Call
    if (findPair(a, n, z))
        cout << "True";
    else
        cout << "False";
    return 0;
}
```

Output

```
False
```

Assuming that each of the operations in the computer takes approximately constant time, let it be **c**. The number of lines of code executed actually depends on the value of **Z**. During analyses of the algorithm, mostly the worst-case scenario is considered, i.e., when there is no pair of elements with sum equals **Z**. In the worst case,

- **N*c** operations are required for input.
- The outer loop **i** loop runs **N** times.
- For each **i**, the inner loop **j** loop runs **N** times.

So total execution time is **N*c + N*N*c + c**. Now ignore the lower order terms since the lower order terms are relatively insignificant for large input, therefore only the highest order term is taken (without constant) which is **N*N** in this case. Different notations are used to describe the limiting behavior of a function, but since the worst case is taken so big-O notation will be used to represent the time complexity.

Hence, the time complexity is **O(N^2)** for the above algorithm. Note that the time complexity is solely based on the number of elements in array **A** i.e the input length, so if the length of the array will increase the time of execution will also increase.

Order of growth is how the time of execution depends on the length of the input. In the above example, it is clearly evident that the time of execution quadratically depends on the length of the array. Order of growth will help to compute the running time with ease.
Another Example: Let's calculate the time complexity of the below algorithm:

```
count = 0
for (int i = N; i > 0; i /= 2)
  for (int j = 0; j < i; j++)
    count++;
```

This is a tricky case. In the first look, it seems like the complexity is **O(N * log N)**. **N** for the **j's** loop and **log(N)** for **i's** loop. But it's wrong. Let's see why.
Think about how many times **count++** will run.

- When **i = N**, it will run **N** times.
- When **i = N / 2**, it will run **N / 2** times.
- When **i = N / 4**, it will run **N / 4** times.
- And so on.

The total number of times **count++** will run is **N + N/2 + N/4+...+1= 2 * N**. So the time complexity will be **O(N)**.

Space Complexity:

Definition -

Problem-solving using computer requires memory to hold temporary data or final result while the program is in execution. The amount of memory required by the algorithm to solve given problem is called **space complexity** of the algorithm.
The space complexity of an algorithm quantifies the amount of space taken by an algorithm to run as a function of the length of the input. Consider an example: Suppose a problem to find the frequency of array elements.
It is the amount of memory needed for the completion of an algorithm.
To estimate the memory requirement we need to focus on two parts:
(1) A fixed part: It is independent of the input size. It includes memory for instructions (code), constants, variables, etc.
(2) A variable part: It is dependent on the input size. It includes memory for recursion stack, referenced variables, etc.
Example : Addition of two scalar variables

```
Algorithm ADD SCALAR(A, B)
//Description: Perform arithmetic addition of two numbers
//Input: Two scalar variables A and B
//Output: variable C, which holds the addition of A and B
C <— A+B
return C
```

The addition of two scalar numbers requires one extra memory location to hold the result. Thus the space complexity of this algorithm is constant, hence S(n) = O(1).
The pseudo-code is as follows:

```
int freq[n];
int a[n];
for(int i = 0; i<n; i++)
{
  cin>>a[i];
  freq[a[i]]++;
}
```

Below is the implementation of the above approach:

```
// C++ program for the above approach
#include <bits/stdc++.h>
```

```
using namespace std;

// Function to count frequencies of array items
void countFreq(int arr[], int n)
{
    unordered_map<int, int> freq;

    // Traverse through array elements and
    // count frequencies
    for (int i = 0; i < n; i++)
        freq[arr[i]]++;

    // Traverse through map and print frequencies
    for (auto x : freq)
        cout << x.first << " " << x.second << endl;
}

// Driver Code
int main()
{
    // Given array
    int arr[] = { 10, 20, 20, 10, 10, 20, 5, 20 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Function Call
    countFreq(arr, n);
    return 0;
}
```

Output

```
5 1

20 4

10 3
```

Here two arrays of length **N**, and variable **i** are used in the algorithm so, the total space used is **N * c + N * c + 1 * c = 2N * c + c**, where **c** is a unit space taken. For many inputs, constant **c** is insignificant, and it can be said that the space complexity is **O(N)**.

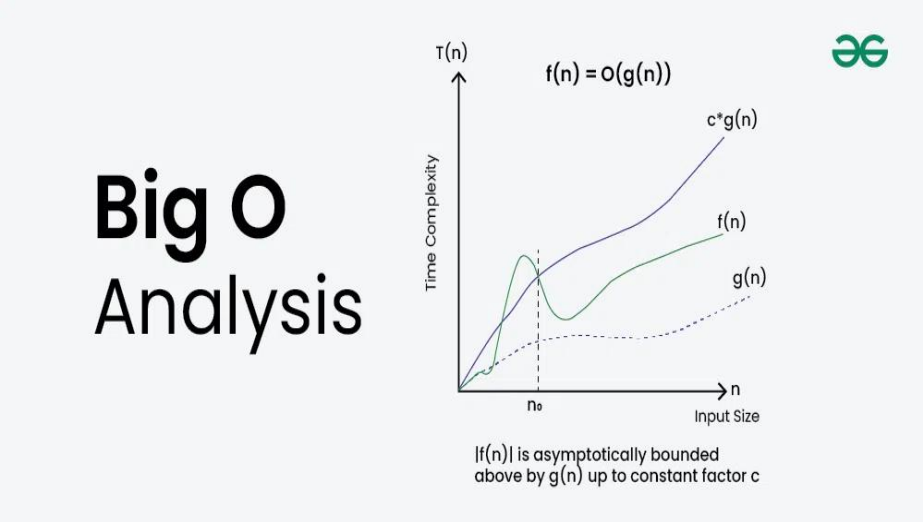
There is also **auxiliary space,** which is different from space complexity. The main difference is where space complexity quantifies the total space used by the algorithm, auxiliary space quantifies the extra space that is used in the algorithm apart from the given input. In the above example, the auxiliary space is the space used by the freq[] array because that is not part of the given input. So total auxiliary space is **N * c + c** which is **O(N)** only.

Big O Notation

Big O notation is a powerful tool used in computer science to describe the time complexity or space complexity of algorithms. **Big-O** is a way to express the **upper bound** of an algorithm's time or space complexity.

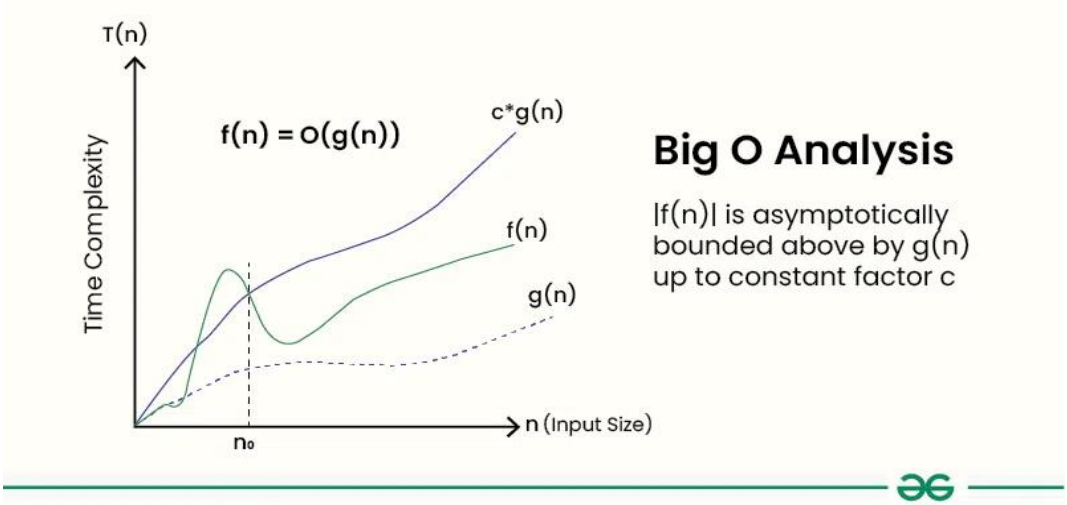
- Describes the asymptotic behavior (order of growth of time or space in terms of input size) of a function, not its exact value.
- Can be used to compare the efficiency of different algorithms or data structures.

- It provides an **upper limit** on the time taken by an algorithm in terms of the size of the input. We mainly consider the worst case scenario of the algorithm to find its time complexity in terms of Big O
- It's denoted as **O(f(n))**, where **f(n)** is a function that represents the number of operations (steps) that an algorithm performs to solve a problem of size **n**.



BIg O Definition

Given two functions **f(n)** and **g(n)**, we say that **f(n)** is **O(g(n))** if there exist constants **c > 0** and **n0 >=** 0 such that **f(n) <= c*g(n)** for all **n >= n0**.
In simpler terms, **f(n)** is **O(g(n))** if **f(n)** grows no faster than **c*g(n)** for all n >= n0 where c and n0 are constants.



Importance of Big O Notation

Big O notation is a mathematical notation used to find an upper bound on time taken by an algorithm or data structure. It provides a way to compare the performance of different algorithms and data structures, and to predict how they will behave as the input size increases.
Big O notation is important for several reasons:

- Big O Notation is important because it helps analyze the efficiency of algorithms.
- It provides a way to describe how the **runtime** or **space requirements** of an algorithm grow as the input size increases.
- Allows programmers to compare different algorithms and choose the most efficient one for a specific problem.
- Helps in understanding the scalability of algorithms and predicting how they will perform as the input size grows.
- Enables developers to optimize code and improve overall performance.

Properties of Big O Notation

Below are some important Properties of Big O Notation:

1. Reflexivity

For any function f(n), f(n) = O(f(n)).

Example:

f(n) = n^2, then f(n) = O(n^2).

2. Transitivity

If f(n) = O(g(n)) and g(n) = O(h(n)), then f(n) = O(h(n)).

Example:

If f(n) = n^2, g(n) = n^3, and h(n) = n^4, then f(n) = O(g(n)) and g(n) = O(h(n)).
Therefore, by transitivity, f(n) = O(h(n)).

3. Constant Factor

For any constant c > 0 and functions f(n) and g(n), if f(n) = O(g(n)), then cf(n) = O(g(n)).

Example:

f(n) = n, g(n) = n^2. Then f(n) = O(g(n)). Therefore, 2f(n) = O(g(n)).

4. Sum Rule

If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) + h(n) = O(max(g(n), k(n)) When combining complexities, only the largest term dominates.

Example:

f(n) = n^2, h(n) = n^3. Then , f(n) + h(n) = O(max(n^2 + n^3) = O (n^3)

5. Product Rule

If f(n) = O(g(n)) and h(n) = O(k(n)), then f(n) * h(n) = O(g(n) * k(n)).

Example:

*f(n) = n, g(n) = n^2, h(n) = n^3, k(n) = n^4. Then f(n) = O(g(n)) and h(n) = O(k(n)). Therefore, f(n) * h(n) = O(g(n) * k(n)) = O(n^6).*

6. Composition Rule

If f(n) = O(g(n)) and g(n) = O(h(n)), then f(g(n)) = O(h(n)).

Example:

f(n) = n^2, g(n) = n, h(n) = n^3. Then f(n) = O(g(n)) and g(n) = O(h(n)). Therefore, f(g(n)) = O(h(n)) = O(n^3).

Common Big-O Notations

Big-O notation is a way to measure the time and space complexity of an algorithm. It describes the upper bound of the complexity in the worst-case scenario. Let's look into the different types of time complexities:

1. Linear Time Complexity: Big O(n) Complexity

Linear time complexity means that the running time of an algorithm grows linearly with the size of the input.

For example, consider an algorithm that traverses through an array to find a specific element:

```
bool findElement(int arr[], int n, int key)
{
```

```
    for (int i = 0; i < n; i++) {
        if (arr[i] == key) {
            return true;
        }
    }
    return false;
}
```

2. Logarithmic Time Complexity: Big O(log n) Complexity

Logarithmic time complexity means that the running time of an algorithm is proportional to the logarithm of the input size.

For example, a binary search algorithm has a logarithmic time complexity:

```
int binarySearch(int arr[], int l, int r, int x)
{
    if (r >= l) {
        int mid = l + (r - l) / 2;
        if (arr[mid] == x)
            return mid;
        if (arr[mid] > x)
            return binarySearch(arr, l, mid - 1, x);
        return binarySearch(arr, mid + 1, r, x);
    }
    return -1;
}
```

3. Quadratic Time Complexity: Big O(n^2) Complexity

Quadratic time complexity means that the running time of an algorithm is proportional to the square of the input size.

For example, a simple bubble sort algorithm has a quadratic time complexity:

```
void bubbleSort(int arr[], int n)
{
    for (int i = 0; i < n - 1; i++) {
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(&arr[j], &arr[j + 1]);
            }
        }
    }
}
```

4. Cubic Time Complexity: Big O(n^3) Complexity

Cubic time complexity means that the running time of an algorithm is proportional to the cube of the input size.

For example, a naive matrix multiplication algorithm has a cubic time complexity:

```
void multiply(int mat1[][N], int mat2[][N], int res[][N])
{
    for (int i = 0; i < N; i++) {
        for (int j = 0; j < N; j++) {
            res[i][j] = 0;
            for (int k = 0; k < N; k++)
                res[i][j] += mat1[i][k] * mat2[k][j];
        }
    }
```

```
}
```

5. Polynomial Time Complexity: Big O(n^k) Complexity

Polynomial time complexity refers to the time complexity of an algorithm that can be expressed as a polynomial function of the input size **n**. In Big **O** notation, an algorithm is said to have polynomial time complexity if its time complexity is **O(n^k)**, where **k** is a constant and represents the degree of the polynomial.

Algorithms with polynomial time complexity are generally considered efficient, as the running time grows at a reasonable rate as the input size increases. Common examples of algorithms with polynomial time complexity include **linear time complexity O(n)**, **quadratic time complexity O(n^2)**, and **cubic time complexity O(n^3)**.

6. Exponential Time Complexity: Big O(2^n) Complexity

Exponential time complexity means that the running time of an algorithm doubles with each addition to the input data set.

For example, the problem of generating all subsets of a set is of exponential time complexity:

```
void generateSubsets(int arr[], int n)
{
   for (int i = 0; i < (1 << n); i++) {
      for (int j = 0; j < n; j++) {
         if (i & (1 << j)) {
            cout << arr[j] << " ";
         }
      }
      cout << endl;
   }
}
```

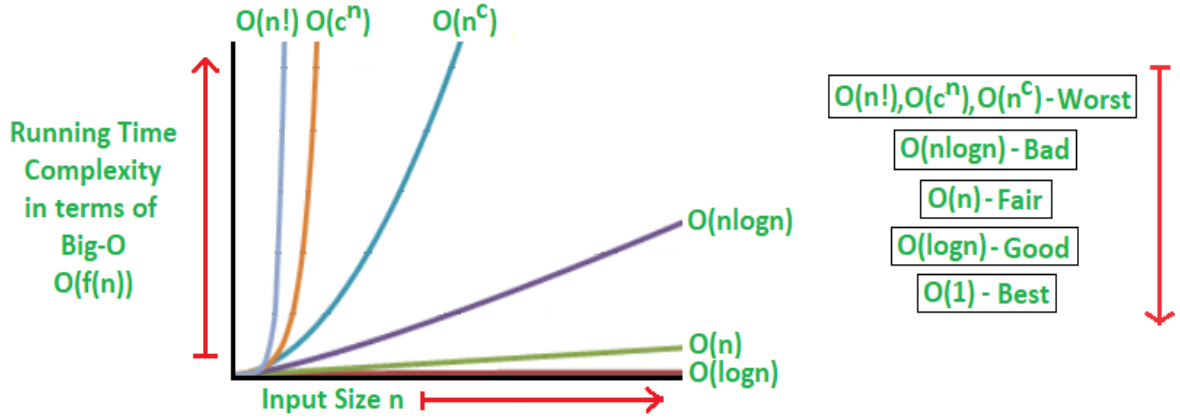
7. Factorial Time Complexity: Big O(n!) Complexity

Factorial time complexity means that the running time of an algorithm grows factorially with the size of the input. This is often seen in algorithms that generate all permutations of a set of data.

Here's an example of a factorial time complexity algorithm, which generates all permutations of an array:

```
void permute(int* a, int l, int r)
{
   if (l == r) {
      for (int i = 0; i <= r; i++) {
         cout << a[i] << " ";
      }
      cout << endl;
   }
   else {
      for (int i = l; i <= r; i++) {
         swap(a[l], a[i]);
         permute(a, l + 1, r);
         swap(a[l], a[i]); // backtrack
      }
   }
}
```

If we plot the most common Big O notation examples, we would have graph like this:



Array

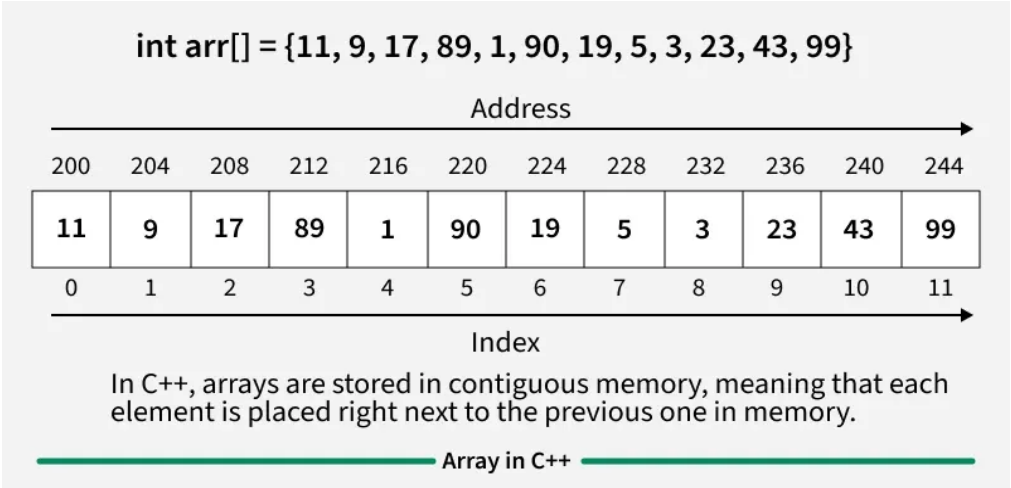
Array is a collection of items of the same variable type that are stored at contiguous memory locations. It is one of the most popular and simple data structures used in programming.

Basic terminologies of Array

- **Array Index:** In an array, elements are identified by their indexes. Array index starts from 0.
- **Array element:** Elements are items stored in an array and can be accessed by their index.
- **Array Length:** The length of an array is determined by the number of elements it can contain.

Memory representation of Array

In an array, all the elements are stored in contiguous memory locations. So, if we initialize an array, the elements will be allocated sequentially in memory. This allows for efficient access and manipulation of elements.



Declaration of Array

Arrays can be declared in various ways in different languages. For better illustration, below are some language-specific array declarations:

```
// This array will store integer type element
int arr[5];

// This array will store char type element
char arr[10];

// This array will store float type element
float arr[20];
```

Initialization of Array

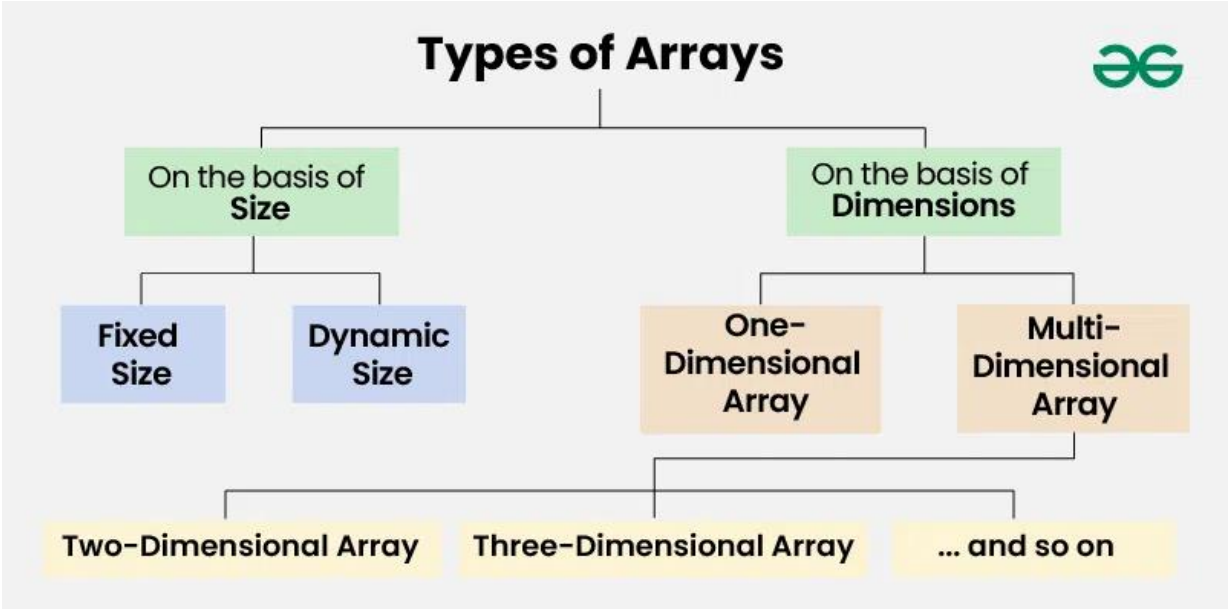
Arrays can be initialized in different ways in different languages. Below are some language-specific array initializations:

```
int arr[] = { 1, 2, 3, 4, 5 };
char arr[5] = { 'a', 'b', 'c', 'd', 'e' };
float arr[10] = { 1.4, 2.0, 24, 5.0, 0.0 };
```

Types of Arrays

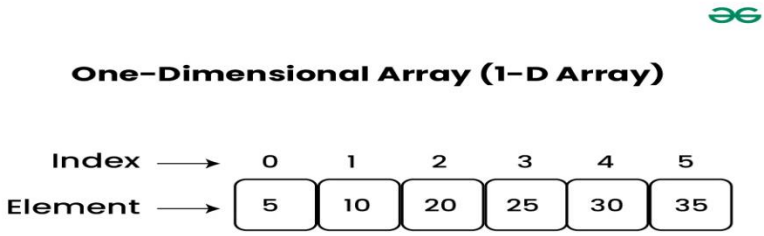
Arrays can be classified in two ways:

- On the basis of Size
- On the basis of Dimensions



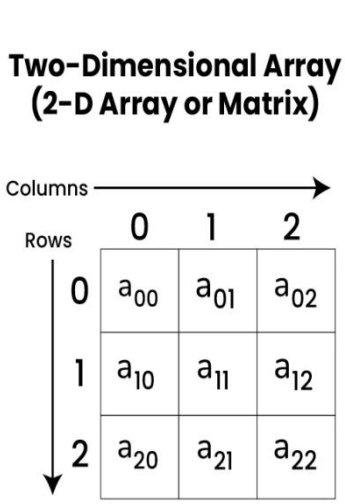
Types of Arrays on the basis of Dimensions

1. One-dimensional Array(1-D Array): You can imagine a 1d array as a row, where elements are stored one after another.

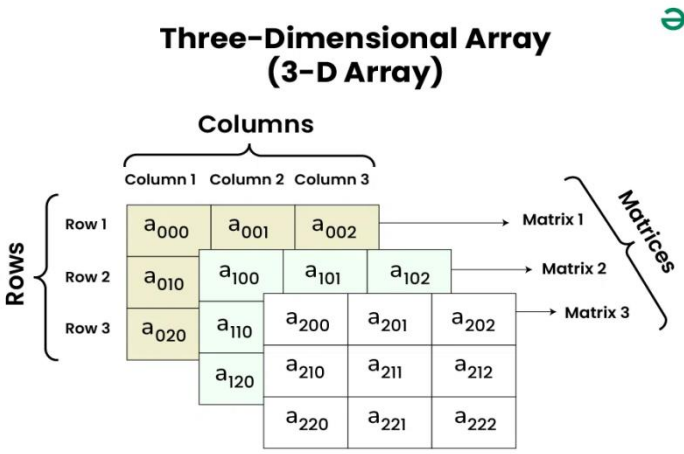


2. Multi-dimensional Array: A multi-dimensional array is an array with more than one dimension. We can use multidimensional array to store complex data in the form of tables, etc. We can have 2-D arrays, 3-D arrays, 4-D arrays and so on.

- **Two-Dimensional Array(2-D Array or Matrix):** 2-D Multidimensional arrays can be considered as an array of arrays or as a matrix consisting of rows and columns.



- **Three-Dimensional Array(3-D Array):** A 3-D Multidimensional array contains three dimensions, so it can be considered an array of two-dimensional arrays.



Operations on Array

1. Array Traversal

Array traversal refers to the process of accessing and processing each element of an array sequentially. This is one of the most fundamental operations in programming, as arrays are widely used data structures for storing multiple elements in a single variable.

How Array Traversal Works?

When an array is created, it occupies a contiguous block of memory where elements are stored in an indexed manner. Each element can be accessed using its index, which starts from 0 in most programming languages.

For example, consider an array containing five integers:

arr = [10, 20, 30, 40, 50]

Here:

- The first element (10) is at **index 0**.
- The second element (20) is at **index 1**.
- The last element (50) is at **index 4**.

Array traversal means accessing each element from start to end (or sometimes in reverse order), usually by using a loop.

Types of Array Traversal

Array traversal can be done in multiple ways based on the requirement:

1. **Sequential (Linear) Traversal**
   - This is the most common way of traversing an array.
   - It involves iterating through the array one element at a time from the first index to the last.
   - Used for printing elements, searching, or performing calculations (such as sum or average).
2. **Reverse Traversal**
   - Instead of starting from index 0, the traversal begins from the last element and moves towards the first.
   - This is useful in cases where we need to process elements from the end.

2. Insertion in Array

Insertion in an array refers to the process of adding a new element at a specific position while maintaining the order of the existing elements. Since arrays have a fixed size in static implementations, inserting an element often requires shifting existing elements to make space.

How Insertion Works in an Array?

Arrays are stored in contiguous memory locations, meaning elements are arranged in a sequential block. When inserting a new element, the following happens:

1. **Identify the Position**: Determine where the new element should be inserted.
2. **Shift Elements**: Move the existing elements one position forward to create space for the new element.
3. **Insert the New Element**: Place the new value in the correct position.
4. **Update the Size (if applicable)**: If the array is dynamic, its size is increased.

For example, if we have the array:

arr = [10, 20, 30, 40, 50]

and we want to insert 25 at index 2, the new array will be:

arr = [10, 20, 25, 30, 40, 50]

Here, elements 30, 40, and 50 have shifted right to make space.

Types of Insertion

1. Insertion at the Beginning (Index 0)

- Every element must shift one position right.
- This is the least efficient case for large arrays as it affects all elements.

2. Insertion at a Specific Index

- Elements after the index shift right.
- If the index is in the middle, half of the array moves.

3. Insertion at the End

- The simplest case since no shifting is required.
- Used in dynamic arrays where size increases automatically (e.g., Python lists, Java ArrayList).

3. Deletion in Array

Deletion in an array refers to the process of removing an element from a specific position while maintaining the order of the remaining elements. Unlike linked lists, where deletion is efficient, removing an element from an array requires shifting elements to fill the gap.

How Deletion Works in an Array?

Since arrays have contiguous memory allocation, deleting an element does not reduce the allocated memory size. Instead, it involves:

1. **Identify the Position**: Find the index of the element to be deleted.
2. **Shift Elements**: Move the elements after the deleted element one position to the left.
3. **Update the Size (if applicable)**: If using a dynamic array, the size might be reduced.

For example, consider the array:

arr = [10, 20, 30, 40, 50]

If we delete the element 30 (index 2), the new array will be:

arr = [10, 20, 40, 50]

Here, elements 40 and 50 shifted left to fill the gap.

Types of Deletion

1. Deletion at the Beginning (Index 0)

- Every element shifts left by one position.
- This is the most expensive case as it affects all elements.

2. Deletion at a Specific Index

- Only elements after the index shift left.
- If the index is in the middle, half of the array moves.

3. Deletion at the End

- The simplest case since no shifting is required.
- The size of the array is reduced (in dynamic arrays).

4. Searching in Array

Searching in an array refers to the process of finding a specific element in a given list of elements. The goal is to determine whether the element exists in the array and, if so, find its index (position).

Searching is a fundamental operation in programming, as it is used in data retrieval, filtering, and processing.

Types of Searching in an Array

There are two main types of searching techniques in an array:

1. Linear Search (Sequential Search)

- This is the simplest search algorithm.
- It traverses the array one element at a time and compares each element with the target value.
- If a match is found, it returns the index of the element.
- If the element is not found, the search continues until the end of the array.

Example:

Consider an array:

arr = [10, 20, 30, 40, 50]

If we search for 30, the algorithm will:

1. Compare 10 with 30 → No match.
2. Compare 20 with 30 → No match.
3. Compare 30 with 30 → **Match found at index 2.**

2. Binary Search (Efficient Search for Sorted Arrays)

- Works only on sorted arrays (in increasing or decreasing order).
- Uses a divide and conquer approach.
- It repeatedly divides the search space in half until the target element is found.

How Binary Search Works?

1. Find the middle element of the array.
2. If the target is equal to the middle element, return its index.
3. If the target is less than the middle element, search the left half.
4. If the target is greater than the middle element, search the right half.
5. Repeat until the element is found or the search space is empty.

Example:

Consider a sorted array:

arr = [10, 20, 30, 40, 50]

If we search for 30:

1. Middle element = 30 → Match found!

2. The search ends in just one step, making it much faster than linear search.

Strings

Strings are sequences of characters. The differences between a character array and a string are, a string is terminated with a special character **'\0'** and strings are typically immutable in most of the programming languages like Java, Python and JavaScript. Below are some examples of strings:
"geeks" , "for", "geeks", "GeeksforGeeks", "Geeks for Geeks", "123Geeks", "@123 Geeks"

Below is the representation of strings in various languages:

```
// C++ program to demonstrate String
// using Standard String representation

#include <iostream>
#include <string>
using namespace std;

int main()
{

    // Declare and initialize the string
    string str1 = "Welcome to GeeksforGeeks!";

    // Initialization by raw string
    string str2("A Computer Science Portal");

    // Print string
    cout << str1 << endl << str2;

    return 0;
}
```

Are Strings Mutable in Different Languages?

- In C/C++, string literals (assigned to pointers) are immutable.
- In C++, string objects are mutable.
- In Python, Java and JavaScript, strings are immutable.

```
#include <iostream>
using namespace std;

int main() {
    const char* str = "Hello, world!";
    str[0] = 'h';  // Error : Assignment to read only
    cout << str;
    return 0;
}
```

General Operations performed on String

Here we are providing you with some must-know concepts of string:

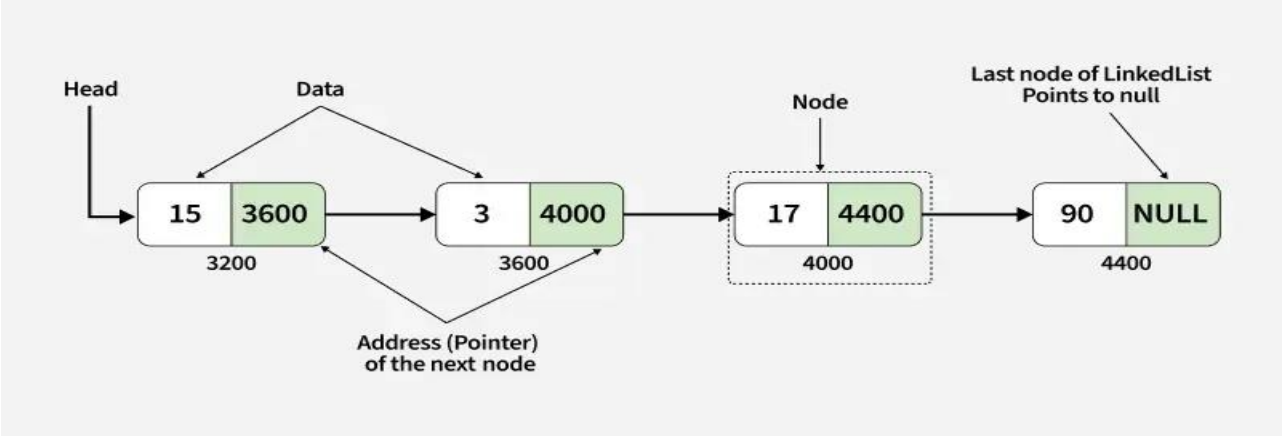
- **Length of String :** The length of a string refers to the total number of characters present in it, including letters, digits, spaces, and special characters. It is a fundamental property of strings in any programming language and is often used in various operations such as validation, manipulation, and comparison.

- **Search a Character :** Searching for a character in a string means finding the position where a specific character appears. If the character is present multiple times, you might need to find its first occurrence, last occurrence, or all occurrences.
- **Check for Substring :** Checking for a substring means determining whether a smaller sequence of characters exists within a larger string. A substring is a continuous part of a string, and checking for its presence is a common operation in text processing, search algorithms, and data validation.
- **Insert a Character :** Inserting a character into a string means adding a new character at a specific position while maintaining the original order of other characters. Since strings are immutable in many programming languages, inserting a character usually involves creating a new modified string with the desired character placed at the specified position.
- **Delete a Character :** Deleting a character from a string means removing a specific character at a given position while keeping the remaining characters intact. Since strings are immutable in many programming languages, this operation usually involves creating a new string without the specified character.
- **Check for Same Strings :** Checking if two strings are the same means comparing them character by character to determine if they are identical in terms of length, order, and content. If every character in one string matches exactly with the corresponding character in another string, they are considered the same.
- **String Concatenation :** String concatenation is the process of joining two or more strings together to form a single string. This is useful in text processing, formatting messages, constructing file paths, or dynamically creating content.
- **Reverse a String :** Reversing a string means arranging its characters in the opposite order while keeping their original positions intact in the reversed sequence. This operation is commonly used in text manipulation, data encryption, and algorithm challenges.
- **Rotate a String** Rotating a string means shifting its characters to the left or right by a specified number of positions while maintaining the order of the remaining characters. The characters that move past the boundary wrap around to the other side.
- **Check for Palindrome :** Checking for a palindrome means determining whether a string reads the same forward and backward. A palindrome remains unchanged when reversed, making it a useful concept in text processing, algorithms, and number theory.

Unit II

Linked List

A **linked list** is a fundamental data structure in computer science. It mainly allows efficient **insertion** and **deletion** operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque. Here's the comparison of Linked List vs Arrays



Linked List:

- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

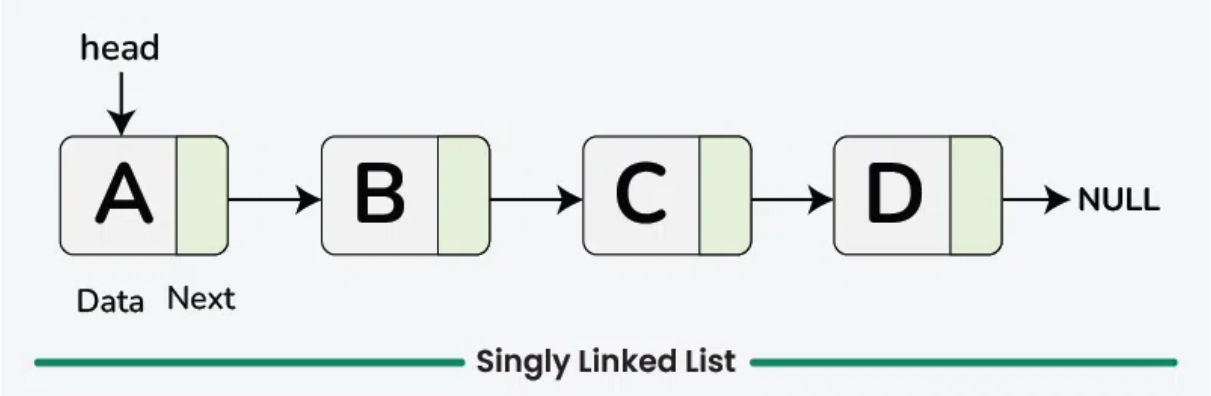
- **Data Structure:** Contiguous
- **Memory Allocation:** Typically allocated to the whole array
- **Insertion/Deletion:** Inefficient
- **Access:** Random

Singly Linked List

A **singly linked list** is a fundamental data structure, it consists of **nodes** where each node contains a **data** field and a **reference** to the next node in the linked list. The next of the last node is **null**, indicating the end of the list. Linked Lists support efficient insertion and deletion operations.

Understanding Node Structure

In a singly linked list, each node consists of two parts: data and a pointer to the next node. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.




```
// Definition of a Node in a singly linked list
struct Node {
```

```
    // Data part of the node
    int data;

    // Pointer to the next node in the list
    Node* next;

    // Constructor to initialize the node with data
    Node(int data)
    {
        this->data = data;
        this->next = nullptr;
    }
};
```

In this example, the Node class contains an integer data field (**data**) to store the information and a pointer to another Node (**next**) to establish the link to the next node in the list.

1. Traversal of Singly Linked List

Traversal in a linked list means visiting each node and performing operations like printing or processing data.

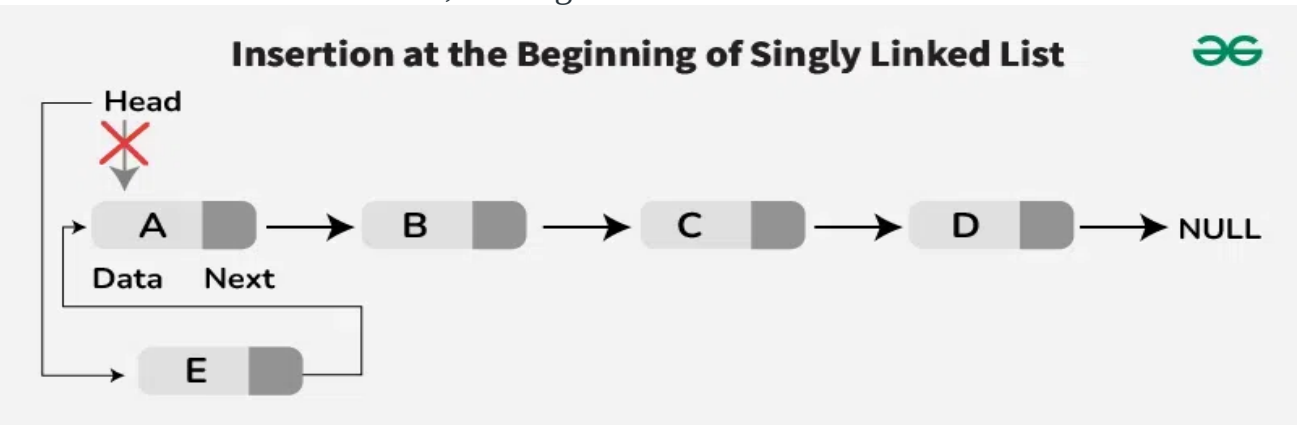
Step-by-step approach:

1. **Initialize a pointer** (current) to the head of the list.
2. **Loop through the list** using a while loop until current becomes NULL.
3. **Process each node** (e.g., print its data).
4. **Move to the next node** by updating current = current->next.

2. Insertion in Singly Linked List

Insertion is a fundamental operation in linked lists that involves adding a new node to the list. There are several scenarios for insertion:

a. Insertion at the Beginning of Singly Linked List: Insertion at the beginning involves adding a new node before the current head, making it the new head.

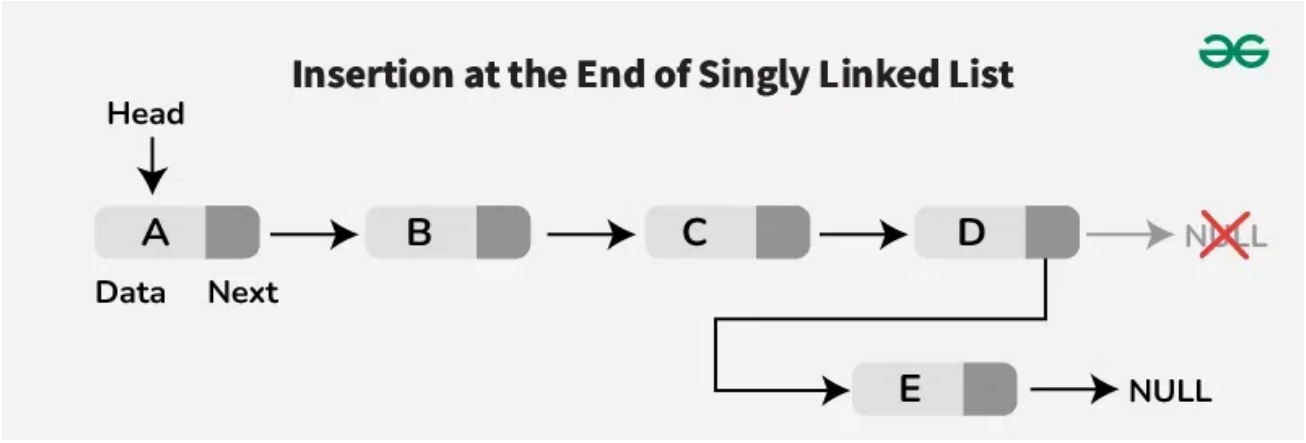


Insert a Node at the Front/Beginning of Linked List

Step-by-step approach:

- Create a new node with the given value.
- Set the **next** pointer of the new node to the current head.
- Move the head to point to the new node.
- Return the new head of the linked list.

b. Insertion at the End of Singly Linked List: To insert a node at the end of the list, traverse the list until the last node is reached, and then link the new node to the current last node

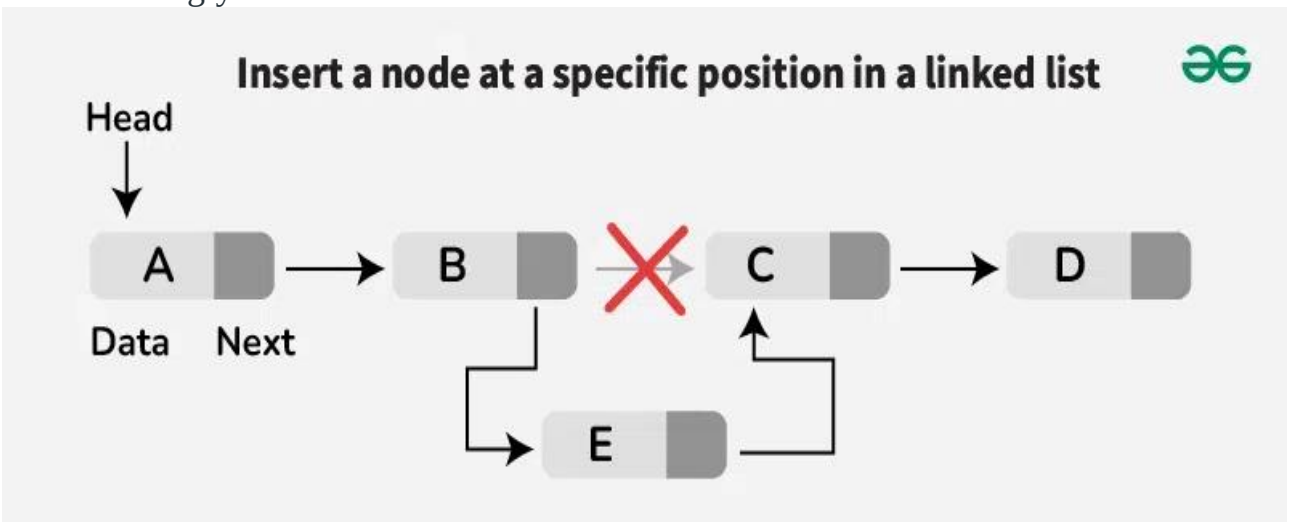


Insertion at end of Linked List

Step-by-step approach:

- Create a new node with the given value.
- Check if the list is empty:
    - If it is, make the new node the head and return.
- Traverse the list until the last node is reached.
- Link the new node to the current last node by setting the last node's next pointer to the new node.

c. Insertion at a Specific Position of the Singly Linked List: To insert a node at a specific position, traverse the list to the desired position, link the new node to the next node, and update the links accordingly.



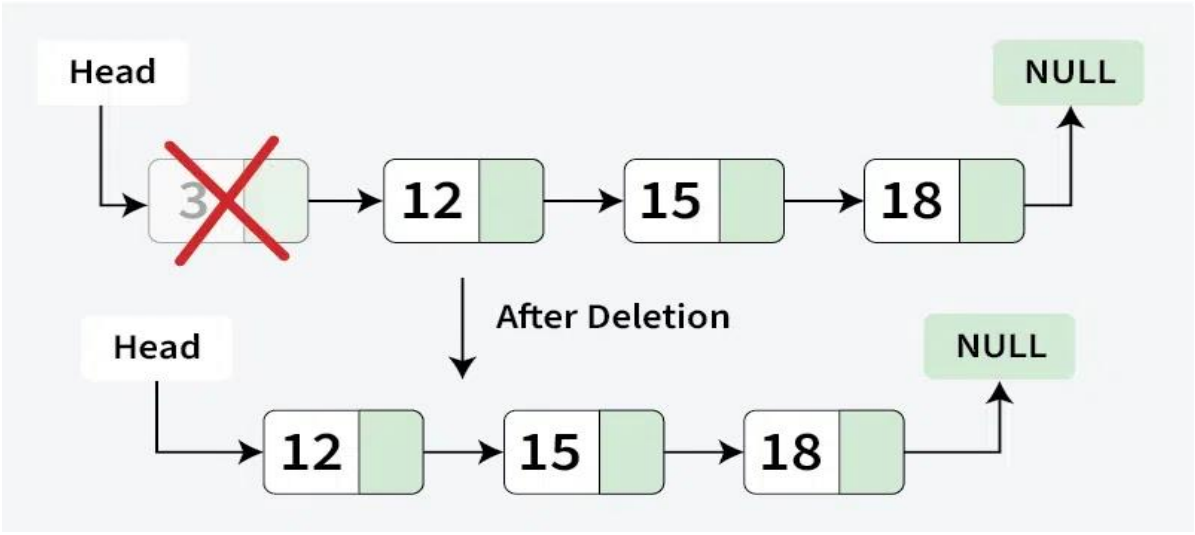
Step-by-step approach:

- **Create a new node** and assign it a value.
- **If inserting at the beginning** (position = 1):
    - Point the new node's next to the current head.
    - Update the head to the new node.
    - **Return** (Insertion done).
- **Otherwise, traverse the list**:
    - Start from the head and move to the (position - 1)th node (just before the desired position).
    - If the position is beyond the list length, **return an error or append at the end**.
- **Insert the new node**:
    - Point the new node's next to the next node of the current position.
    - Update the previous node's next to the new node.
- **Return the updated list.**

5. Deletion in Singly Linked List

Deletion involves removing a node from the linked list. Similar to insertion, there are different scenarios for deletion:

a. Deletion at the Beginning of Singly Linked List: To delete the first node, update the head to point to the second node in the list.

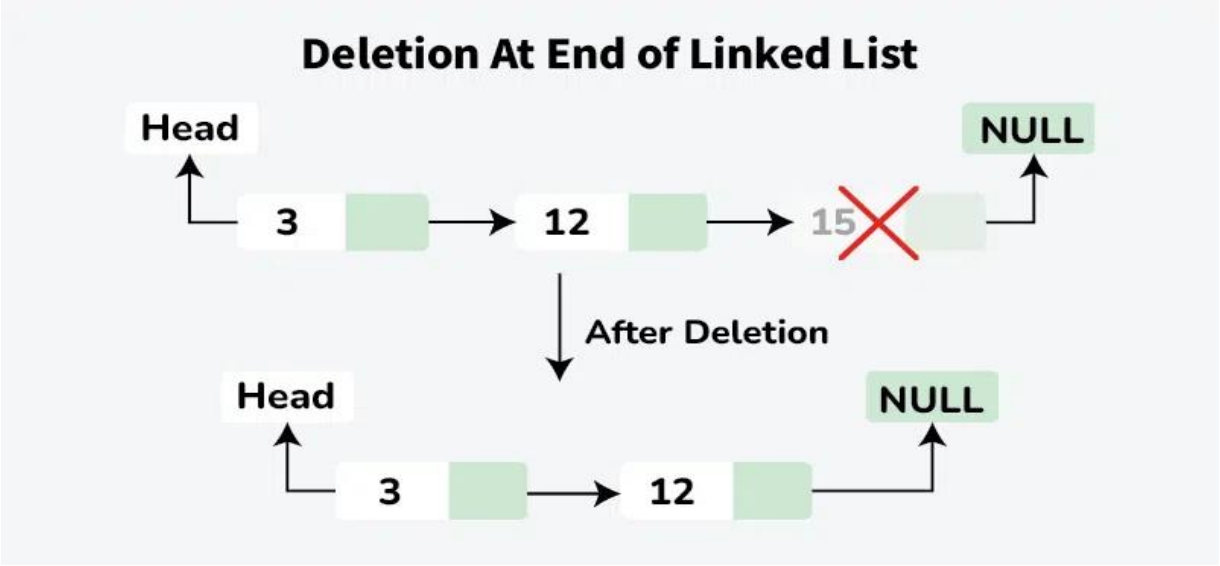


Deletion at beginning in a Linked List

Steps-by-step approach:

- Check if the head is **NULL**.
    - If it is, return **NULL** (the list is empty).
- Store the current head node in a temporary variable **temp**.
- Move the head pointer to the next node.
- Delete the temporary node.
- Return the new head of the linked list.

b. Deletion at the End of Singly Linked List: To delete the last node, traverse the list until the second-to-last node and update its next field to None.

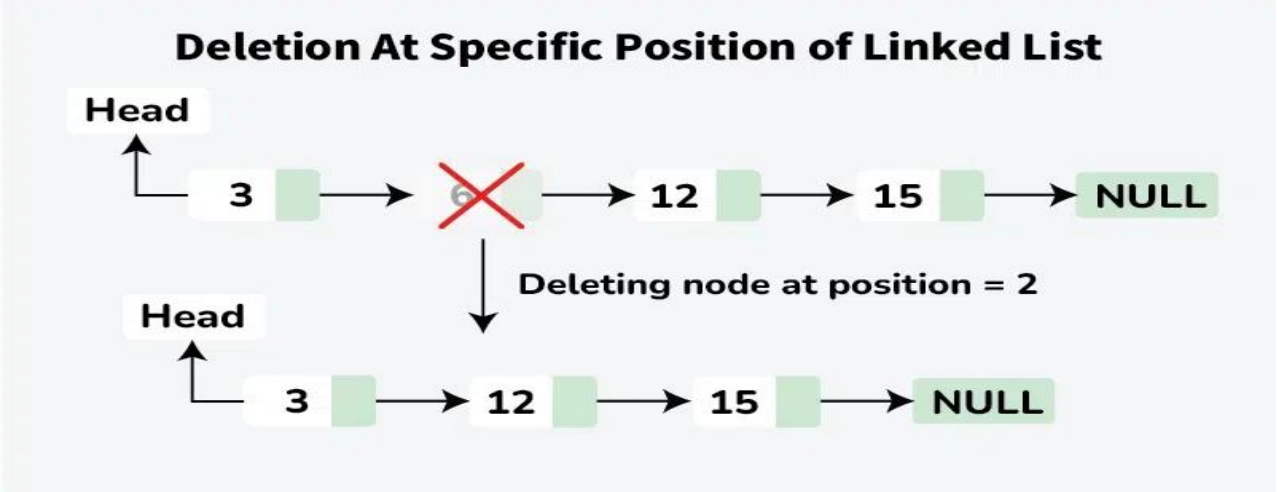


Deletion at the end of linked list

Step-by-step approach:

- Check if the head is **NULL**.
    - If it is, return NULL (the list is empty).
- Check if the head's **next** is **NULL** (only one node in the list).
    - If true, delete the head and return **NULL**.
- Traverse the list to find the second last node (**second_last**).
- Delete the last node (the node after **second_last**).
- Set the **next** pointer of the second last node to **NULL**.
- Return the head of the linked list.

c. Deletion at a Specific Position of Singly Linked List: To delete a node at a specific position, traverse the list to the desired position, update the links to bypass the node to be deleted.



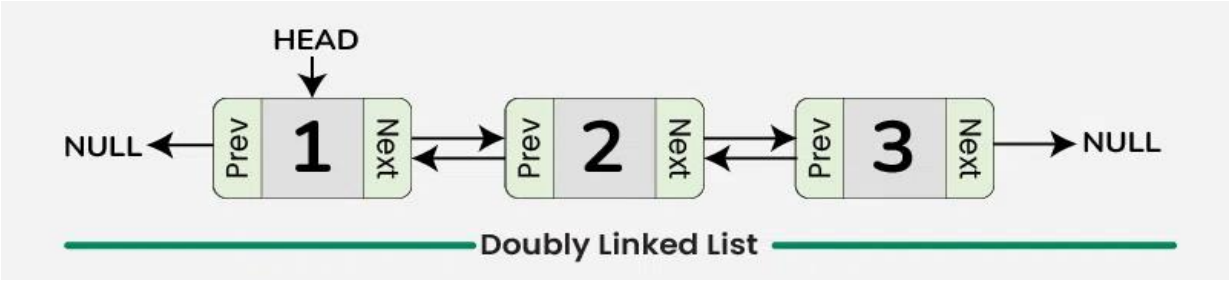
Delete a Linked List node at a given position

Step-by-step approach:

- Check if the list is empty or the position is invalid, return if so.
- If the head needs to be deleted, update the head and delete the node.
- Traverse to the node before the position to be deleted.
- If the position is out of range, return.
- Store the node to be deleted.
- Update the links to bypass the node.
- Delete the stored node.

Doubly Linked List

A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous node and a pointer to the next node. This allows for quick and easy insertion and deletion of nodes from the list, as well as efficient traversal of the list in both directions.

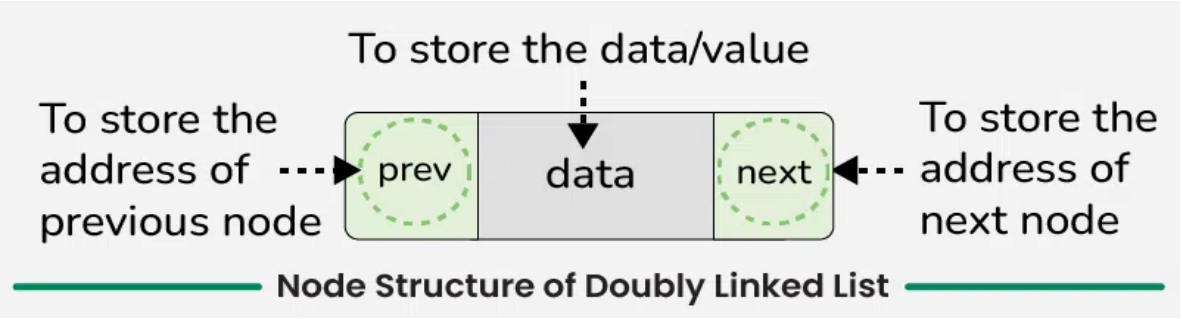


Doubly Linked List

Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)



Node Structure of Doubly Linked List

Node Definition

Here is how a node in a Doubly Linked List is typically represented:

```
struct Node {

    // To store the Value or data.
    int data;

    // Pointer to point the Previous Element
    Node* prev;

    // Pointer to point the Next Element
    Node* next;

    // Constructor
    Node(int d) {
        data = d;
        prev = next = nullptr;
    }
};
```

Each node in a **Doubly Linked List** contains the **data** it holds, a pointer to the **next** node in the list, and a pointer to the **previous** node in the list. By linking these nodes together through the **next** and **prev** pointers, we can traverse the list in both directions (forward and backward), which is a key feature of a Doubly Linked List.

Advantages of Doubly Linked List

- **Efficient traversal in both directions:** Doubly linked lists allow for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.
- **Easy insertion and deletion of nodes:** The presence of pointers to both the previous and next nodes makes it easy to insert or delete nodes from the list, without having to traverse the entire list.
- **Can be used to implement a stack or queue:** Doubly linked lists can be used to implement both stacks and queues, which are common data structures used in programming.

Disadvantages of Doubly Linked List

- **More complex than singly linked lists:** Doubly linked lists are more complex than singly linked lists, as they require additional pointers for each node.
- **More memory overhead:** Doubly linked lists require more memory overhead than singly linked lists, as each node stores two pointers instead of one.

Applications of Doubly Linked List

- Implementation of undo and redo functionality in text editors.
- Cache implementation where quick insertion and deletion of elements are required.

- Browser history management to navigate back and forth between visited pages.
- Music player applications to manage playlists and navigate through songs efficiently.
- Implementing data structures like Deque (double-ended queue) for efficient insertion and deletion at both ends.

Circular Linked List

A circular linked list is a data structure where the last node points back to the first node, forming a closed loop.

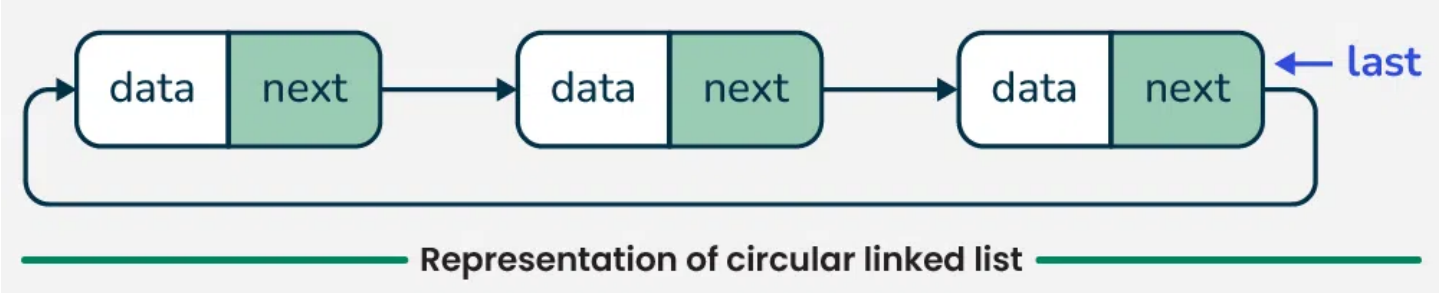
- **Structure:** All nodes are connected in a circle, enabling continuous traversal without encountering NULL.
- Difference from Regular Linked List: In a regular linked list, the last node points to NULL, whereas in a circular linked list, it points to the first node.
- **Uses:** Ideal for tasks like scheduling and managing playlists, where smooth and repeated.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked lists and doubly linked lists. So, circular linked lists are basically of two types:

1. Circular Singly Linked List

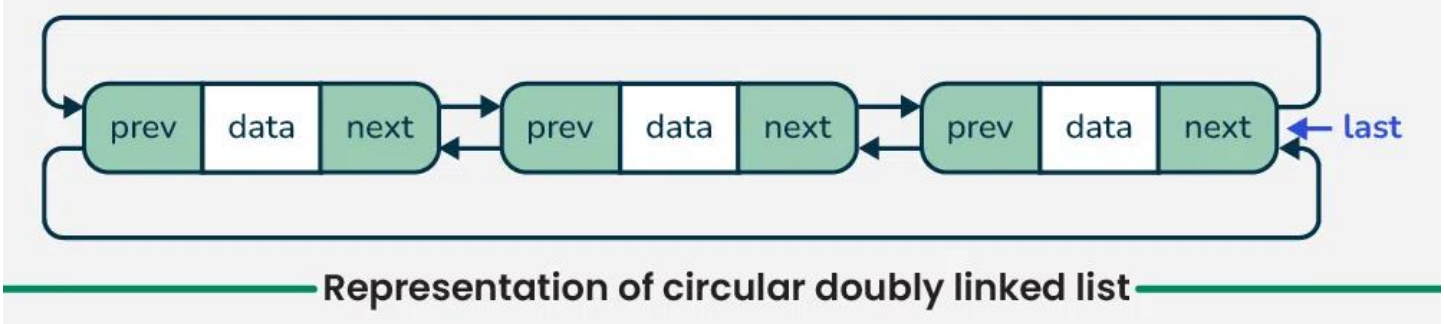
In Circular Singly Linked List, each node has just one pointer called the "next" pointer. The next pointer of the last node points back to the first node and this results in forming a circle. In this type of Linked list, we can only move through the list in one direction.



Representation of Circular Singly Linked List

2. Circular Doubly Linked List:

In circular doubly linked list, each node has two pointers prev and next, similar to doubly linked list. The prev pointer points to the previous node and the next points to the next node. Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.

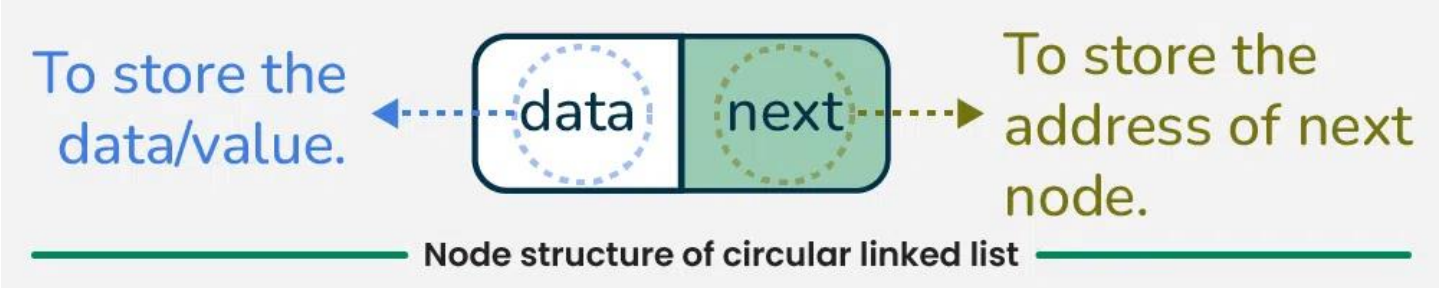


Representation of Circular Doubly Linked List

Note: IHere, we will use the singly linked list to explain the working of circular linked lists.

Representation of a Circular Singly Linked List

Let's take a look on the structure of a circular linked list.



Representation of a Circular Singly Linked List

Create/Declare a Node of Circular Linked List

Syntax to Declare a Circular Linked List in Different Languages:

```
// Node structure
struct Node {
    int data;
    Node* next;

    Node(int value){
        data = value;
        next = nullptr;
    }
};
```

In the code above, each node has **data** and a **pointer** to the next node. When we create multiple nodes for a circular linked list, we only need to connect the last node back to the first one.

Applications of Circular Linked Lists

- It is used for time-sharing among different users, typically through a **Round-Robin scheduling mechanism.**
- In multiplayer games, a circular linked list can be used to switch between players. After the last player's turn, the list cycles back to the first player.
- Circular linked lists are often used in buffering applications, such as streaming data, where data is continuously produced and consumed.
- In media players, circular linked lists can manage playlists, this allowing users to loop through songs continuously.
- Browsers use circular linked lists to manage the cache. This allows you to navigate back through your browsing history efficiently by pressing the BACK button.

Advantages of Linked Lists (or Most Common Use Cases):

- Linked Lists are mostly used because of their effective insertion and deletion. We only need to change few pointers (or references) to insert (or delete) an item in the middle
- Insertion and deletion at any point in a linked list take O(1) time. Whereas in an array data structure, insertion / deletion in the middle takes O(n) time.
- This data structure is simple and can be also used to implement a stack, queues, and other abstract data structures.
- Implementation of Queue and Deque data structures : Simple array implementation is not efficient at all. We must use circular array to efficiently implement which is complex. But with linked list, it is easy and straightforward. That is why most of the language libraries use Linked List internally to implement these data structures..
- Linked List might turn out to be more space efficient compare to arrays in cases where we cannot guess the number of elements in advance. In case of arrays, the whole memory for items is allocated together. Even with dynamic sized arrays like vector in C++ or list in

Python or ArrayList in Java. the internal working involves de-allocation of whole memory and allocation of a bigger chunk when insertions happen beyond the current capacity.

Applications of Linked Lists:

- Linked Lists can be used to implement stacks, queue, deque, sparse matrices and adjacency list representation of graphs.
- Dynamic memory allocation in operating systems and compilers (linked list of free blocks).
- Manipulation of polynomials
- Arithmetic operations on long integers.
- In operating systems, they can be used in Memory management, process scheduling (for example circular linked list for round robin scheduling) and file system.
- Algorithms that need to frequently insert or delete items from large collections of data.
- LRU cache, which uses a doubly linked list to keep track of the most recently used items in a cache.

Applications of Linked Lists in real world:

- The list of songs in the music player are linked to the previous and next songs.
- In a web browser, previous and next web page URLs can be linked through the previous and next buttons (Doubly Linked List)
- In image viewer, the previous and next images can be linked with the help of the previous and next buttons (Doubly Linked List)
- Circular Linked Lists can be used to implement things in round manner where we go to every element one by one.
- Linked List are preferred over arrays for implementations of Queue and Deque data structures because of fast deletions (or insertions) from the front of the linked lists.

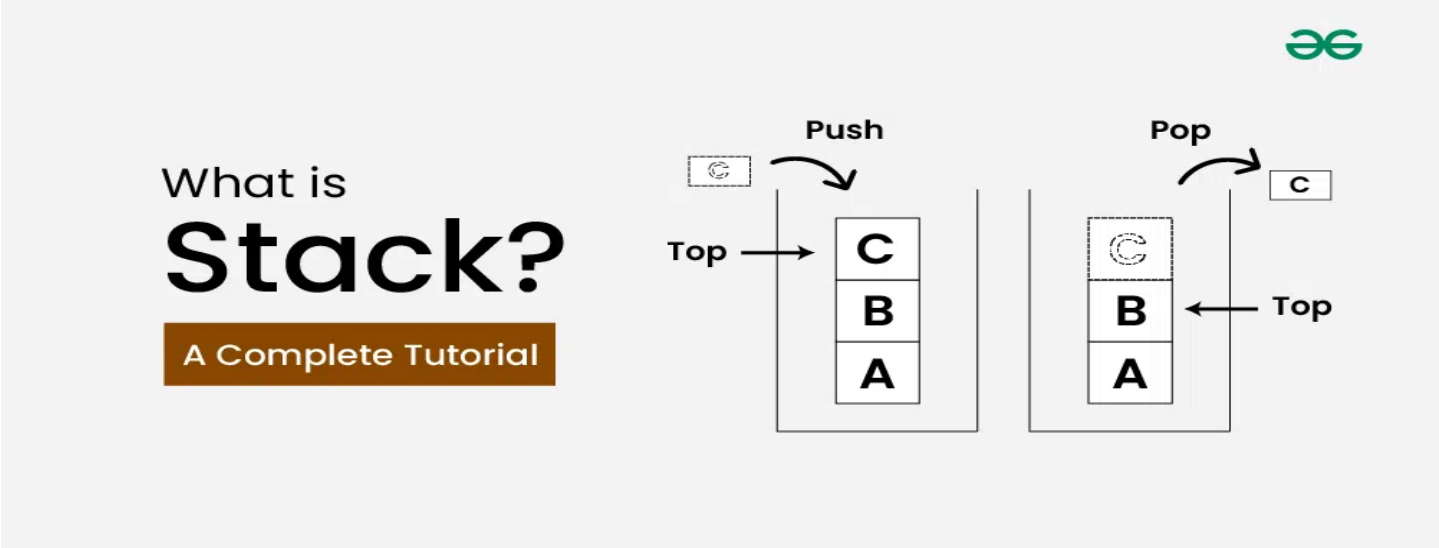
Disadvantages of Linked Lists:

Linked lists are a popular data structure in computer science, but like any other data structure, they have certain disadvantages as well. Some of the key disadvantages of linked lists are:

- **Slow Access Time:** Accessing elements in a linked list can be slow, as you need to traverse the linked list to find the element you are looking for, which is an O(n) operation. This makes linked lists a poor choice for situations where you need to access elements quickly.
- **Pointers or References:** Linked lists use pointers or references to access the next node, which can make them more complex to understand and use compared to arrays. This complexity can make linked lists more difficult to debug and maintain.
- **Higher overhead:** Linked lists have a higher overhead compared to arrays, as each node in a linked list requires extra memory to store the reference to the next node.
- **Cache Inefficiency:** Linked lists are cache-inefficient because the memory is not contiguous. This means that when you traverse a linked list, you are not likely to get the data you need in the cache, leading to cache misses and slow performance.

What is Stack Data Structure?

Stack is a linear data structure that follows **LIFO (Last In First Out) Principle**, the last element inserted is the first to be popped out. It means both insertion and deletion operations happen at one end only.



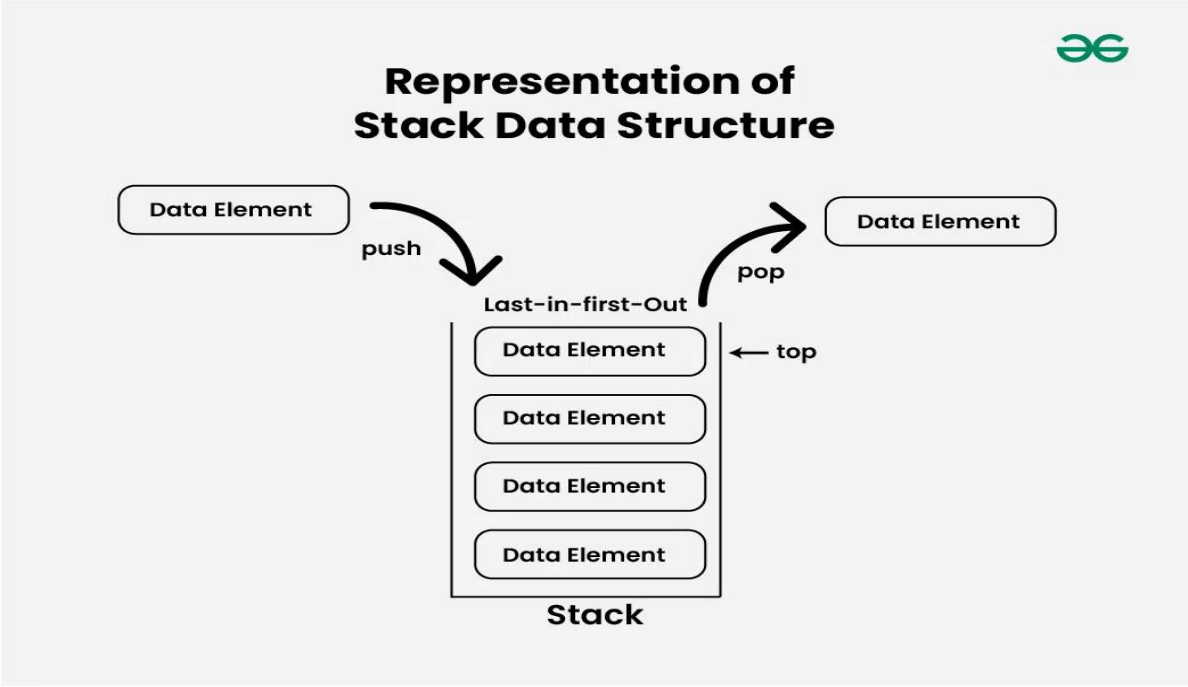
LIFO(Last In First Out) Principle

Here are some real world examples of LIFO

- Consider a stack of plates. When we add a plate, we add at the top. When we remove, we remove from the top.
- A **shuttlecock box** (or any other box that is closed from one end) is another great real-world example of the **LIFO (Last In, First Out)** principle where do insertions and removals from the same end.

Representation of Stack Data Structure:

Stack follows LIFO (Last In First Out) Principle so the element which is pushed last is popped first.



Types of Stack:

- **Fixed Size Stack** : As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- **Dynamic Size Stack** : A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack

is empty, it decreases its size. This type of stack is implemented using a linked list, as it allows for easy resizing of the stack.

Basic Operations on Stack:

In order to make manipulations in a stack, there are certain operations provided to us.

- **push()** to insert an element into the stack
- **pop()** to remove an element from the stack
- **top()** Returns the top element of the stack.
- **isEmpty()** returns true if stack is empty else false.
- **isFull()** returns true if the stack is full else false.

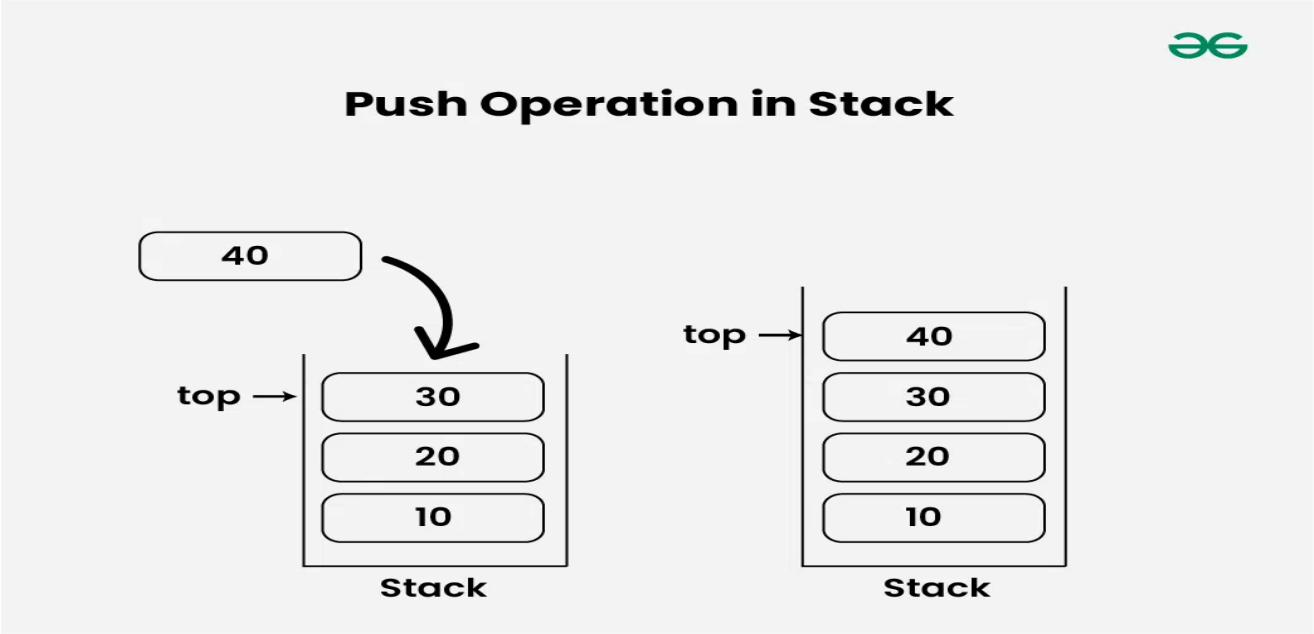
To implement stack, we need to maintain reference to the top item.

Push Operation on Stack

Adds an item to the stack. If the stack is full, then it is said to be an **Overflow condition.**

Algorithm for Push Operation:

- Before pushing the element to the stack, we check if the stack is **full** .
- If the stack is full **(top == capacity-1)** , then **Stack Overflows** and we cannot insert the element to the stack.
- Otherwise, we increment the value of top by 1 **(top = top + 1)** and the new value is inserted at **top position** .
- The elements can be pushed into the stack till we reach the **capacity** of the stack.

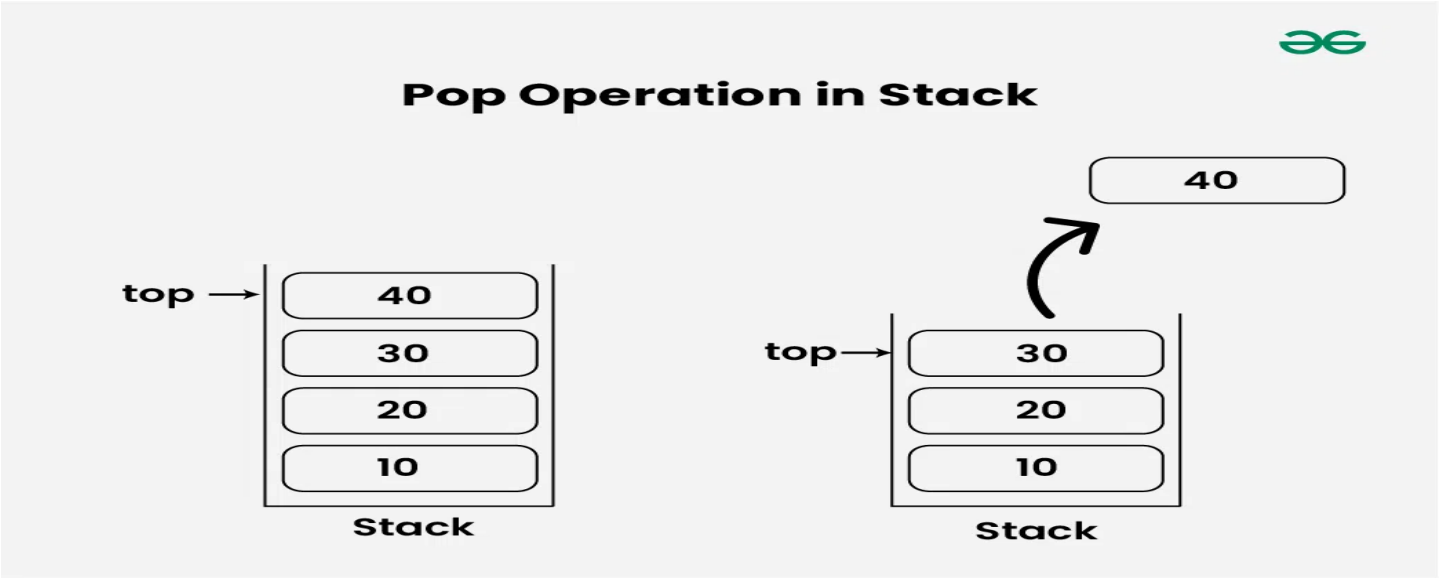


Pop Operation in Stack

Removes an item from the stack. The items are popped in the reversed order in which they are pushed. If the stack is empty, then it is said to be an **Underflow condition.**

Algorithm for Pop Operation:

- Before popping the element from the stack, we check if the stack is **empty** .
- If the stack is empty (top == -1), then **Stack Underflows** and we cannot remove any element from the stack.
- Otherwise, we store the value at top, decrement the value of top by 1 **(top = top - 1)** and return the stored top value.



Applications of Stacks:

- **Function calls:** Stacks are used to keep track of the return addresses of function calls, allowing the program to return to the correct location after a function has finished executing.
- **Recursion:** Stacks are used to store the local variables and return addresses of recursive function calls, allowing the program to keep track of the current state of the recursion.
- **Expression evaluation:** Stacks are used to evaluate expressions in postfix notation (Reverse Polish Notation).
- **Syntax parsing:** Stacks are used to check the validity of syntax in programming languages and other formal languages.
- **Memory management:** Stacks are used to allocate and manage memory in some operating systems and programming languages.
- Used to solve popular problems like Next Greater, Previous Greater, Next Smaller, Previous Smaller, Largest Area in a Histogram and Stock Span Problems.

Advantages of Stacks:

- **Simplicity:** Stacks are a simple and easy-to-understand data structure, making them suitable for a wide range of applications.
- **Efficiency:** Push and pop operations on a stack can be performed in constant time **(O(1))**, providing efficient access to data.
- **Last-in, First-out (LIFO):** Stacks follow the LIFO principle, ensuring that the last element added to the stack is the first one removed. This behavior is useful in many scenarios, such as function calls and expression evaluation.
- **Limited memory usage:** Stacks only need to store the elements that have been pushed onto them, making them memory-efficient compared to other data structures.

Disadvantages of Stacks:

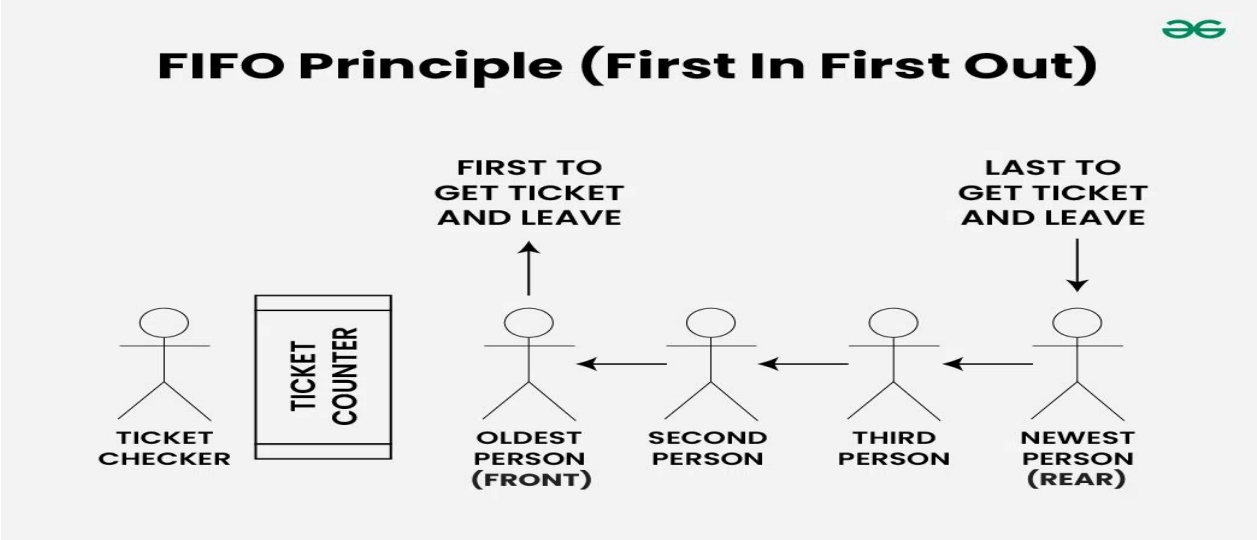
- **Limited access:** Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- **Potential for overflow:** If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.
- **Not suitable for random access:** Stacks do not allow for random access to elements, making them unsuitable for applications where elements need to be accessed in a specific order.
- **Limited capacity:** Stacks have a fixed capacity, which can be a limitation if the number of elements that need to be stored is unknown or highly variable.

Queue Data Structure

Queue is a linear data structure that follows **FIFO (First In First Out) Principle**, so the first element inserted is the first to be popped out.

FIFO Principle in Queue:

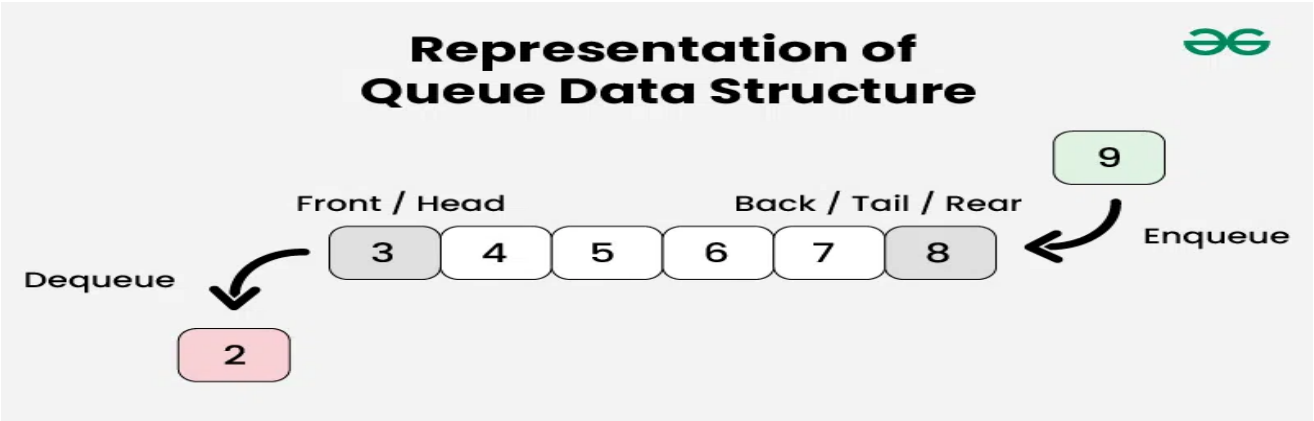
FIFO Principle states that the first element added to the Queue will be the first one to be removed or processed. So, Queue is like a line of people waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).



Basic Terminologies of Queue

- **Front:** Position of the entry in a queue ready to be served, that is, the first entry that will be removed from the queue, is called the **front** of the queue. It is also referred as the **head** of the queue.
- **Rear:** Position of the last entry in the queue, that is, the one most recently added, is called the **rear** of the queue. It is also referred as the **tail** of the queue.
- **Size:** Size refers to the **current** number of elements in the queue.
- **Capacity:** Capacity refers to the **maximum** number of elements the queue can hold.

Representation of Queue



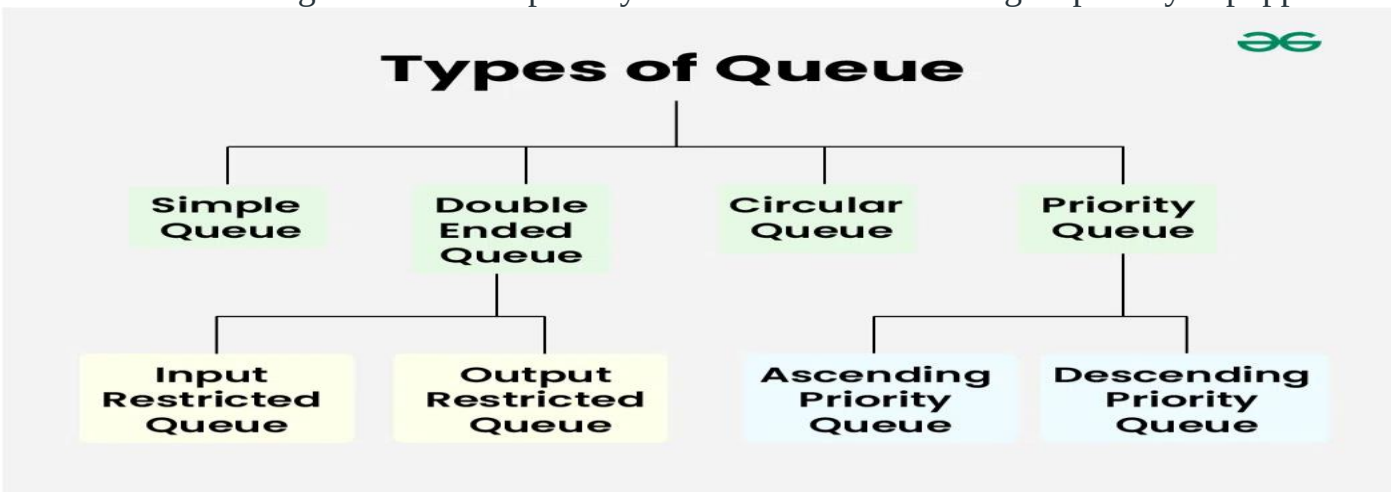
Queue Operations

1. **Enqueue**: Adds an element to the end (rear) of the queue. If the queue is full, an overflow error occurs.
2. **Dequeue**: Removes the element from the front of the queue. If the queue is empty, an underflow error occurs.
3. **Peek/Front**: Returns the element at the front without removing it.
4. **Size**: Returns the number of elements in the queue.
5. **isEmpty**: Returns true if the queue is empty, otherwise false.
6. **isFull**: Returns true if the queue is full, otherwise false.

Types of Queues

Queue data structure can be classified into 4 types:

1. **Simple/Linear Queue:** Simple Queue simply follows **FIFO** Structure. We can only insert the element at the back and remove the element from the front of the queue. A simple queue is efficiently implemented either using a linked list or a circular array.
2. **Double-Ended Queue (Deque):** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
    a. **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
    b. **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
3. **Circular Queue:** Circular Queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle. It is also called **'Ring Buffer'**. This queue is primarily used in the following cases:
    - **Memory Management:** The unused memory locations in the case of ordinary queues can be utilized in circular queues.
    - **Traffic system:** In a computer-controlled traffic system, circular queues are used to switch on the traffic lights one by one repeatedly as per the time set.
    - **CPU Scheduling:** Operating systems often maintain a queue of processes that are ready to execute or that are waiting for a particular event to occur.
    - The time complexity for the circular Queue is O(1).
4. **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:
    - **Ascending Priority Queue:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.
    - **Descending Priority Queue:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popped first.



types of queues

Applications of Queue Data Structure

Application of queue is common. In a computer system, there may be queues of tasks waiting for the printer, for access to disk storage, or even in a time-sharing system, for use of the CPU. Within a single program, there may be multiple requests to be kept in a queue, or one task may create other tasks, which must be done in turn by keeping them in a queue.

- A Queue is always used as a buffer when we have a speed mismatch between a producer and consumer. For example keyboard and CPU.

- Queue can be used where we have a single resource and multiple consumers like a single CPU and multiple processes.
- In a network, a queue is used in devices such as a router/switch and mail queue.
- Queue can be used in various algorithm techniques like Breadth First Search, Topological Sort, etc.

Basic Operations on Queue

Some of the basic operations for Queue in Data Structure are:

- **enqueue() -** Insertion of elements to the queue.
- **dequeue() -** Removal of elements from the queue.
- **getFront()-** Acquires the data element available at the front node of the queue without deleting it.
- **getRear() -** This operation returns the element at the rear end without removing it.
- **isFull() -** Validates if the queue is full.
- **isEmpty() -** Checks if the queue is empty.
- **size() -** This operation returns the size of the queue i.e. the total number of elements it contains.

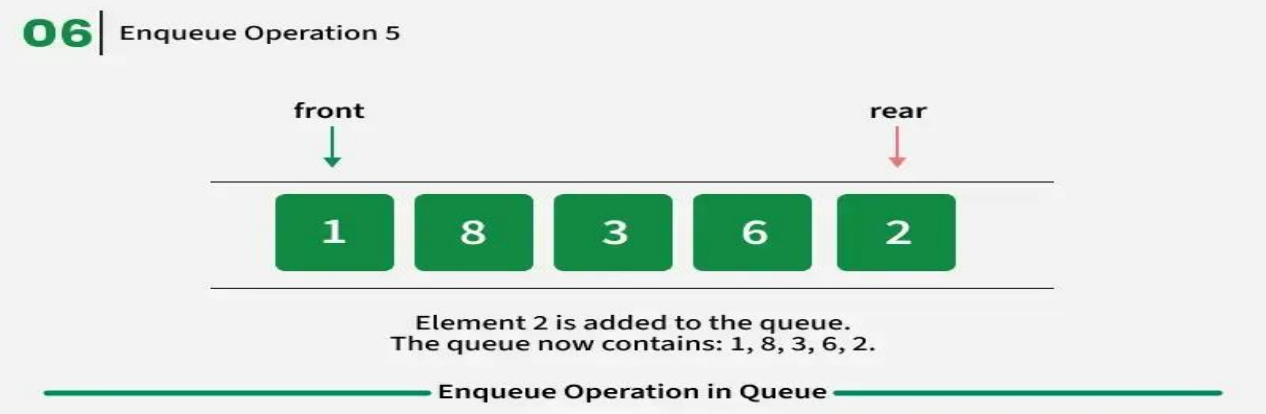
Queue Data Structure

Operation 1: enqueue()

Inserts an element at the end of the queue i.e. at the rear end.

The following steps should be taken to enqueue (insert) data into a queue:

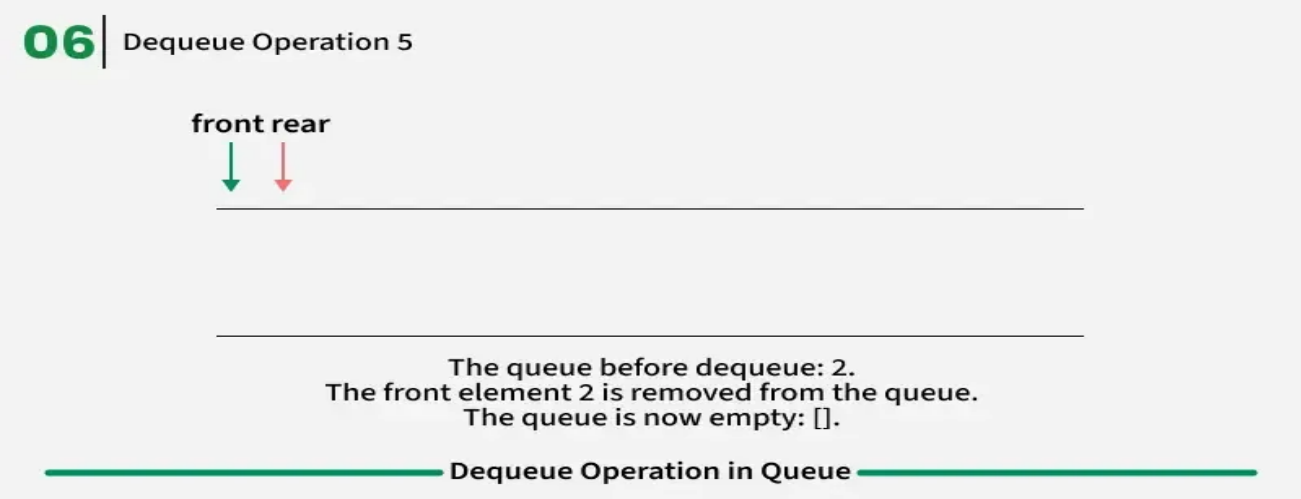
- Check if the queue is full.
- If the queue is full, return overflow error and exit.
- If the queue is not full, increment the rear pointer to point to the next empty space.
- Add the data element to the queue location, where the rear is pointing.
- return success.



Operation 2: dequeue()

This operation removes and returns an element that is at the front end of the queue.
The following steps are taken to perform the dequeue operation:

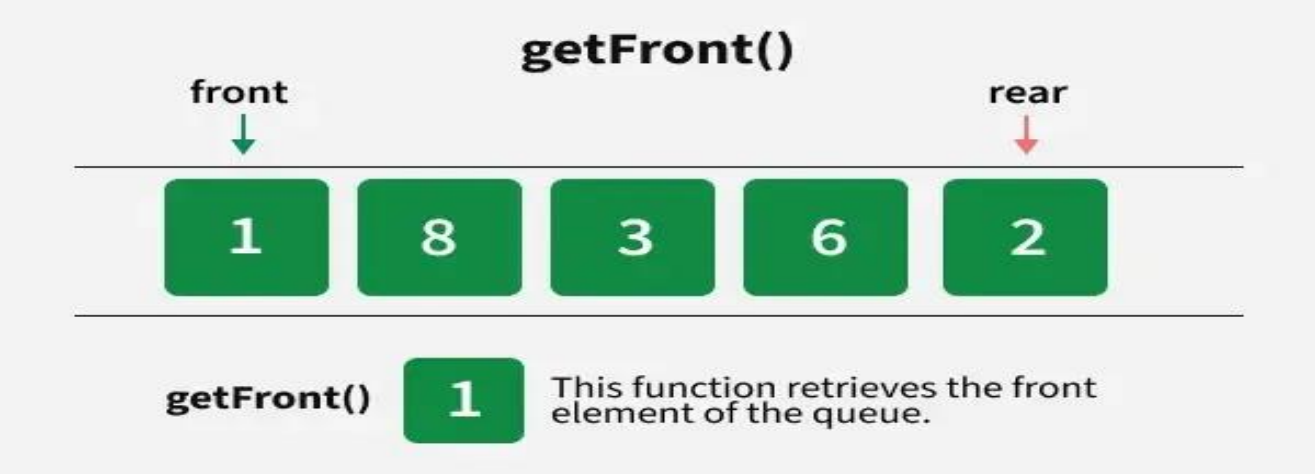
- Check if the queue is empty.
- If the queue is empty, return the underflow error and exit.
- If the queue is not empty, access the data where the front is pointing.
- Increment the front pointer to point to the next available data element.
- The Return success.



Operation 3: getFront()

This operation returns the element at the front end of the queue without removing it.
The following steps are taken to perform the getFront() operation:

- If the queue is empty, return the most minimum value (e.g., -1).
- Otherwise, return the front value of the queue.

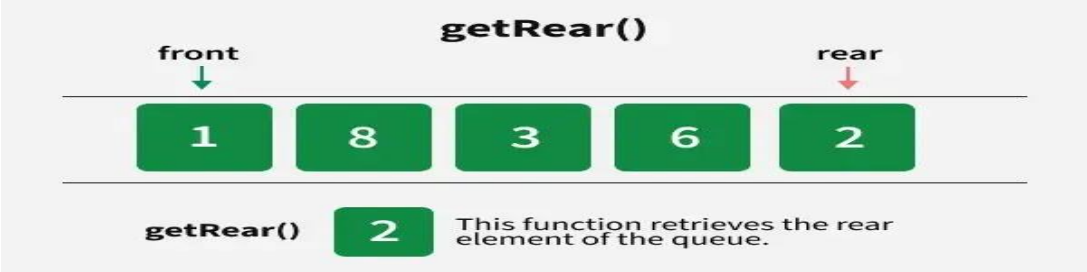


getFront

Operation 4: getRear()

This operation returns the element at the rear end without removing it.
The following steps are taken to perform the rear operation:

- If the queue is empty return the most minimum value.
- otherwise, return the rear value.

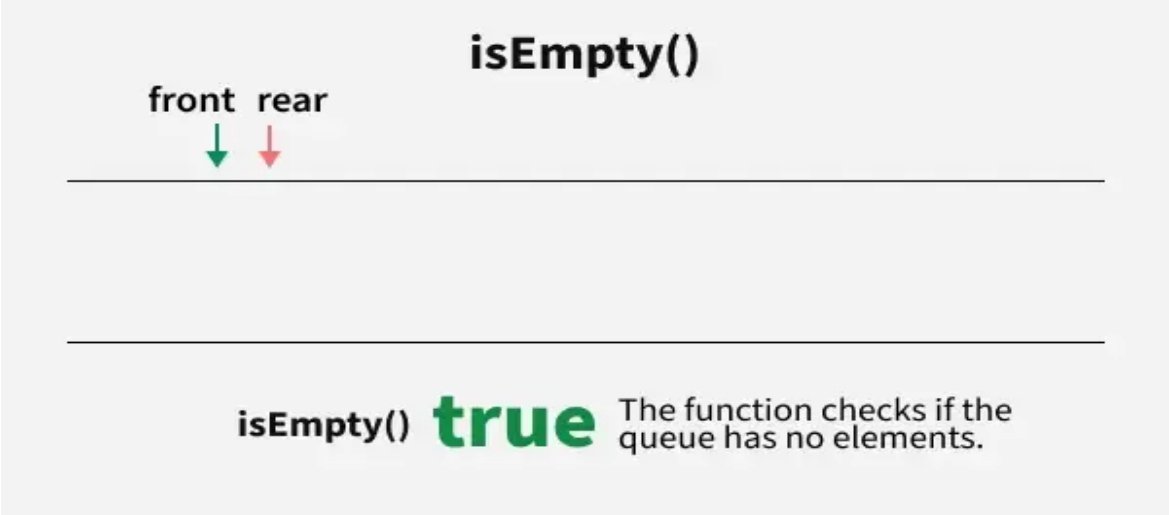


getRear

Operation 5: isEmpty()

This operation returns a boolean value that indicates whether the queue is empty or not.
The following steps are taken to perform the Empty operation:

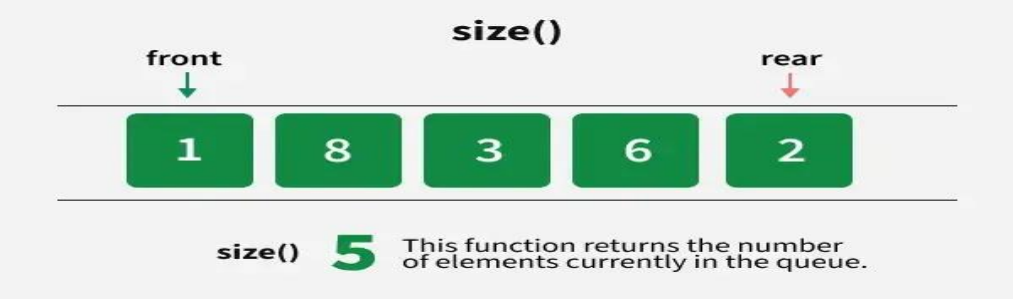
- check if front value is equal to -1 or not, if yes then return true means queue is empty.
- Otherwise return false, means queue is not empty.



isEmpty

Operation 6: size()

This operation returns the size of the queue i.e. the total number of elements it contains.



size()

Imlementation using Array

Array-based implementation in data structures refers to the use of arrays as the underlying storage mechanism for various abstract data types (ADTs). Arrays, being contiguous blocks of memory, offer efficient direct access to elements based on their index.

Key characteristics of array-based implementation:

- **Contiguous memory allocation:**
  Elements are stored in adjacent memory locations, allowing for fast access.
- **Fixed size (typically):**
  Arrays are often declared with a predetermined maximum size, which can lead to limitations if the data grows beyond that size. Dynamic arrays or resizing mechanisms are sometimes used to address this.
- **Direct access:**
  Elements can be accessed directly using their index (e.g., `array[i]`), providing O(1) time complexity for access operations.
- **Space efficiency:**
  Compared to some other data structures (like linked lists), arrays can be more space-efficient as they don't require extra memory for pointers.

Examples of ADTs commonly implemented using arrays:

- **Lists/Vectors:**

A dynamic array, like a C++ `std::vector` or Python `list`, expands or shrinks as needed by reallocating a larger array and copying elements when the current capacity is exceeded.

- **Stacks:**
  A stack can be implemented using an array where elements are added and removed from one end (the "top" of the stack).
- **Queues:**
  A queue can be implemented using an array, where elements are added at one end (the "rear") and removed from the other (the "front"). Circular arrays are often used to efficiently manage space.
- **Hash Tables:**
  Arrays are used as the primary storage for key-value pairs in hash tables, with a hash function mapping keys to array indices.
- **Heaps:**
  A heap, a tree-based data structure that satisfies the heap property, can be efficiently represented using an array.
- **Matrices:**
  Multi-dimensional arrays are a natural choice for representing matrices in various applications.

Advantages of array-based implementation:

- **Fast access:** O(1) for element access.
- **Simplicity:** Conceptually straightforward to understand and implement.
- **Cache efficiency:** Contiguous memory locations can lead to better cache performance.

Disadvantages of array-based implementation:

- **Fixed size limitations:** Resizing can be computationally expensive if frequent.
- **Inefficient insertions/deletions (in the middle):** Shifting elements to maintain contiguity can be slow, especially for large arrays.
- **Memory overhead for sparse data:** If an array is declared with a large size but only a few elements are stored, it can lead to wasted memory.

Imlementation using Linked List

A linked list is a linear data structure that consists of a sequence of nodes, where each node contains data and a reference (or pointer) to the next node in the sequence. Unlike arrays, linked lists do not store elements in contiguous memory locations, offering flexibility in size and efficient insertion/deletion operations.

Implementation Steps:
Node Structure Definition.

Define a structure or class for a `Node`. Each node typically contains:

- `data`: The actual value stored in the node.
- `next`: A pointer or reference to the next node in the list. In a singly linked list, the last node's `next` pointer points to `NULL` (or equivalent).

C

```
struct Node {
    int data;
    struct Node* next;
};
```

Linked List Class (Optional but Recommended).

Create a class (e.g., `LinkedList`) to encapsulate the linked list operations and manage the `head` (and potentially `tail`) of the list.

Java

```
class LinkedList {
    Node head; // Reference to the first node

    // Constructor, methods for operations
}
```

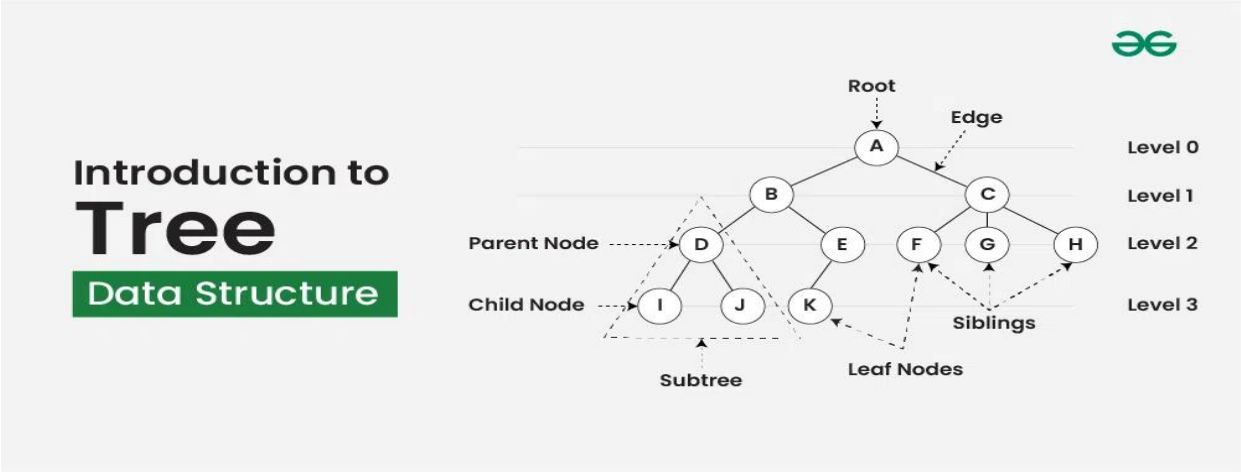
Unit III

Tree Data Structure

Tree data structure is a hierarchical structure that is used to represent and organize data in the form of parent child relationship. The following are some real world situations which are naturally a tree.

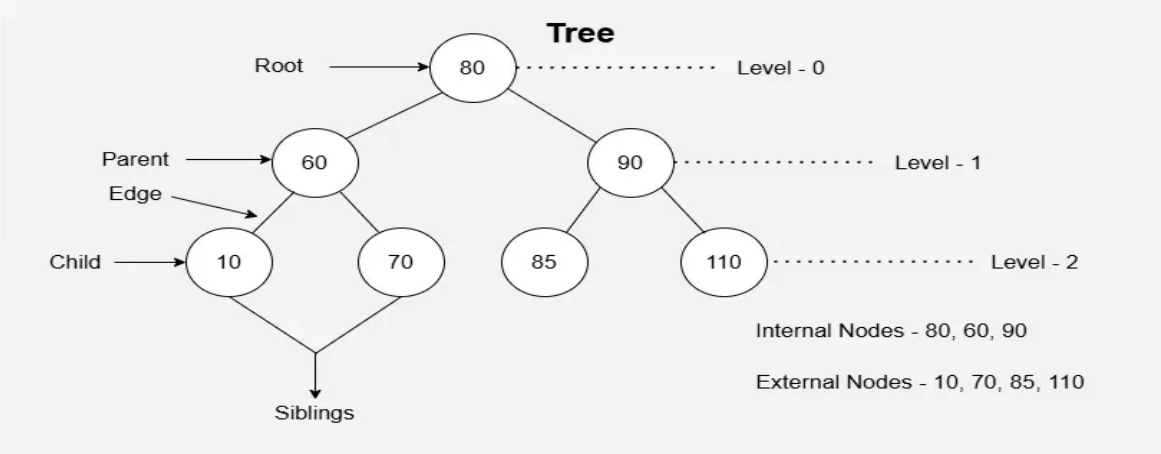
- Folder structure in an operating system.
- Tag structure in an HTML (root tag the as html tag) or XML document.

The topmost node of the tree is called the **root**, and the nodes below it are called the child nodes. Each node can have multiple child nodes, and these child nodes can also have their own child nodes, forming a recursive structure.



Basic Terminologies In Tree Data Structure:

- **Parent Node:** The node which is an immediate predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.
- **Child Node:** The node which is the immediate successor of a node is called the child node of that node. Examples: **{D, E}** are the child nodes of **{B}.**
- **Root Node:** The topmost node of a tree or the node which does not have any parent node is called the root node. {A**}** is the root node of the tree. A non-empty tree must contain exactly one root node and exactly one path from the root to all other nodes of the tree.
- **Leaf Node or External Node:** The nodes which do not have any child nodes are called leaf nodes. **{I, J, K, F, G, H}** are the leaf nodes of the tree.
- **Ancestor of a Node:** Any predecessor nodes on the path of the root to that node are called Ancestors of that node. **{A,B}** are the ancestor nodes of the node **{E}**
- **Descendant:** A node x is a descendant of another node y if and only if y is an ancestor of x.
- **Sibling:** Children of the same parent node are called siblings. **{D,E}** are called siblings.
- **Level of a node:** The count of edges on the path from the root node to that node. The root node has level **0**.
- **Internal node:** A node with at least one child is called Internal Node.
- **Neighbour of a Node:** Parent or child nodes of that node are called neighbors of that node.
- **Subtree**: Any node of the tree along with its descendant.



Basic Terminologies In Tree

Types of Tree data structures:

Tree data structure can be classified into three types based upon the number of children each node of the tree can have. The types are:

- **Binary tree:** In a binary tree, each node can have a maximum of two children linked to it. Some common types of binary trees include full binary trees, complete binary trees, balanced binary trees, and degenerate or pathological binary trees. Examples of Binary Tree are Binary Search Tree and Binary Heap.
- **Ternary Tree:** A Ternary Tree is a tree data structure in which each node has at most three child nodes, usually distinguished as “left”, “mid” and “right”.
- **N-ary Tree or Generic Tree:** Generic trees are a collection of nodes where each node is a data structure that consists of records and a list of references to its children(duplicate references are not allowed). Unlike the linked list, each node stores the address of multiple nodes.

Please refer Types of Trees in Data Structures for details.

Basic Operations Of Tree Data Structure:

- **Create** – create a tree in the data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check whether it is present or not.
- **Traversal**:
    - Depth-First-Search Traversal
    - Breadth-First-Search Traversal

Implementation of Tree Data Structure:

```
// C++ program to demonstrate some of the above
// terminologies
#include <bits/stdc++.h>
using namespace std;

// Function to add an edge between vertices x and y
void addEdge(int x, int y, vector<vector<int> >& adj)
{
   adj[x].push_back(y);
   adj[y].push_back(x);
}

// Function to print the parent of each node
void printParents(int node, vector<vector<int> >& adj,
                  int parent)
{
```

```
    // current node is Root, thus, has no parent
    if (parent == 0)
        cout << node << "->Root" << endl;
    else
        cout << node << "->" << parent << endl;

    // Using DFS
    for (auto cur : adj[node])
        if (cur != parent)
            printParents(cur, adj, node);
}

// Function to print the children of each node
void printChildren(int Root, vector<vector<int> >& adj)
{
    // Queue for the BFS
    queue<int> q;

    // pushing the root
    q.push(Root);

    // visit array to keep track of nodes that have been
    // visited
    int vis[adj.size()] = { 0 };

    // BFS
    while (!q.empty()) {
        int node = q.front();
        q.pop();
        vis[node] = 1;
        cout << node << "-> ";
        for (auto cur : adj[node])
            if (vis[cur] == 0) {
                cout << cur << " ";
                q.push(cur);
            }
        cout << endl;
    }
}

// Function to print the leaf nodes
void printLeafNodes(int Root, vector<vector<int> >& adj)
{
    // Leaf nodes have only one edge and are not the root
    for (int i = 1; i < adj.size(); i++)
        if (adj[i].size() == 1 && i != Root)
            cout << i << " ";
    cout << endl;
}

// Function to print the degrees of each node
void printDegrees(int Root, vector<vector<int> >& adj)
{
    for (int i = 1; i < adj.size(); i++) {
        cout << i << ": ";
```

```
        // Root has no parent, thus, its degree is equal to
        // the edges it is connected to
        if (i == Root)
            cout << adj[i].size() << endl;
        else
            cout << adj[i].size() - 1 << endl;
    }
}
// Driver code
int main()
{
    // Number of nodes
    int N = 7, Root = 1;
    // Adjacency list to store the tree
    vector<vector<int> > adj(N + 1, vector<int>());
    // Creating the tree
    addEdge(1, 2, adj);
    addEdge(1, 3, adj);
    addEdge(1, 4, adj);
    addEdge(2, 5, adj);
    addEdge(2, 6, adj);
    addEdge(4, 7, adj);

    // Printing the parents of each node
    cout << "The parents of each node are:" << endl;
    printParents(Root, adj, 0);

    // Printing the children of each node
    cout << "The children of each node are:" << endl;
    printChildren(Root, adj);

    // Printing the leaf nodes in the tree
    cout << "The leaf nodes of the tree are:" << endl;
    printLeafNodes(Root, adj);

    // Printing the degrees of each node
    cout << "The degrees of each node are:" << endl;
    printDegrees(Root, adj);

    return 0;
}
```

Output

```
The parents of each node are:
1->Root
2->1
5->2
6->2
3->1
4->1
7->4
```

```
The children of each node are:
1-> 2 3 4
2-> 5 6
3->
4-> 7
5->
6->
7->
The leaf nodes of the tree are:
3 5 6 7
The degrees o...
```

Properties of Tree Data Structure:

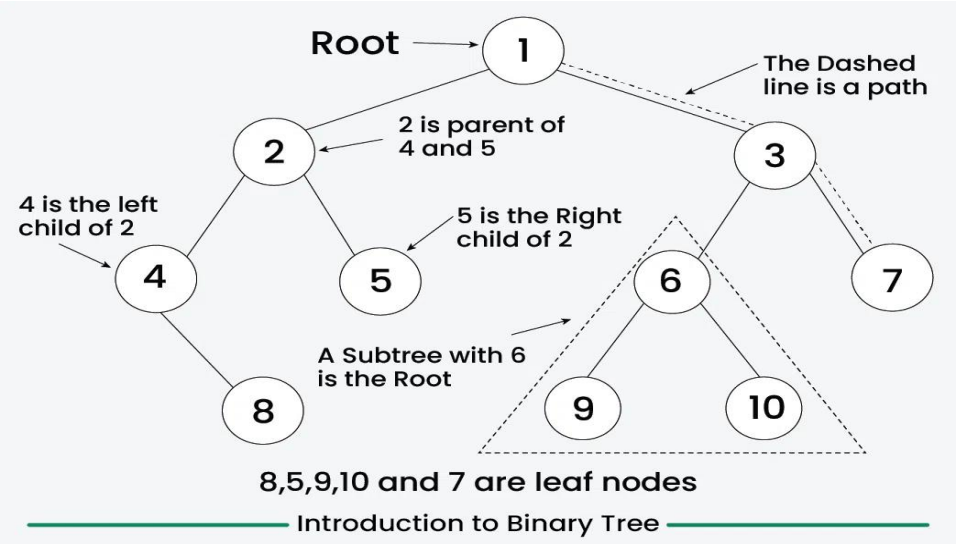
Number of edges: An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.

Depth of a node: The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

- **Height of a node:** The height of a node can be defined as the length of the longest path from the node to a leaf node of the tree.
- **Height of the Tree:** The height of a tree is the length of the longest path from the root of the tree to a leaf node of the tree.
- **Degree of a Node:** The total count of subtrees attached to that node is called the degree of the node. The degree of a leaf node must be **0**. The degree of a tree is the maximum degree of a node among all the nodes in the tree.

Binary Tree

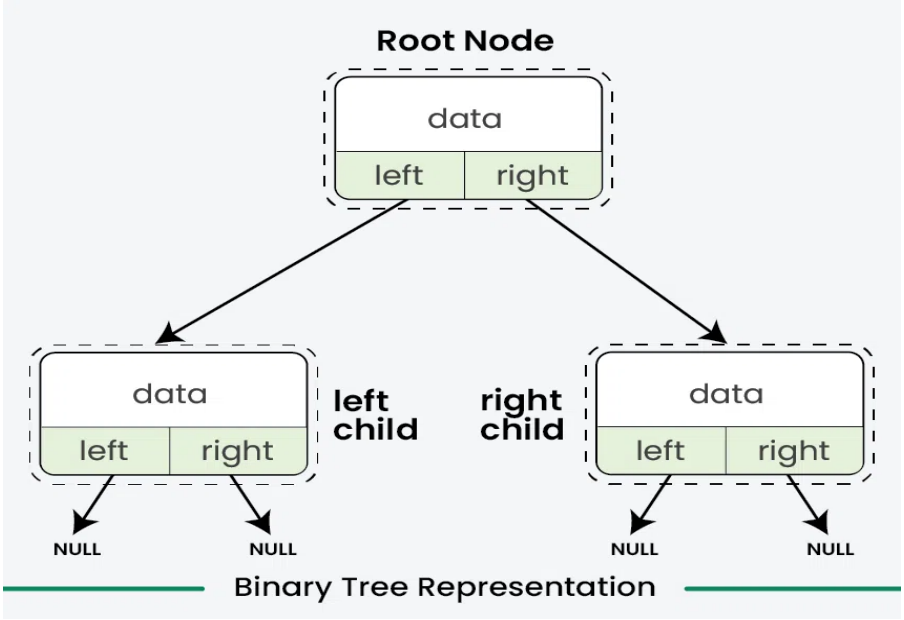
Binary Tree is a **non-linear** and **hierarchical** data structure where each node has at most two children referred to as the **left child** and the **right child**. The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**.



Representation of Binary Tree

Each node in a Binary Tree has three parts:

- Data
- Pointer to the left child
- Pointer to the right child



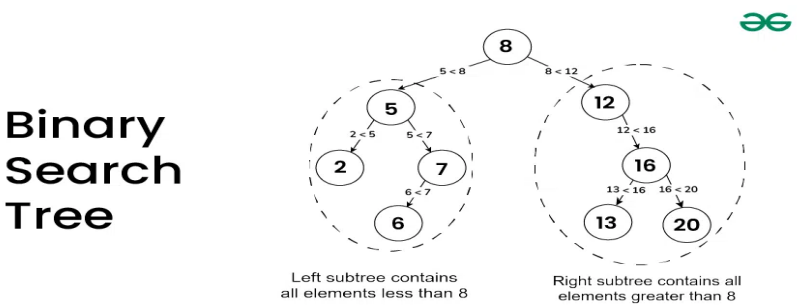
Binary Tree Representation

Properties of Binary Tree

- The maximum number of nodes at level **L** of a binary tree is 2^L
- The maximum number of nodes in a binary tree of height **H** is $2^H - 1$
- Total number of leaf nodes in a binary tree = total number of nodes with 2 children + 1
- In a Binary Tree with **N** nodes, the minimum possible height or the minimum number of levels is $Log_2(N+1)$
- A Binary Tree with **L** leaves has at least $|\log_2(L)| + 1$ levels

Binary Search Tree

Binary Search Tree is a data structure used in computer science for organizing and storing data in a sorted manner. Binary search tree follows all properties of binary tree and for every nodes, its **left** subtree contains values less than the node and the **right** subtree contains values greater than the node. This hierarchical structure allows for efficient **Searching**, **Insertion**, and **Deletion** operations on the data stored in the tree.



Binary Search Tree

Properties of Binary Search Tree:

- The left subtree of a node contains only nodes with keys less than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree.
- There must be no duplicate nodes(BST may have duplicate values with different handling approaches).

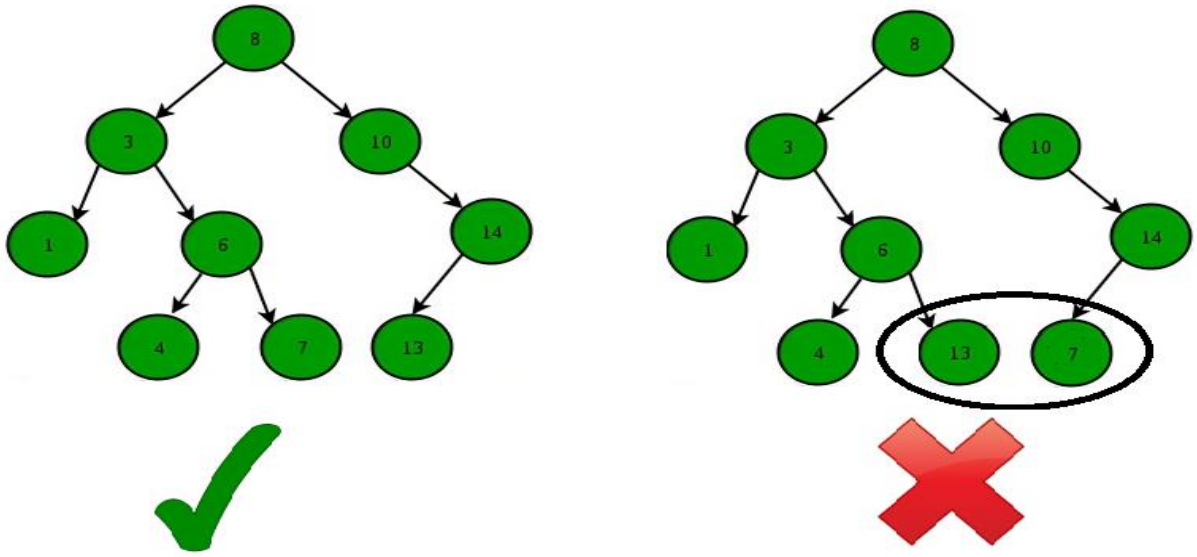
Important Points about BST

- A Binary Search Tree is useful for maintaining sorted stream of data. It allows search, insert, delete, ceiling, max and min in O(h) time. Along with these, we can always traverse the tree items in sorted order.
- With Self Balancing BSTs, we can ensure that the height of the BST is bound be Log n. Hence we achieve, the above mentioned O(h) operations in O(Log n) time.
- When we need only search, insert and delete and do not need other operations, we prefer Hash Table over BST as a Hash Table supports these operations in O(1) time on average.

Applications

Binary Search Tree (BST) is a data structure that is commonly used to implement efficient searching, insertion, and deletion operations along with maintaining sorted sequence of data. Please remember the following properties of BSTs before moving forward.

- The left subtree of a node contains only nodes with keys lesser than the node's key.
- The right subtree of a node contains only nodes with keys greater than the node's key.
- The left and right subtree each must also be a binary search tree. There must be no duplicate nodes.



Binary Search Tree - Structure

A BST supports operations like search, insert, delete, maximum, minimum, floor, ceil, greater, smaller, etc in O(h) time where h is height of the BST. To keep height less, self balancing BSTs (like AVL and Red Black Trees) are used in practice. These Self-Balancing BSTs maintain the height as O(Log n). Therefore all of the above mentioned operations become O(Log n). Together with these, BST also allows sorted order traversal of data in O(n) time.

1. A Self-Balancing Binary Search Tree is used to maintain sorted stream of data. For example, suppose we are getting online orders placed and we want to maintain the live data (in RAM) in sorted order of prices. For example, we wish to know number of items purchased at cost below a given cost at any moment. Or we wish to know number of items purchased at higher cost than given cost.

2. A Self-Balancing Binary Search Tree is used to implement doubly ended priority queue. With a Binary Heap, we can either implement a priority queue with support of extractMin() or with extractMax(). If we wish to support both the operations, we use a Self-Balancing Binary Search Tree to do both in O(Log n)
3. There are many more algorithm problems where a Self-Balancing BST is the best suited data structure, like count smaller elements on right, Smallest Greater Element on Right Side, etc.
4. A BST can be used to sort a large dataset. By inserting the elements of the dataset into a BST and then performing an in-order traversal, the elements will be returned in sorted order. When compared to normal sorting algorithms, the advantage here is, we can later insert / delete items in O(Log n) time.
5. Variations of BST like B Tree and B+ Tree are used in Database indexing.
6. **TreeMap** and **TreeSet** **in** Java, and set and map in C++ are internally implemented using self-balancing BSTs, more formally a Red-Black Tree.

Advantages of Binary Search Tree (BST):

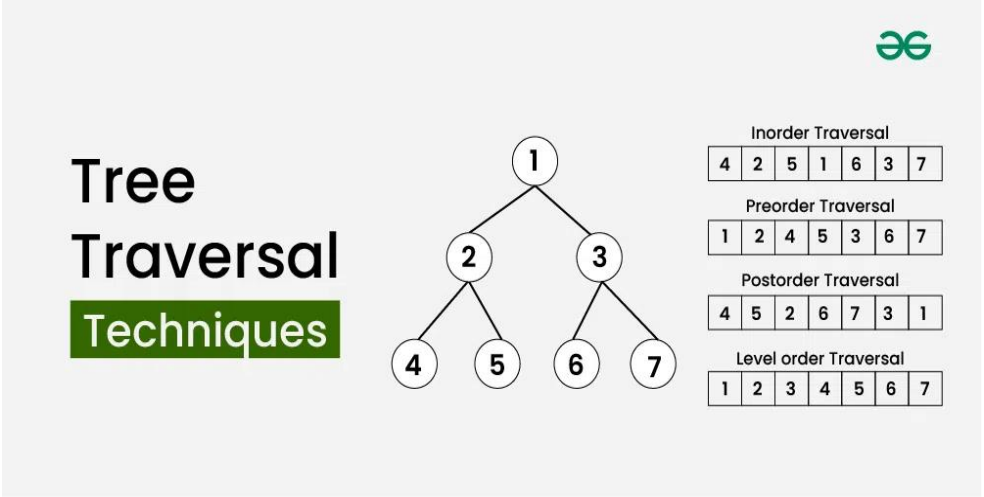
- **Efficient searching:** O(log n) time complexity for searching with a self balancing BST
- **Ordered structure:** Elements are stored in sorted order, making it easy to find the next or previous element
- **Dynamic insertion and deletion:** Elements can be added or removed efficiently
- **Balanced structure:** Balanced BSTs maintain a logarithmic height, ensuring efficient operations
- **Doubly Ended Priority Queue**: In BSTs, we can maintain both maximum and minimum efficiently

Disadvantages of Binary Search Tree (BST):

- **Not self-balancing:** Unbalanced BSTs can lead to poor performance
- **Worst-case time complexity:** In the worst case, BSTs can have a linear time complexity for searching and insertion
- **Memory overhead:** BSTs require additional memory to store pointers to child nodes
- **Not suitable for large datasets:** BSTs can become inefficient for very large datasets
- **Limited functionality:** BSTs only support searching, insertion, and deletion operations

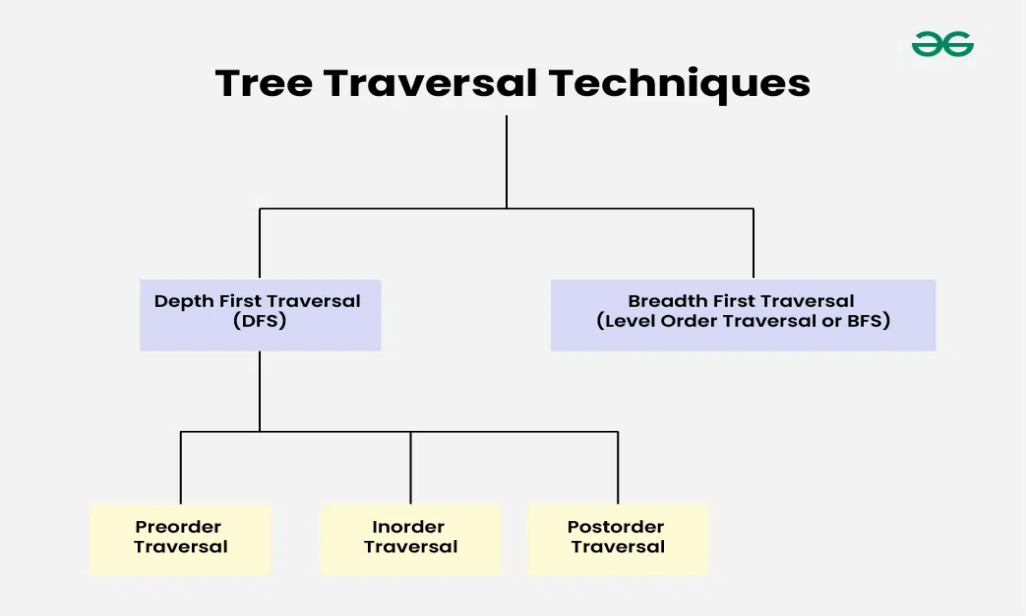
Tree Traversal Techniques

Tree Traversal techniques include various ways to visit all the nodes of the tree. Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. In this article, we will discuss all the tree traversal techniques along with their uses.



Tree Traversal

Tree Traversal refers to the process of visiting or accessing each node of the tree exactly once in a certain order. Tree traversal algorithms help us visit and process all the nodes of the tree. Since a tree is not a linear data structure, there can be multiple choices for the next node to be visited. Hence we have many ways to traverse a tree.



There are multiple tree traversal techniques that decide the order in which the nodes of the tree are to be visited. These are defined below:

- Depth First Search or DFS
    - Inorder Traversal
    - Preorder Traversal
    - Postorder Traversal
- Level Order Traversal or Breadth First Search or BFS

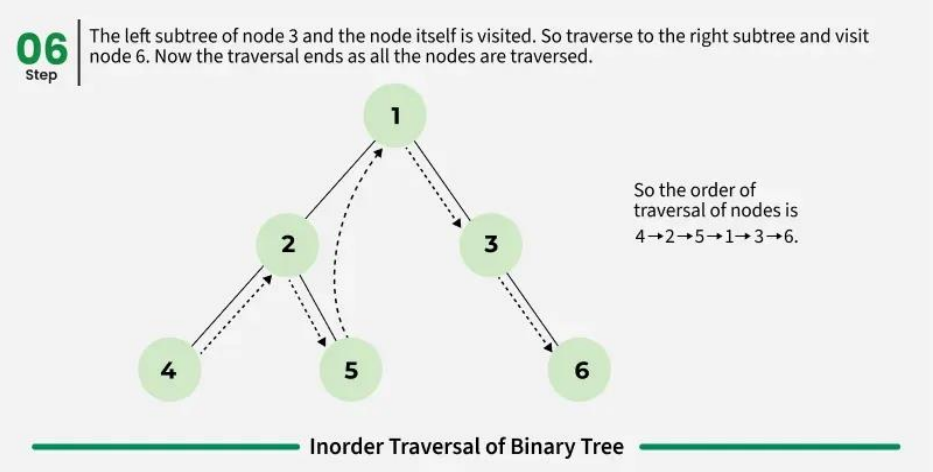
Inorder Traversal

Inorder traversal visits the node in the order: **Left -> Root -> Right**

Algorithm for Inorder Traversal

Inorder(tree)

- *Traverse the left subtree, i.e., call Inorder(left->subtree)*
- *Visit the root.*
- *Traverse the right subtree, i.e., call Inorder(right->subtree)*



Uses of Inorder Traversal

- In the case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order.
- To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.
- Inorder traversal can be used to evaluate arithmetic expressions stored in expression trees.

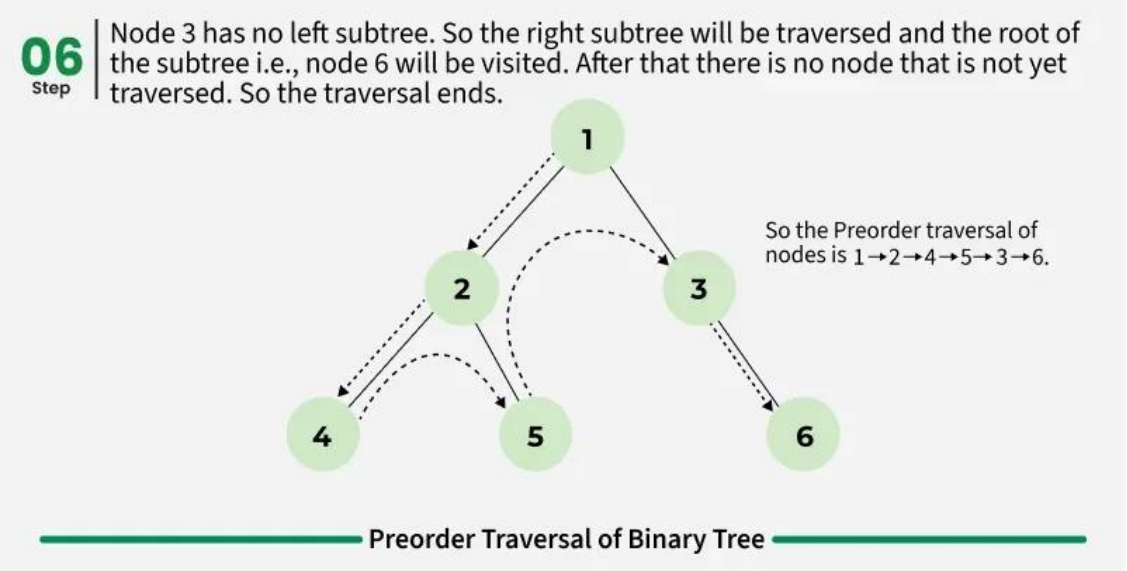
Preorder Traversal

Preorder traversal visits the node in the order: **Root -> Left -> Right**

Algorithm for Preorder Traversal

Preorder(tree)

- *Visit the root.*
- *Traverse the left subtree, i.e., call Preorder(left->subtree)*
- *Traverse the right subtree, i.e., call Preorder(right->subtree)*



Uses of Preorder Traversal

- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expressions on an expression tree.

Also Check: *Refer Preorder Traversal of Binary Tree for more*

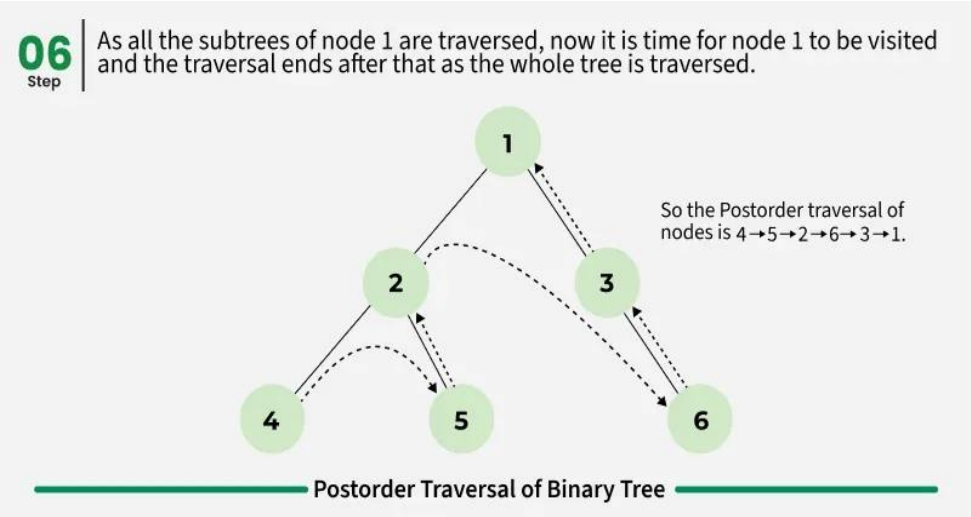
Postorder Traversal

Postorder traversal visits the node in the order: **Left -> Right -> Root**

Algorithm for Postorder Traversal:

Postorder(tree)

- *Traverse the left subtree, i.e., call Postorder(left->subtree)*
- *Traverse the right subtree, i.e., call Postorder(right->subtree)*
- *Visit the root*

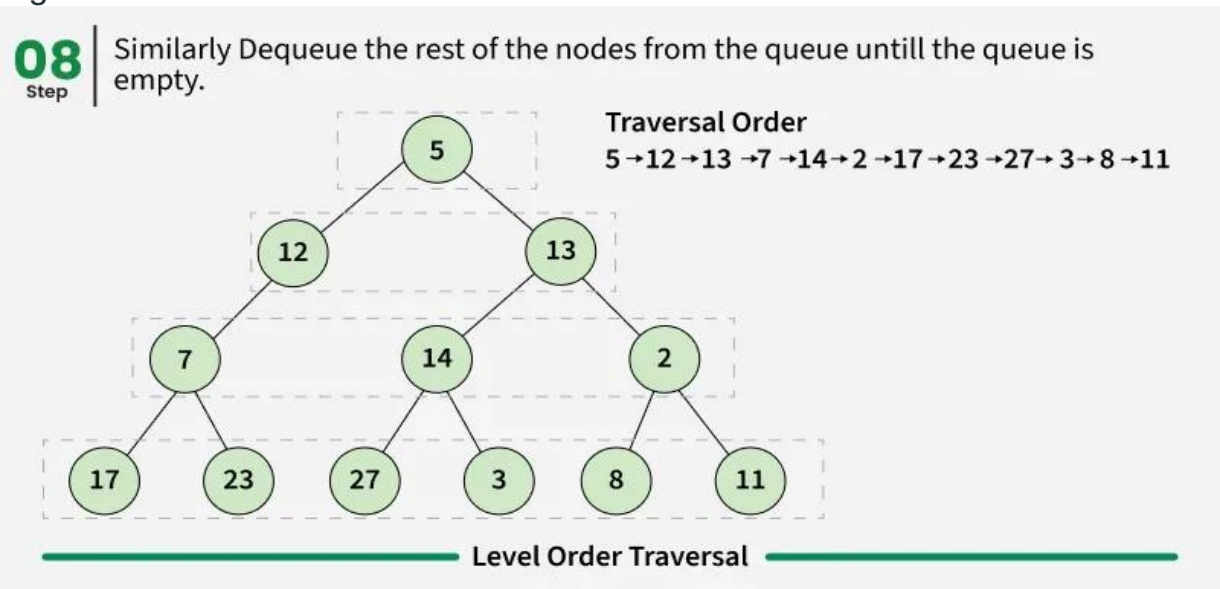


Uses of Postorder Traversal

- Postorder traversal is used to delete the tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.
- Postorder traversal can help in garbage collection algorithms, particularly in systems where manual memory management is used.

Level Order Traversal

Level Order Traversal visits all nodes present in the same level completely before visiting the next level.



Algorithm for Level Order Traversal

LevelOrder(tree)

● *Create an empty queue Q*
● *Enqueue the root node of the tree to Q*
● *Loop while Q is not empty*
○*Dequeue a node from Q and visit it*
○ *Enqueue the left child of the dequeued node if it exists*
○ *Enqueue the right child of the dequeued node if it exists .*

Uses of Level Traversal

1. Level-wise node processing, like finding maximum/minimum at each level.
2. Tree serialization/deserialization for efficient storage and reconstruction.

3. Solving problems like calculating the "maximum width of a tree" by processing nodes level by level.

What is Searching in Data Structures?

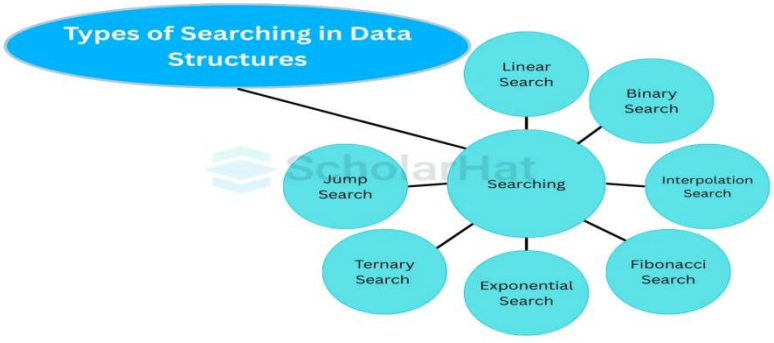
Searching is the fundamental process of locating a specific element or item within a collection of data. This collection of data can be **arrays**, **lists**, **trees**, or other structured representations. **Data structures** are complex systems designed to organize vast amounts of information. Searching within a data structure is one of the most critical operations performed on stored data.
The goal is to find the desired information with its precise location quickly and with minimal computational resources. It plays an important role in various computational tasks and real-world applications, including information retrieval, data analysis, decision-making processes, etc.

Characteristics of Searching

- **Target Element/Key:** It is the 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:** It 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.
- **Deterministic vs. Non-deterministic:** The algorithms that follow a clear, systematic approach, like binary search, are deterministic. Others, such as linear search, are non-deterministic, as they may need to examine the entire search space in the worst case.

Searching Algorithms in Data Structures

Data structures require specialized search algorithms to enable effective retrieval of a variety of information, from web content to scientific data. Computers would struggle to process large amounts of data effectively without these techniques. Searching Algorithms are designed to check for an element or retrieve an element from any data structure where it is stored. Hence, it is important to understand different search algorithms and why it is important to use one over another.



Searching Techniques: Linear Search and Binary Search

Linear Search

1. Start at the beginning of the list.
2. Compare the target value with the current element in the list.
3. If the current element matches the target value, the search is successful, and the position or index of the element is returned.
4. If the current element does not match the target value, move to the next element in the list.
5. Repeat steps 2-4 until a match is found or the end of the list is reached.
6. If the end of the list is reached without finding a match, the search is unsuccessful, and a special value (e.g., -1) may be returned to indicate the absence of the target value

Pseudocode for Linear Search Algorithm

- LinearSearch(array, target):
- for each element in array, from left to right:
- if element equals target:
- return index of element
- return -1 // target not found in the array

Example of Linear Search

```
#include <iostream>
using namespace std;

int search(int array[], int n, int key) {
   for (int i = 0; i < n; i++)
      if (array[i] == key)
         return i;
   return -1;
}

int main() {
   int num = 5;
   int arr[] = {10, 20, 30, 40, 50};
   int search = 30;
   bool found = false;

   for (int cnt = 0; cnt < num; cnt++) {
      if (arr[cnt] == search) {
         cout << search << " is present at location " << (cnt + 1) << "." << endl;
         found = true;
         break;
      }
   }

   if (!found) {
      cout << search << " is not present in the array." << endl;
```

```
    }

    return 0;
```

Output

30 is present at location 3.

Binary Search

1. Start with a sorted array or list. For binary search to work correctly, the elements must be in ascending or descending order.
2. Set two pointers, low and high, to the beginning and end of the search space, respectively. Initially, low = 0 and high = length of the array - 1.
3. Calculate the middle index using the formula: mid = (low + high) / 2. This will give you the index of the element in the middle of the current search space.
4. Compare the target value with the element at the middle index:
    - If they are equal, the target value has been found. Return the index of the middle element.
    - If the target value is less than the middle element, set high = mid - 1 and go to step 3.
    - If the target value is greater than the middle element, set low = mid + 1 and go to step 3.
5. Repeat steps 3-4 until the target value is found or low > high. If low becomes greater than high, it means the target value is not present in the array.

Pseudocode for Binary Search Algorithm

- BinarySearch(array, target): left = 0 // Initialize the left pointer
- right = Length(array) - 1 // Initialize the right pointer while left <= right:
- mid = (left + right) / 2 // Calculate the middle index if array[mid] == target:
- return mid // Target found at the middle index
- else if array[mid] < target:
- left = mid + 1 // Adjust left pointer
- else:
- right = mid - 1 // Adjust right pointer return -1 // Target not found in the array

Example of Binary Search

```
#include <iostream>
using namespace std;

int main() {
   int size = 6;
   int arr[] = {1, 3, 5, 4, 10, 7};
   int sElement = 10;

   int f = 0;
   int l = size - 1;
   int m = (f + l) / 2;
```

```
    while (f <= l) {
        if (arr[m] < sElement) {
            f = m + 1;
        } else if (arr[m] == sElement) {
            cout << "Element found at index " << m << "." << endl;
            break;
        } else {
            l = m - 1;
        }
        m = (f + l) / 2;
    }

    if (f > l) {
        cout << "Element not found in the list." << endl;
    }

    return 0;
}
```

Output

Element found at index 4.

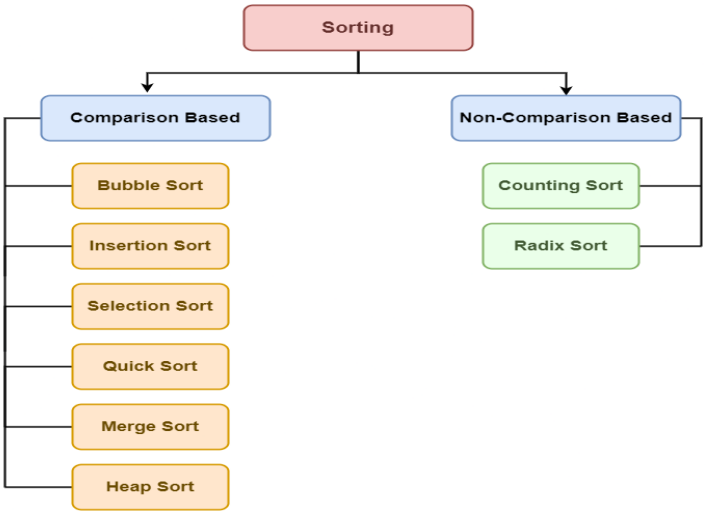
Sorting

Sorting refers to rearrangement of a given array or list of elements according to a comparison operator on the elements. The comparison operator is used to decide the new order of elements in the respective data structure.

Types of Sorting Techniques

There are various sorting algorithms are used in data structures. The following two types of sorting algorithms can be broadly classified:

1. **Comparison-based:** We compare the elements in a comparison-based sorting algorithm)
2. **Non-comparison-based:** We do not compare the elements in a non-comparison-based sorting algorithm)



Sorting Algorithms:

Bubble Sort - O(n^2) Time and O(1) Space

It is a simple sorting algorithm that repeatedly swaps adjacent elements if they are in the wrong order. It performs multiple passes through the array, and in each pass, the largest unsorted element moves to its correct position at the end.
After each pass, we ignore the last sorted elements and continue comparing and swapping remaining adjacent pairs. After k passes, the last k elements are sorted.

Code

```
#include <iostream>
#include <vector>
using namespace std;

// An optimized version of Bubble Sort
void bubbleSort(vector<int>& arr) {
    int n = arr.size();
    bool swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        for (int j = 0; j < n - i - 1; j++) {
            if (arr[j] > arr[j + 1]) {
                swap(arr[j], arr[j + 1]);
                swapped = true;
            }
        }

        // If no two elements were swapped, then break
        if (!swapped)
            break;
    }
}
```

```
int main() {
    vector<int> arr = { 5, 6, 1, 3 };
    bubbleSort(arr);
    for (int num : arr)
        cout << num << " ";
}
```

Output
11 12 22 25 34 64 90

Insertion Sort - O(n^2) Time and O(1) Space

It is a simple sorting algorithm that builds the sorted array one element at a time. It works like sorting playing cards in your hand, where each new card is inserted into its correct position among the already sorted cards.
We start with the second element, assuming the first is already sorted. If the second element is smaller, we shift the first element and insert the second in the correct position. Then we move to the third element and place it correctly among the first two. This process continues until the entire array is sorted.

Code

```
#include <iostream>
#include <vector>
using namespace std;

void insertionsort(vector<int>& arr, int n) {
    for (int i = 1; i < n; ++i) {
        int key = arr[i];
        int j = i - 1;

        // move elements greater than key one position ahead
        while (j >= 0 && arr[j] > key) {
            arr[j + 1] = arr[j];
            j--;
        }
        arr[j + 1] = key;
    }
}

int main() {
    vector<int> arr = {12, 11, 13, 5, 6};
    int n = arr.size();

    insertionsort(arr, n);

    for (int i = 0; i < n; ++i)
        cout << arr[i] << " ";

    return 0;
}
```

Output

```
5 6 11 12 13
```

Selection Sort - O(n^2) Time and O(1) Space

It is a comparison-based sorting algorithm that repeatedly selects the smallest (or largest) element from the unsorted part of the array and swaps it with the first unsorted element. This process continues until the array is fully sorted.
We start by finding the smallest element and swap it with the first element. Then we find the next smallest element among the remaining and swap it with the second element. This continues until all elements are placed in their correct positions.

Code

```
#include <iostream>
#include <vector>
using namespace std;

void selectionSort(vector<int> &arr) {
    int n = arr.size();

    for (int i = 0; i < n - 1; ++i) {

        // Assume the current position holds
        // the minimum element
        int min_idx = i;

        // Iterate through the unsorted portion
        // to find the actual minimum
        for (int j = i + 1; j < n; ++j) {
            if (arr[j] < arr[min_idx]) {

                // Update min_idx if a smaller
                // element is found
                min_idx = j;
            }
        }

        // Move minimum element to its
        // correct position
        swap(arr[i], arr[min_idx]);
    }
}

int main() {
    vector<int> arr = {64, 25, 12, 22, 11};
    selectionSort(arr);

    for (int &val : arr) {
        cout << val << " ";
```

```
        }

        return 0;
    }
```

Output

```
11 12 22 25 64
```