Introductory Concepts

- **Data**

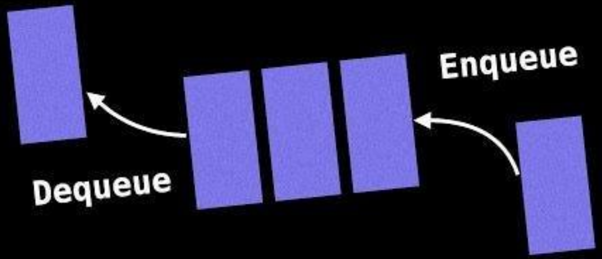
  Data is a collection of facts, numbers, words, observations or other useful information.

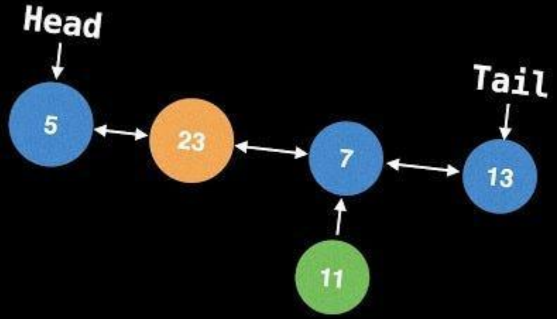
- **Data Sets**

  A data set is a collection of related data.

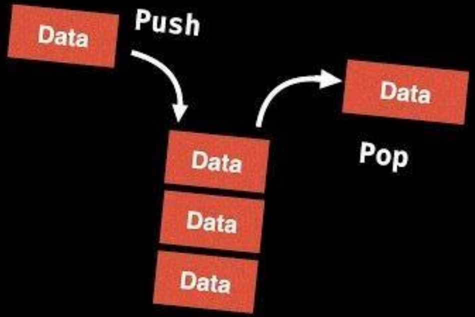
- **Entity**

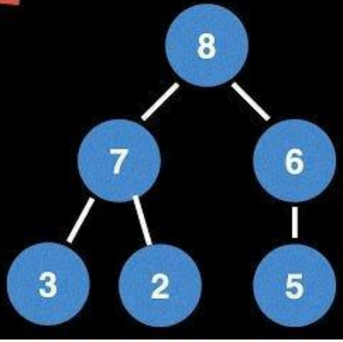
  An entity is a "thing" or "object" in the real world. An entity contains attributes and or properties which may be assigned values. The values may be either numeric or non-numeric.

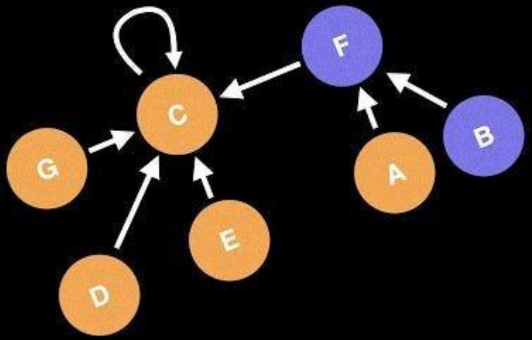




Data Structures







Data Structure

- Data structure is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.
- Defines the relationship between different sets of data.

Data + Structures = Data Structures

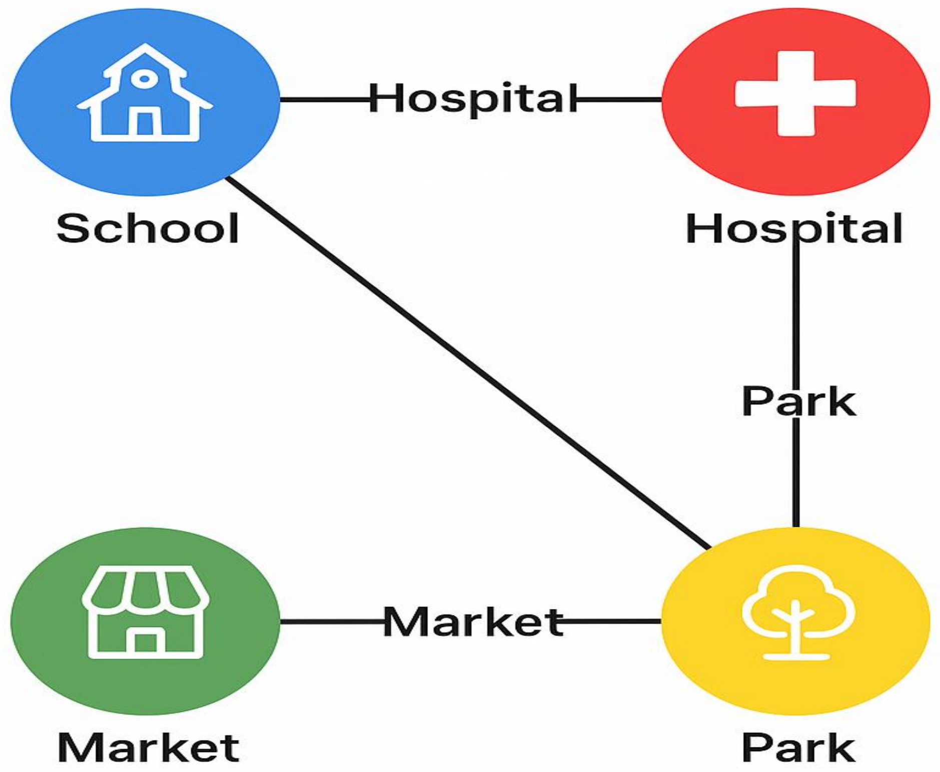
Examples: Arrays,Linked List ,Stack etc.,

Town as A Data Structure

Consider a small town with:

- 🏫School
- 🏥Hospital
- 🛒Market
- 🌳Park

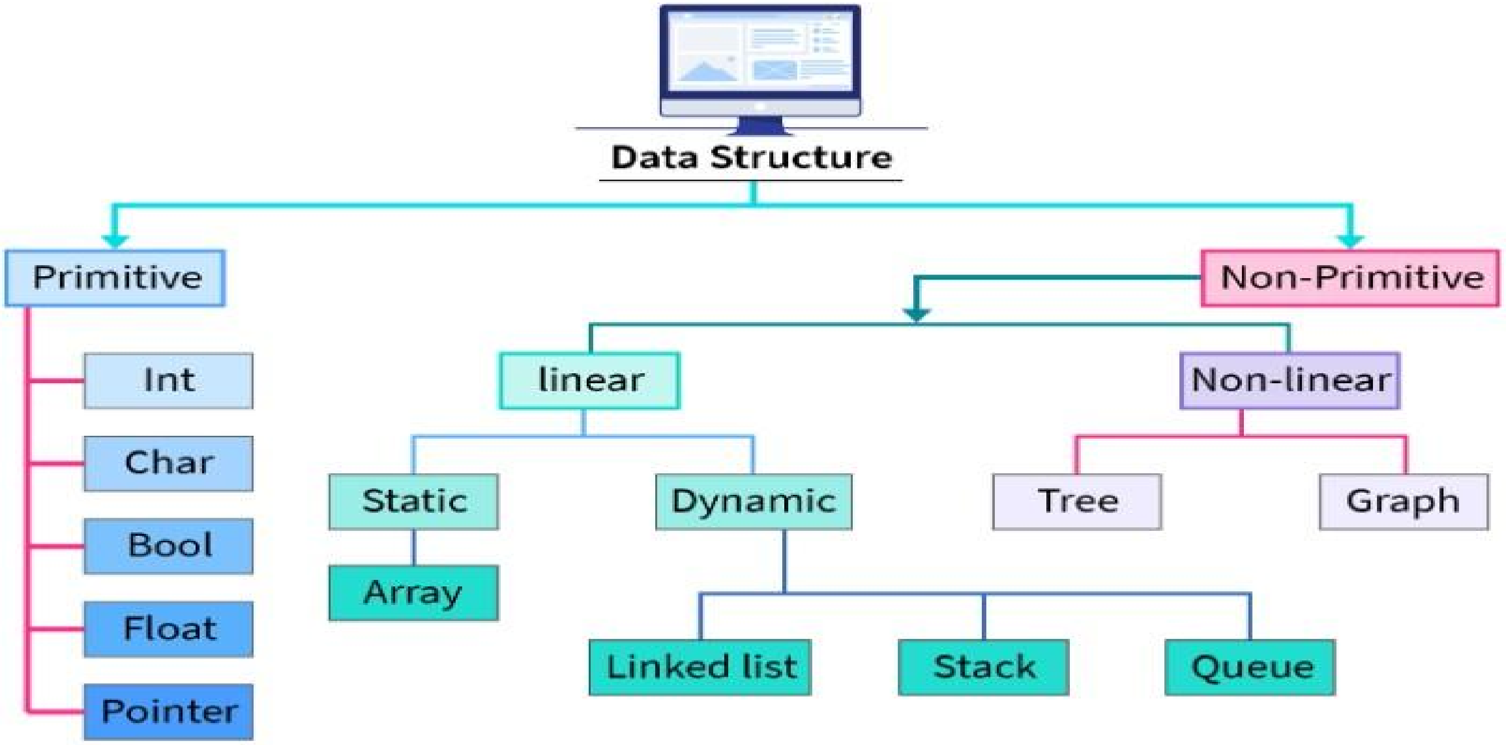
Each location is a **node** and Roads connecting them are **edges.**


Hospital
School
Hospital
Park
Market
Market
Park

Need for Data Structures

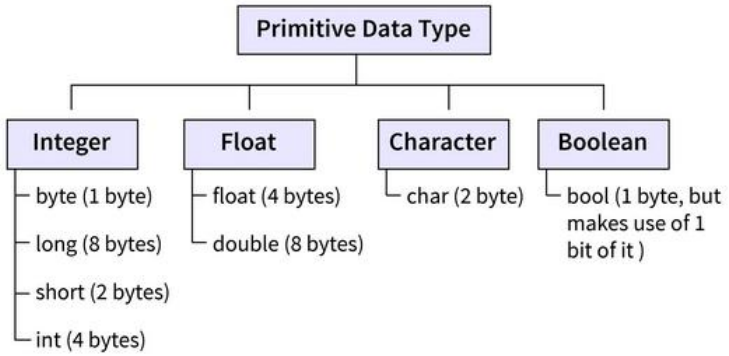
- Efficient Data Organization and Management
- Improved Program Performance
- Problem-Solving in Various Fields
- Enhancing Code Readability and Maintainability
- Building Blocks of Algorithms
- Optimizing Resource Usage

Classification of Data Structure



Primitive Data structure

- The primitive data structures are built-in(primitive) data types.
- The int, char, float, double are the primitive data structures that can hold a single value.
- These data structures can be manipulated or operated directly by machine-level instructions.



Non-Primitive Data structure

- Non-primitive data structures are those data structures which are derived from primitive data structures and can store data of multiple types.
- These data structures can’t be manipulated or operated directly by machine-level instructions.
- Example: Arrays,linked lists, stacks, queues, trees, and graphs.
  – Non-primitive data structures are further divided into two categories:

  1. Linear Data Structures
  2. Non-Linear Data Structures

Linear Data Structures

- ➔ A linear data structure is one where elements are arranged sequentially, meaning each element is connected to its predecessor and successor.
- ➔ Examples include arrays, linked lists, stacks, and queues.
- ➔ These structures are useful for storing and accessing data in a sequential manner.

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

- **Static Data Structures**

  Static Data Structures have a predetermined size allocated during compilation, and users cannot alter this size after compilation.

  Example :Array

- **Dynamic Data Structures**

  Dynamic Data Structures are those whose size can change during runtime, with memory allocated as needed. Users can modify both the size and the data elements stored within these structures while the code is executing.

  Examples :Linked Lists, Stacks, and Queues.

What is the primary benefit of using a dynamic data structure over a static data structure?

a. Faster access to elements

b.Fixed size during runtime

c.Ability to change size during runtime

d.Simpler implementation

Non-Linear Data Structure

- A Non-Linear Data Structure is a type of data structure in which elements are not arranged sequentially or in a linear order. Instead, they are connected in a hierarchical or complex relationship, allowing multiple paths for traversal.
- These structures are used to represent relationships between data elements more efficiently than linear structures.

Examples: Trees and Graphs

What is the main difference between linear and non-linear data structures?

a.Linear data structures have elements arranged in a sequence; non-linear data structures do not.

b.Linear data structures store data in multiple types; non-linear data structures store data in a single type.

c.Linear data structures are static; non-linear data structures are dynamic.

d.Linear data structures are only used in databases; non-linear data structures are used in operating systems.

Applications of Data Structure

- Artificial intelligence
- Compiler design
- Machine learning
- Database design and management
- Blockchain
- Numerical and Statistical analysis
- Operating system development
- Image & Speech Processing
- Cryptography

Advantages of Data Structures

- **Efficiency**: Proper selection of data structures enhances the program's efficiency in terms of time and space.
- **Reusability:** Data structures can be reused across multiple programs, enhancing their utility.
- **Abstraction**: Data structures defined by ADTs offer a level of abstraction, allowing clients to use them without needing to understand the internal implementation details.

Operations of Data Structures

- **Search Operation**

  Searching in a data structure involves looking for specific data elements that meet certain conditions.

- **Traversal Operation**

  Traversing a data structure involves going through each data element one by one to manage or process it.

- **Insertion Operation**

  Insertion means adding new data elements to a collection.

- **Deletion Operation**

  Deletion involves removing a specific data element from a list.

Operations of Data Structures

- **Update Operation**

  The Update operation enables us to modify existing data within the data structure.

- **Sorting Operation**

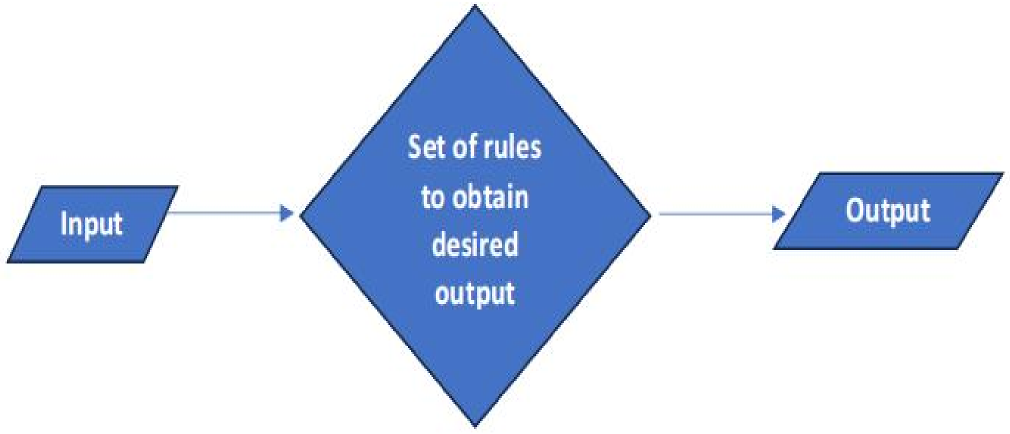
  Sorting involves arranging data elements in either ascending or descending order, depending on the application's requirements.

- **Merge Operation**

  Merge involves combining data elements from two sorted lists to create a single sorted list. This process ensures that the resulting list maintains the sorted order of the original lists.

Algorithm

An Algorithm is the step-by-step instructions to solve the problem.



Algorithm

Algorithm : Characteristics

- **Input**: An algorithm has some input values. We can pass 0 or some input value to an algorithm.
- **Output**: We will get 1 or more output at the end of an algorithm.
- **Definiteness**: - Every step of the algorithm should be clear and well defined.
- **Finiteness**: An algorithm should have finiteness. Here, finiteness means that the algorithm should contain a limited number of instructions, i.e., the instructions should be countable.
- **Effectiveness**: An algorithm should be effective as each instruction in an algorithm affects the overall process.
- **Language independent**: An algorithm must be language-independent so that the instructions in an algorithm can be implemented in any of the languages with the same output.

Algorithm: Example

Adding Two Numbers

A simple algorithm to add two numbers entered by the user:

1. Start
2. Declare three variables: a, b, and sum.
3. Input the values of a and b (prompt the user to enter the numbers).
4. Calculate the sum: sum = a + b.
5. Output the sum (display the result to the user).
6. Stop

Algorithm: Example

```
import java.util.Scanner;

public class AddTwoNumbers {

    public static void main(String[] args) {

        Scanner sc = new Scanner(System.in);

        System.out.print("Enter first number: ");

        int a = sc.nextInt();

       System.out.print("Enter second number: ");

        int b = sc.nextInt();

         int sum = a + b;

        System.out.println("The sum is: " + sum);

        }

}
```

Algorithm : Dataflow

- **Problem**: A problem can be a real-world problem or any instance from the real-world problem for which we need to create a program or the set of instructions.
- **Algorithm**: An algorithm will be designed for a problem which is a step by step procedure.
- **Input**: After designing an algorithm, the required and the desired inputs are provided to the algorithm.
- **Processing unit**: The input will be given to the processing unit, and the processing unit will produce the desired output.
- **Output**: The output is the outcome or the result of the program.

Analysis of Algorithms

- **How good is the algorithm?**
  - Correctness
  - Time efficiency
  - Space efficiency
- **Does there exist a better algorithm?**
  - Lower bounds
  - Optimality

Analysis of Algorithms

Priori Analysis:

– Here, priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm. Various factors can be considered before implementing the algorithm like processor speed, which has no effect on the implementation part.

Posterior Analysis:

– Here, posterior analysis is a practical analysis of an algorithm. The practical analysis is achieved by implementing the algorithm using any programming language. This analysis basically evaluate that how much running time and space taken by the algorithm.

Complexity Analysis

- How much time does this algorithm need to finish?
- How much space does this algorithm need for its computation?

Analysis of Algorithms

- An algorithm is said to be efficient and fast, if **it takes less time to execute and consumes less memory space**.
- The performance of an algorithm is measured on the basis of following properties :
  1. Time Complexity
  2. Space Complexity

Analysis of Algorithms

- ➢ **Time Complexity**

  –The time complexity of an algorithm is the amount of time required to complete the execution.

  –The time complexity of algorithms is most commonly expressed using the big O notation. It's an asymptotic notation to represent the time complexity.

- ➢ **Space complexity**

  –An algorithm’s space complexity is the amount of space required to solve a problem and produce an output.

Types of Analysis

To analyze an algorithm we need some kind of syntax, and that forms the base for asymptotic analysis/notation. There are three types of analysis:

• **Worst case**

○ Defines the input for which the algorithm takes a long time (slowest time to complete).

○ Input is the one for which the algorithm runs the slowest.

• **Best case**

○ Defines the input for which the algorithm takes the least time (fastest time to complete).

○ Input is the one for which the algorithm runs the fastest.

• **Average case**

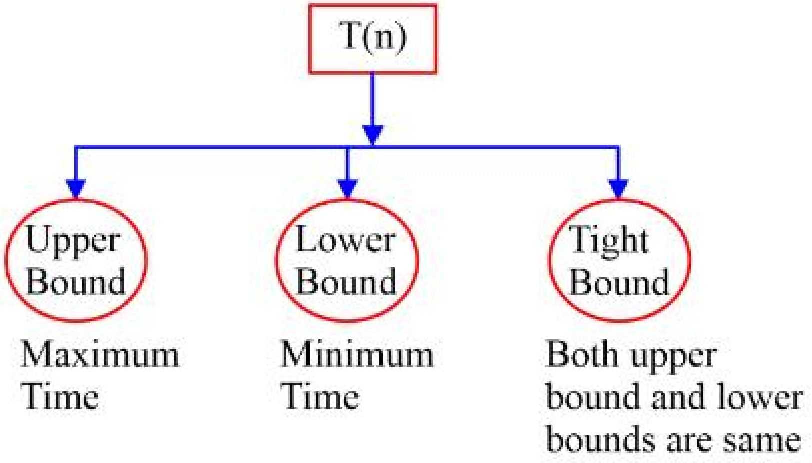
○ Provides a prediction about the running time of the algorithm.

○ Run the algorithm many times, using many different inputs that come from some distribution that generates these inputs, compute the total running time (by adding the individual times), and divide by the number of trials.

○ Assumes that the input is random.

Asymptotic Notations

Suppose, T(n) be a function of time for any algorithm



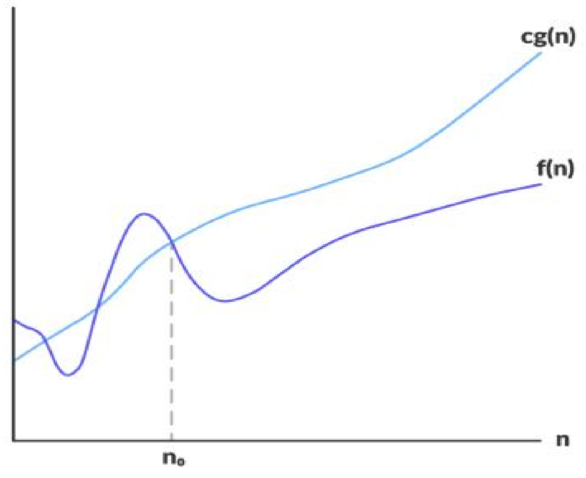
Asymptotic Notations

There are mainly three asymptotic notations:

- Big-O notation
- Omega notation
- Theta notation

Big-O Notation (O-notation)

Big-O notation represents the upper bound of the running time of an algorithm. Thus, it gives the worst-case complexity of an algorithm. Big-O gives the upper bound of a function.



f(n) = O(g(n))

Big-O Notation (O-notation)

- If f(n) and g(n) are the two functions defined for positive integers, then f(n) = O(g(n)) as f(n) is big oh of g(n) or f(n) is on the order of g(n)) if there exists constants c and n0 such that:

  $f(n) \leq c.g(n)$ for all $n \geq n0$

- This implies that f(n) does not grow faster than g(n), or g(n) is an upper bound on the function f(n).
- In this case, we are calculating the growth rate of the function which eventually calculates the worst time complexity of a function, i.e., how worst an algorithm can perform.
- Let's understand through examples
- Example 1: $f(n)=2n+3$, $g(n)=n$

Now, we have to find Is $f(n)=O(g(n))$?

- To check $f(n)=O(g(n))$, it must satisfy the given condition:$f(n)<=c.g(n)$

Big-O Notation (O-notation)

First, we will replace f(n) by 2n+3 and g(n) by n.

2n+3 <= c.n

- Let's assume c=5, n=1 then

2*1+3<=5*1

5<=5

- For n=1, the above condition is true.

If n=2

2*2+3<=5*2

7<=10

Big-O Notation (O-notation)

For n=2, the above condition is true.

We know that for any value of n, it will satisfy the above condition, i.e., 2n+3<=c.n. If the value of c is equal to 5, then it will satisfy the condition 2n+3<=c.n.

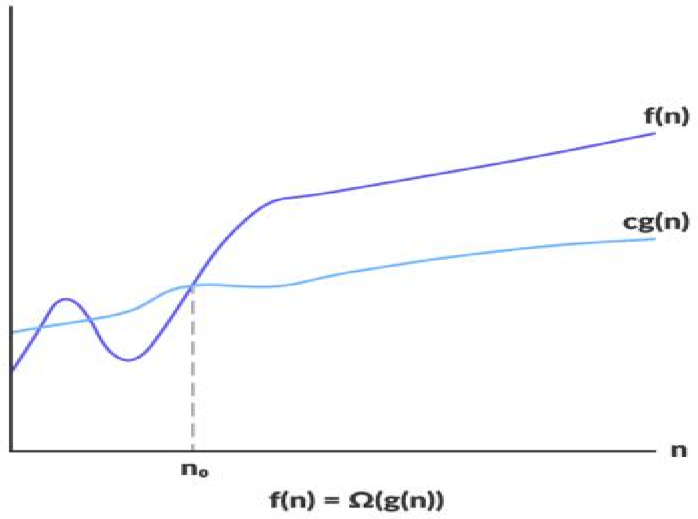
- We can take any value of n starting from 1, it will always satisfy. Therefore, we can say that for some constants c and for some constants n0, it will always satisfy 2n+3<=c.n.

- As it is satisfying the above condition, so f(n) is big oh of g(n) or we can say that f(n) grows linearly.

Therefore, it concludes that c.g(n) is the upper bound of the f(n).

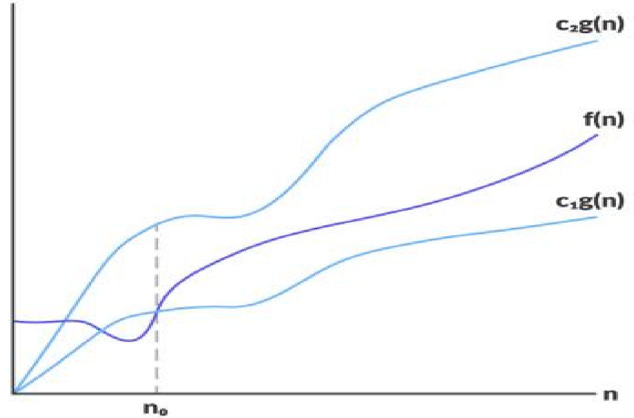
Omega Notation (Ω-notation)

- Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm. Omega gives the lower bound of a function.
- $\Omega(g(n))$ = { f(n): there exist positive constants c and n0 such that $0 \leq cg(n) \leq f(n)$ for all $n \geq n0$ }



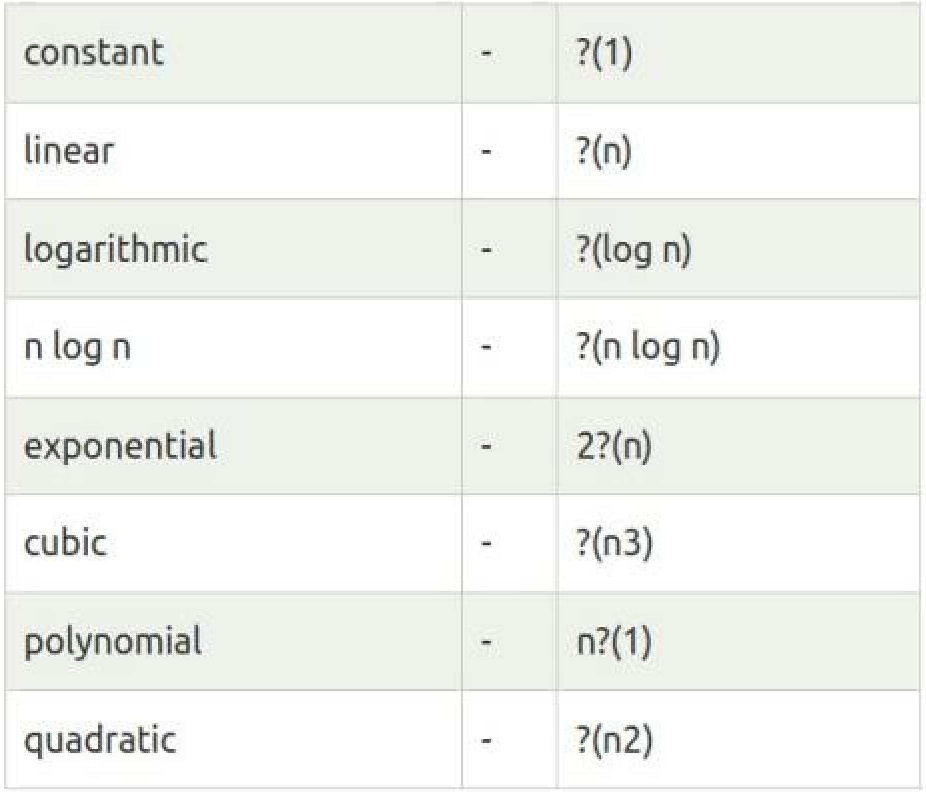
Theta Notation (Θ-notation)

- Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the average-case complexity of an algorithm. Theta bounds the function within constants factors
- $\Theta(g(n)) = \{ f(n)$: there exist positive constants c1, c2 and n0 such that $0 \le c1g(n) \le f(n) \le c2g(n)$ for all $n \ge n0 \}$



f(n) = Θ(g(n))

Common Asymptotic Notations

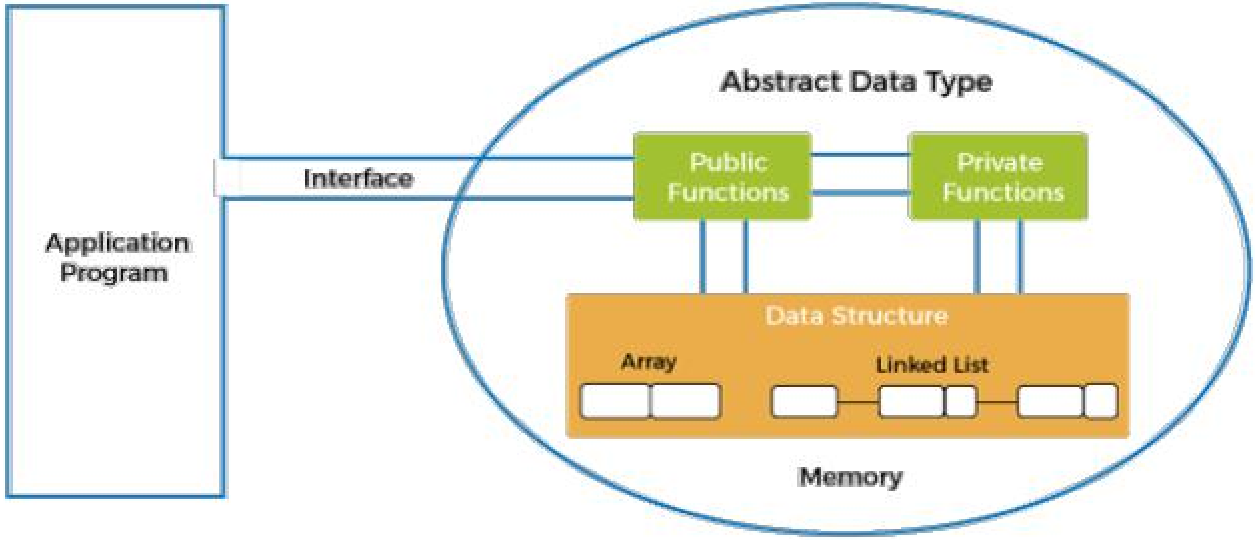


constant	-	?(1)
linear	-	?(n)
logarithmic	-	?(log n)
n log n	-	?(n log n)
exponential	-	2?(n)
cubic	-	?(n3)
polynomial	-	n?(1)
quadratic	-	?(n2)

Abstract Data Type (ADT)

Abstract Data type (ADT) is a type (or class) for objects whose behaviour is defined by a set of value and a set of operations.

- 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 gives an implementation-independent view.


Application Program
Interface
Abstract Data Type
Public Functions
Private Functions
Data Structure
Array
Linked List
Memory

ADTs

Some common examples of ADTs include:

– **List:** A list is a collection of ordered elements. Common operations on lists include adding and removing elements, accessing elements by index, and searching for elements.

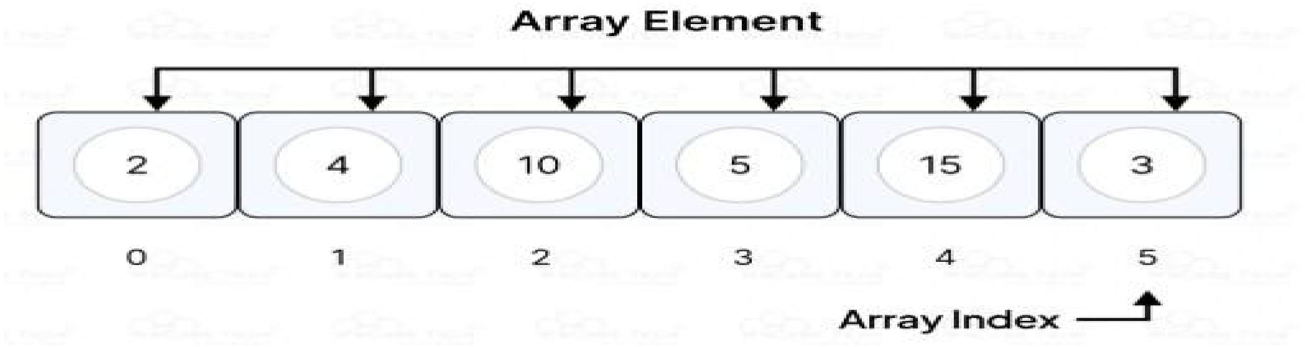
– **Set:** A set is a collection of unordered, unique elements. Common operations on sets include adding and removing elements, checking if an element is in a set, and finding the union and intersection of two sets.

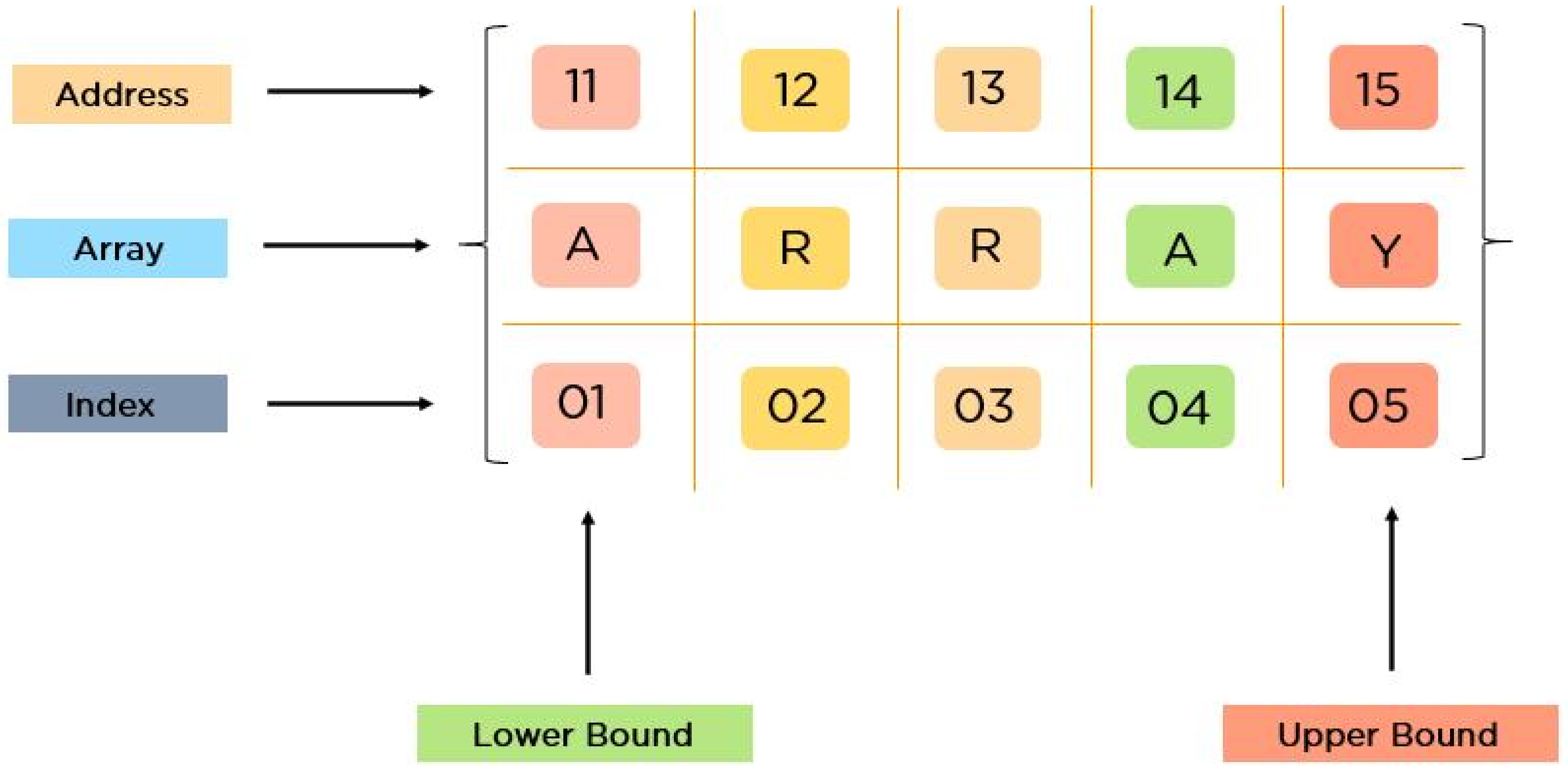
– **Stack:** A stack is a data structure that follows the last-in-first-out (LIFO) principle. Elements are added to and removed from the top of the stack.

– **Queue:** A queue is a data structure that follows the first-in-first-out (FIFO) principle. Elements are added to the back of the queue and removed from the front of the queue.

Arrays

- An array is a linear data structure that collects elements of the same data type and stores them in contiguous and adjacent memory locations. Arrays work on an index system starting from 0 to (n-1), where n is the size of the array.




Address
Array
Index
11
12
13
14
15
A
R
R
A
Y
01
02
03
04
05
Lower Bound
Upper Bound

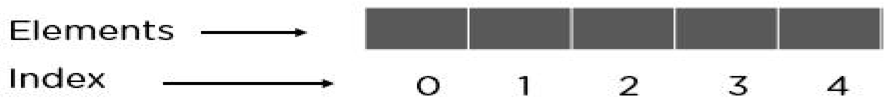
➔Types of Arrays

There are majorly two types of arrays, they are:

- **One-Dimensional Arrays**
- **Multi-Dimensional Arrays**

➔ One-Dimensional Arrays:

You can imagine a 1d array as a row, where elements are stored one after another.

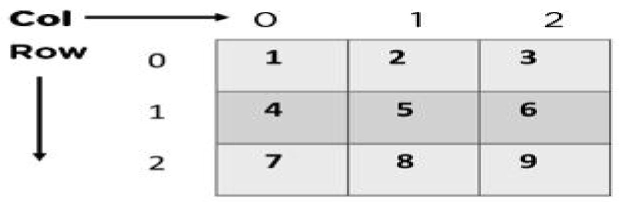


➔Multi-Dimensional Arrays:

These multi-dimensional arrays are again of two types. They are:

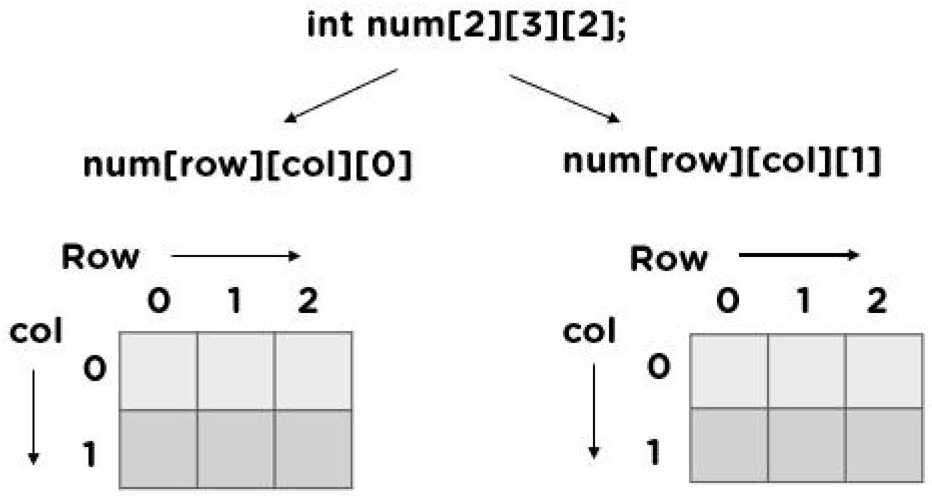
1. Two-Dimensional Arrays :

You can imagine it like a table where each cell contains elements.



2. Three-Dimensional Arrays:

You can imagine it like a cuboid made up of smaller cuboids where each cuboid can contain an element.



Why Do You Need an Array in Data Structures?

- ➢ Sorting and searching a value in an array is easier.
- ➢ Arrays are best to process multiple values quickly and easily.
- ➢ Arrays are good for storing multiple values in a single variable - In computer programming, most cases require storing a large number of data of a similar type. To store such an amount of data, we need to define a large number of variables.

Declaration

- Java arrays are declared with a type and can be initialized inline.

```
public class Main {
    public static void main(String[] args) {
        int[] ages = {14, 16, 15, 14, 16}; // Declaration and initialization
        System.out.println("Third age: " + ages[2]); // Accessing the third element
    }
}
```

Basic Operations

- ➢ **Traversal** - This operation is used to print the elements of the array.
- ➢ **Insertion** - It is used to add an element at a particular index.
- ➢ **Deletion** - It is used to delete an element from a particular index.
- ➢ **Search** - It is used to search an element using the given index or by the value.
- ➢ **Update** - It updates an element at a particular index.

➔ Traversal Operation

- This operation traverses through all the elements of an array. We use loop statements to carry this out.
- **Example:-**

```
public class ArrayTraversalForLoop {
    public static void main(String[] args) {
      int[] numbers = {10, 20, 30, 40, 50};

      System.out.print("Array elements (for loop): ");
      for (int i = 0; i < numbers.length; i++) {
        System.out.print(numbers[i] + " ");
      }
      System.out.println();
    }
  }
```

➔ Insertion Operation

In the insertion operation, we are adding one or more elements to the array.

Algorithm

```
1. Get the element value which needs to be inserted.
2. Get the position value.
3. Check whether the position value is valid or not.
4. If it is valid,
        Shift all the elements from the last index to position index by 1 position to the right.
        insert the new element in arr[position]
5. Otherwise,
        Invalid Position.
```

Insertion

```
import java.util.Arrays;

public class ArrayInsertion {

    public static int[] insertElement(int[] arr, int position, int value) {

        // Check if position is valid

       if (position < 0 || position > arr.length) {

            System.out.println("Invalid Position.");

            return arr;

        } // Create a new array with an extra space

        int[] newArr = new int[arr.length + 1];

      // Shift elements to the right from the position

        System.arraycopy(arr, 0, newArr, 0, position);

        newArr[position] = value;
```

Insertion

```
System.arraycopy(arr, position, newArr, position + 1, arr.length - position);

return newArr;

    }

 public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

   // Sample array

      int[] arr = {10, 20, 30, 40, 50};

// Get element value to insert

        System.out.print("Enter the element to insert: ");

        int value = scanner.nextInt();

// Get position value
```

Insertion

```
 System.out.print("Enter the position to insert (0 to " + arr.length + "): ");
        int position = scanner.nextInt();


        // Perform insertion
        arr = insertElement(arr, position, value);


        // Display updated array
        System.out.println("Updated Array: " + Arrays.toString(arr));


        scanner.close();
    }
}
```

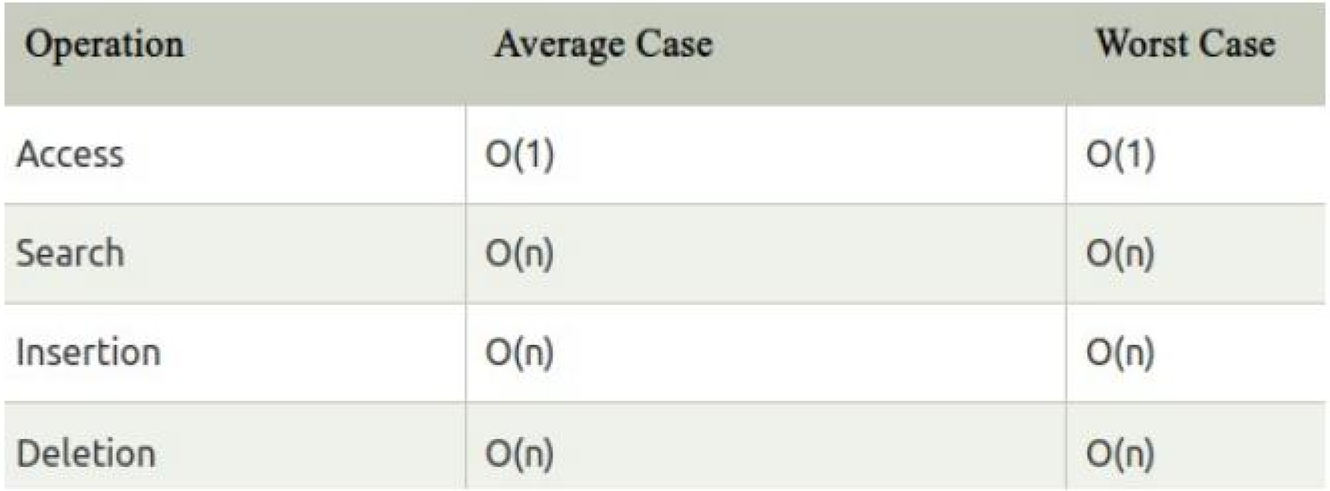
➔Deletion Operation

In this array operation, we delete an element from the particular index of an array.

Algorithm

1. Define an array.
2. Shift elements left after the target index.
3. Reduce array size.

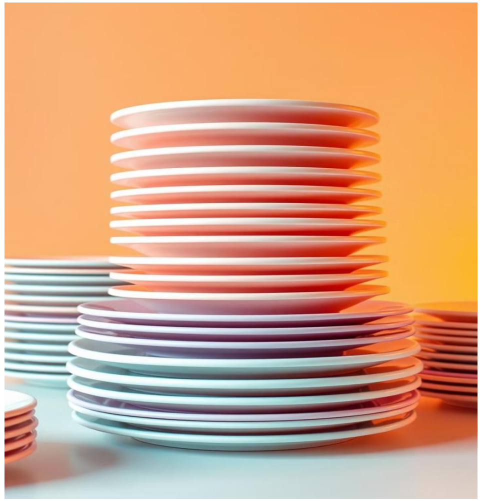
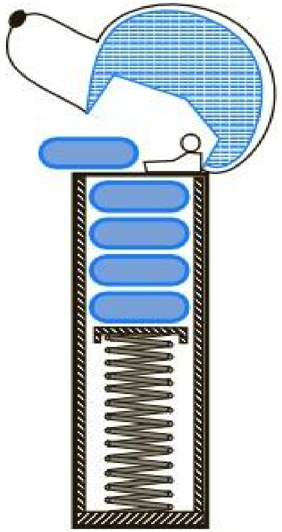
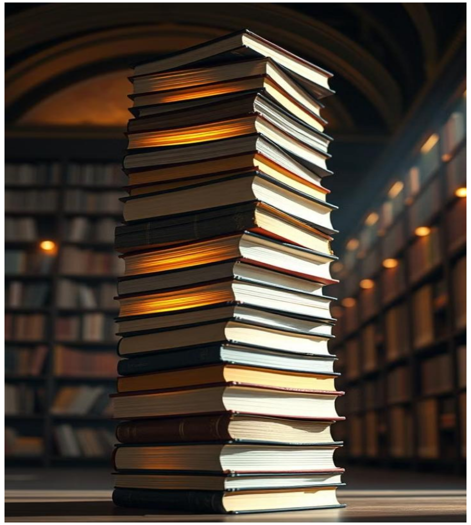
Complexity



Operation	Average Case	Worst Case
Access	O(1)	O(1)
Search	O(n)	O(n)
Insertion	O(n)	O(n)
Deletion	O(n)	O(n)

Stacks

- The name "stack" is inspired by the real-world analogy of **stacking objects**, where you place items on top of one another.A stack is a linear data structure and a collection of objects that are inserted and removed according to the last-in, first-out (LIFO) principle.
- Objects can be inserted into a stack at any time, but only the most recently inserted (that is, "last") object can be removed at any time.

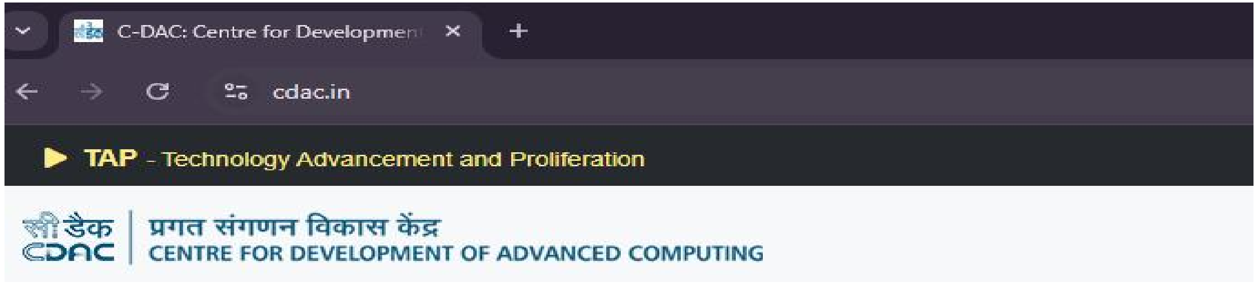
→ Example

First page: www.google.com

Second page: www.somepage.com/stack

If we press the back button on the second page, we will again go back to www.google.com result page.

Browser back button uses the Last In First Out (LIFO) principle.


C-DAC: Centre for Developmen
cdac.in
TAP - Technology Advancement and Proliferation
सी डैक CDAC
प्रगत संगणन विकास केंद्र
CENTRE FOR DEVELOPMENT OF ADVANCED COMPUTING

❖ Operations

• push(): When we insert an element in a stack then the operation is known as a push. If the stack is full then the overflow condition occurs.

• pop(): When we delete an element from the stack, the operation is known as a pop. If the stack is empty means that no element exists in the stack, this state is known as an underflow state.

• isEmpty(): It determines whether the stack is empty or not.

• isFull(): It determines whether the stack is full or not.'

• peek(): It returns the element at the given position.

• count(): It returns the total number of elements available in a stack.

• change(): It changes the element at the given position.

• display(): It prints all the elements available in the stack.

❖ Push Operation

To push data into the stack,

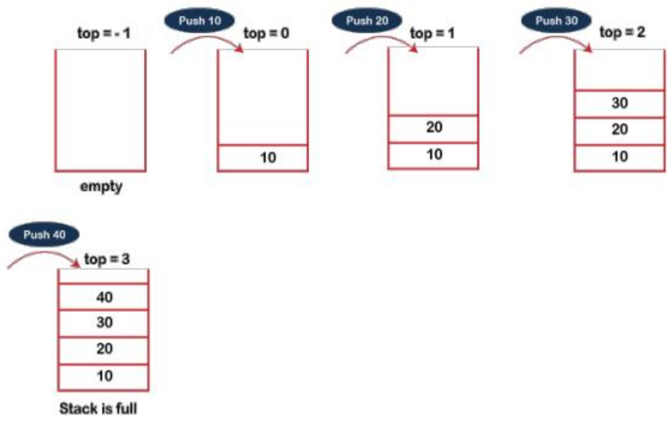
Check if the **stack is full**.

If its full

we can't insert data.

Otherwise

increment the top variable by 1 and push the data.



❖ Pop Operation

To pop the data from the stack,

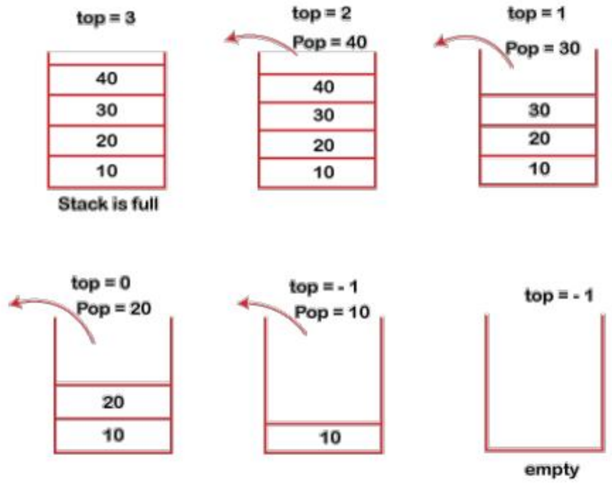
Check if the stack is **empty**

if it's **empty**

We can't pop an element from the empty stack. So if **top == -1**, we should not do anything.

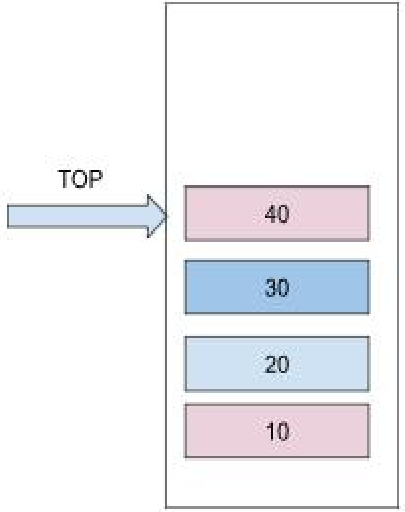
Otherwise

Pop the element and **decrement the top value by 1**.



❖ Peek Operation

The peek or top operation retrieves the top element of the stack without removing it. This allows you to see what is at the top of the stack without modifying its content.

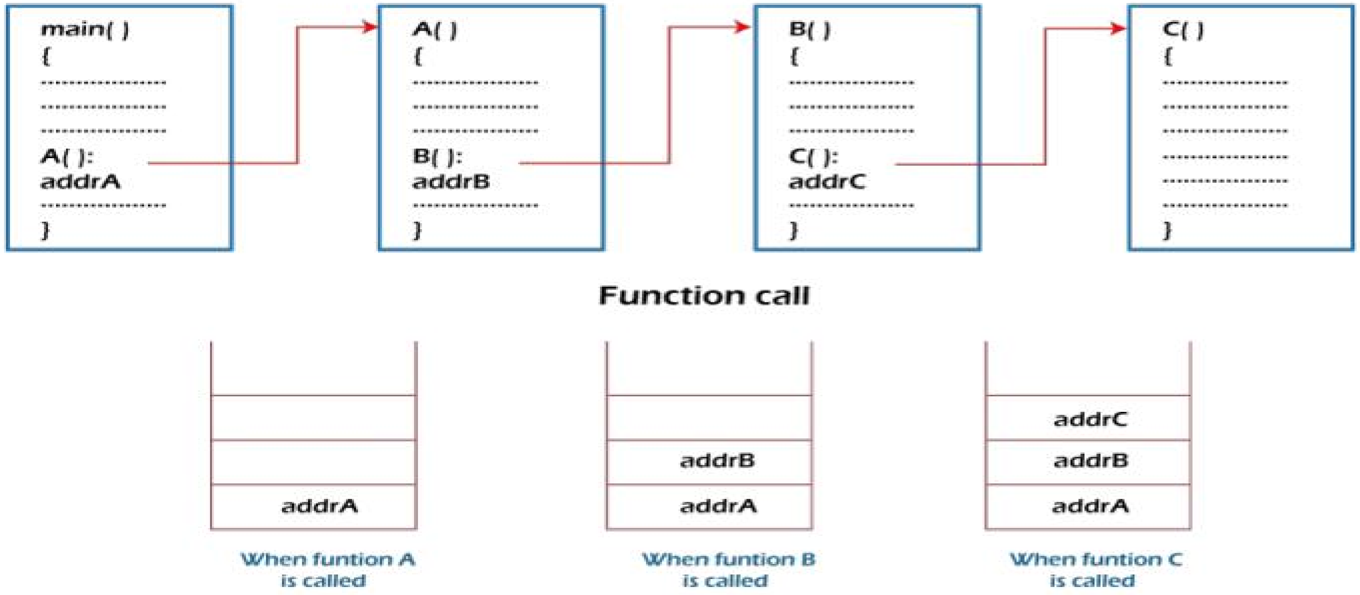


Stack Applications

- Evaluation of Arithmetic Expressions
- Backtracking
- Delimiter Checking
- Reverse a Data
- Processing Function Calls

Function Call

Different states of stack



Evaluation of Arithmetic Expressions

- **Infix Notation**: Each operator is placed between the operands in the infix notation, which is a convenient manner of constructing an expression.
  - Example 1 : A + B
- **Prefix Notation**: The operator is placed before the operands in the prefix notation. This system was created by a Polish mathematician, and is thus known as polish notation.
  - Example 1 : +AB
- **Postfix Notation** : The operator is placed after the operands in postfix notation. This notation is known as Reverse Polish notation since it is the inverse of Polish notation.
  - Example 1 : AB+

Conversion of Infix to Postfix

Rules for the conversion from infix to postfix expression

Initially we have a infix expression given to us to convert to postfix notation. The infix notation is parsed from left to right, and then converted to postfix.

1. If we have an opening parenthesis "(", we push it into the stack.
2. If we have an operand, we append it to our postfix expression.
3. If we have a closing parenthesis ")" we keep popping out elements from the top of the stack and append them to our postfix expression until we encounter an opening parenthesis. We pop out the left parenthesis without appending it.

Conversion of Infix to Postfix

4.If we encounter an operator:-

4.1. If the operator has higher precedence than the one on top of the stack (We can compare), we push it in the stack.

4.2. If the operator has lower or equal precedence than the one on top of the stack, we keep popping out and appending it to the postfix expression.

5. When the last token of infix expression has been scanned, we pop the remaining elements from stack and append them to our postfix expression.*

Infix expression: K + L - M*N + (O^P) * W/U/V * T + Q

Conversion of Infix to Postfix

In the conversion of infix to postfix, what should you do if an operator has lower or equal precedence than the one on top of the stack?

1. Push it onto the stack

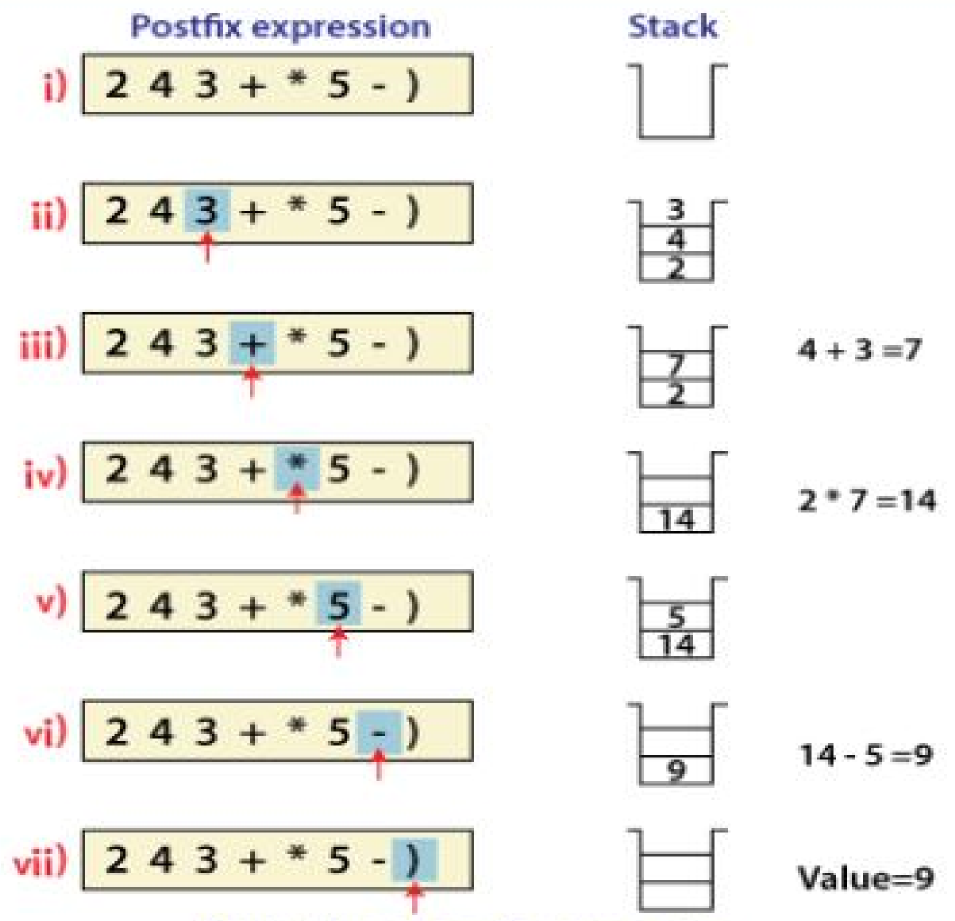
2.Append it to the postfix expression immediately

3.Pop the stack and append to the postfix expression

Evaluation of Postfix Expression

A postfix expression can be evaluated using the Stack data structure. To evaluate a postfix expression using Stack data structure we can use the following steps...

1. Read all the symbols one by one from left to right in the given Postfix Expression
2. If the reading symbol is operand, then push it on to the Stack.
3. If the reading symbol is operator (+ , - , * , / etc.,), then perform TWO pop operations and store the two popped oparands in two different variables (operand1 and operand2). Then perform reading symbol operation using operand1 and operand2 and push result back on to the Stack.
4. Finally! perform a pop operation and display the popped value as final result.



Evaluation of postfix expression

Balanced parentheses

Balanced parentheses means that each opening symbol has a corresponding closing symbol and the pairs of parentheses are properly nested.

Problem Statement

Given an input expression string of length n consisting of three types of parentheses – {,},(,),[,].Check for balanced parentheses in the expression (well-formedness) using Stack. Parentheses are balanced if:

1. The same kind of parentheses are used to close any open ones.
2. The proper sequence must be used to close any open parentheses.

Balanced parentheses

Example

1. Input: expression = ~([]){}[[(){}]{}]

   Output: Yes, Balanced

2. Input: expression = [(]))

   Output: No, Not Balanced

3. It’s real life application is during the compilation of any code

Algorithm

- Declare an empty stack.
- Now start traversing the input string.
- If you come across an opening bracket while traversing the string, add it to the stack.
- Else the current character is a closing bracket. In this case, check to see if the top element of the stack is of the corresponding opening type and if it is then pop the top element from the stack and if not, then return false.
- If the stack is empty after traversing the string, return true; otherwise, return false.

Exercise 1

Suppose an initially empty stack S has performed a total of 26 push operations, 13 top operations, and 9 pop operations, 1 of which returned null to indicate an empty stack. What is the current size of S?

Exercise 1

- Push operations: 26 items were pushed onto the stack, so the stack has 26 items initially.
- Pop operations: 9 pop operations were performed, but 1 of them returned null, meaning the stack was empty during that operation. So, only 8 pop operations were successful, and they removed 8 items from the stack.
- Top operations: The 13 top operations do not affect the size of the stack, as they only check the top item without removing it.
- - After 26 push operations, the stack has 26 items.
- - After 8 successful pop operations, the stack loses 8 items.
- So, the current size of the stack is: 26 - 8 = 18

Exercise 2

What values are returned during the following series of stack operations, if executed upon an initially empty stack?

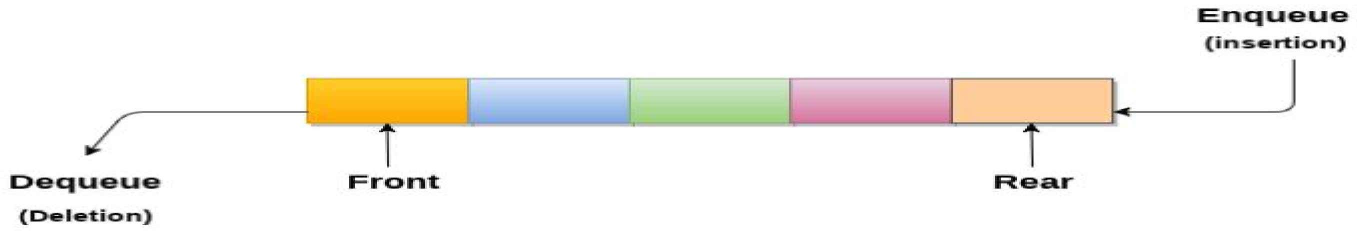
- push(5), push(3), pop(), push(2), push(8), pop(), pop(), push(9), push(1), pop(), push(7), push(6), pop(), pop(), push(4), pop(), pop()

Exercise 3

- Suppose Mustafa has picked three distinct integers and placed them into a stack D in random order.
- Write a short, straight-line piece of pseudocode (with no loops or recursion) that uses only one comparison and only one variable x, yet that results in variable x storing the largest of Mustafa's three integers with probability 2/3.

Queue

- A queue can be defined as an ordered list which enables insert operations to be performed at one end called REAR and delete operations to be performed at another end called FRONT.
- Queue is referred to be as First In First Out list.
- For example, people waiting in line for a rail ticket form a queue.



Queue Operations

Enqueue: This operation adds an item to the back of the queue.

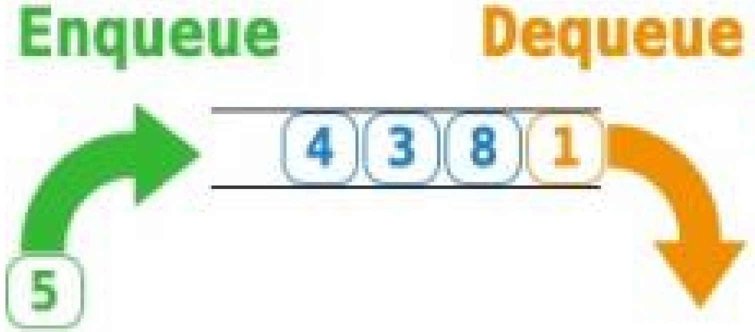
Dequeue: This operation removes an item from the front of the queue.

Peek/Front: This operation returns the item at the front of the queue without removing it.

IsEmpty: Checks if the queue is empty.

IsFull: Checks if the queue is full (applicable mainly to array-based implementations).

Queue



First **I**n,
First **O**ut

Exercise 1

- Suppose an initially empty queue Q has performed a total of 32 enqueue operations, 10 first operations, and 15 dequeue operations, 5 of which returned null to indicate an empty queue
- What is the current size of Q?

Exercise 1

- 1. Enqueue operations: 32 enqueue operations add 32 elements to the queue.
- 2. Dequeue operations: 15 dequeue operations attempt to remove elements from the queue.
- Out of these, 5 dequeue operations returned null, which means the queue was empty at the time, and no element was removed during these operations.
- Therefore, only \(15 - 5 = 10 \) dequeue operations successfully removed elements.
- 3. First operations: The first operation just returns the first element in the queue without removing it, so the 10 first operations do not affect the size of the queue.
- Thus, the number of elements currently in the queue is the number of enqueue operations minus the number of successful dequeue operations: Size of Q = 32 - 10 = 22

Exercise 2

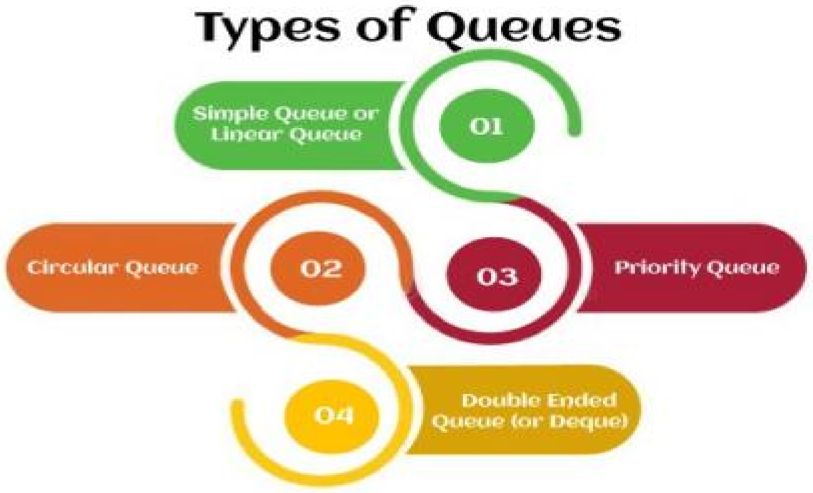
What values are returned during the following sequence of queue operations, if executed on an initially empty queue?

- enqueue(5), enqueue(3), dequeue(), enqueue(2), enqueue(8), dequeue(), dequeue(), enqueue(9), enqueue(1), dequeue(), enqueue(7), enqueue(6), dequeue(), dequeue(), enqueue(4), dequeue(), dequeue().

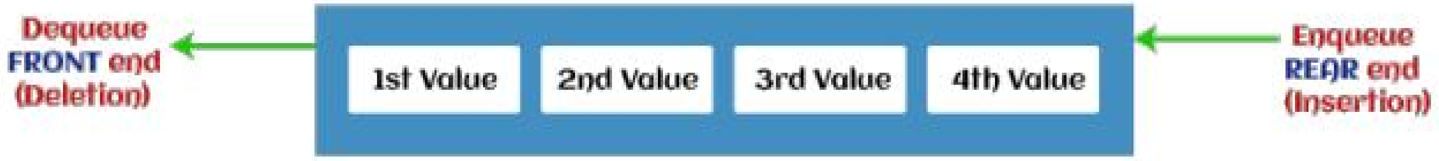
Exercise 3

- What values are returned during the following sequence of deque ADT operations, on an initially empty deque?
- addFirst(3), addLast(8), addLast(9), addFirst(1), last( ), isEmpty( ), addFirst(2), removeLast( ), addLast(7), first( ), last( ), addLast(4), size( ), removeFirst( ), removeFirst( ).

Queue: Types


Types of Queues
Simple Queue or Linear Queue
01
Circular Queue
02
03
Priority Queue
04
Double Ended Queue (or Deque)

Simple Queue



Operations

addFront(element): Adds an element to the front of the deque.

addRear(element): Adds an element to the rear of the deque.

removeFront(): Removes the front element from the deque.

removeRear(): Removes the rear element from the deque.

peekFront(): Returns the front element without removing it.

peekRear(): Returns the rear element without removing it.

isEmpty(): Checks if the deque is empty.

size(): Returns the number of elements in the deque.

Limitations of Queue

- ➢ A queue is not readily searchable: You might have to maintain another queue to store the dequeued elements in search of the wanted element.

- ➢ Traversal possible only once: The front most element needs to be dequeued to access the element behind it, this happens throughout the queue while traversing through it. In this process, the queue becomes empty.

- ➢ Memory Wastage: In a Static Queue, the array's size determines the queue capacity, the space occupied by the array remains the same, no matter how many elements are in the queue.

Applications of Queue

Job Scheduling

Multiprogramming

Traffic Management Systems

Data Buffers in Networking