U ▪
S T

DATA STRUCTURES AND ALGORITHMS

BATCH 3

INTRODUCTION

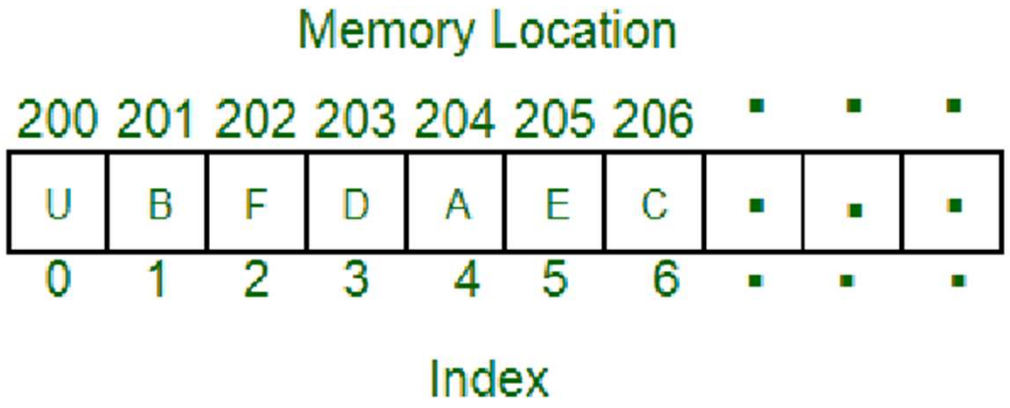
- A data structure is a collection of data values and the relationships between them.
- Data structures allow programs to store and process data effectively.
- There are many different data structures, each with its own advantages and disadvantages.
- Some of the most common data structures are arrays, lists, trees, and graphs.

WHAT IS DATA STRUCTURE

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently.
- Examples of Data Structures are arrays, Linked List, Stack, Queue, etc.
- Data Structures are widely used in almost every aspect of Computer Science i.e. Operating System, Compiler Design, Artificial intelligence, Graphics and many more.
- Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way.
- It plays a vital role in enhancing the performance of a software or a program as the main function of the software is to store and retrieve the user's data as fast as possible

DATA STRUCTURE

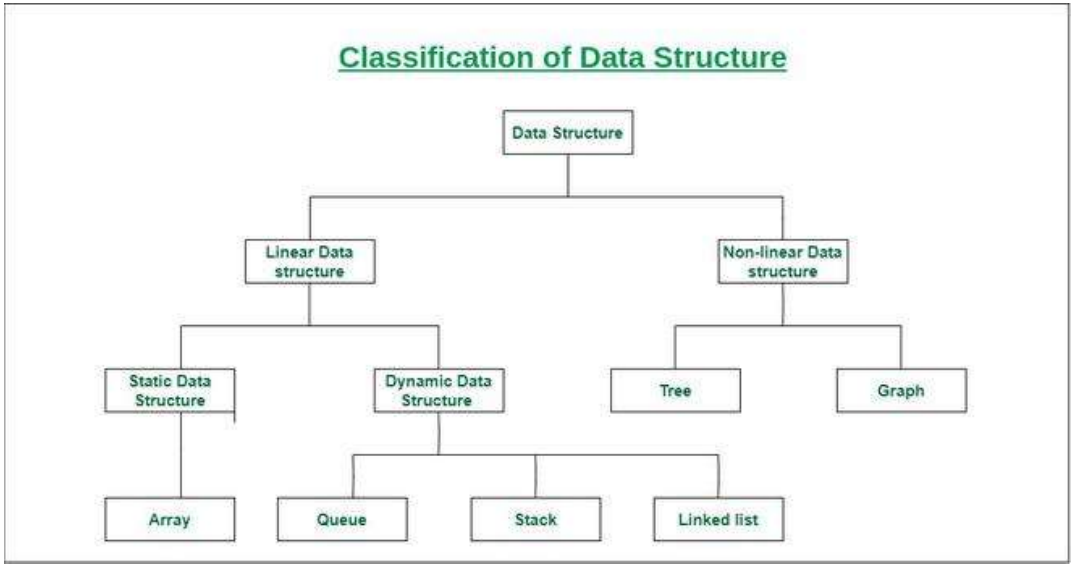
- A data structure is a particular way of organizing data in a computer so that it can be used effectively.
- For example we can store a list of items having the same data-type using



TYPES

Basically, data structures are divided into two categories:

- Linear data structure
- Non-linear data structure



TYPES

1.Linear data structure

- The data items are arranged in sequential order, one after the other.
- All the items are present on the single layer.
- It can be traversed on a single run. That is, if we start from the first element, we can traverse all the elements sequentially in a single pass.
- The memory utilization is not efficient.
- The time complexity increase with the data size.
- Example:

Array Data Structure, Stack Data Structure, Queue Data Structure, Linked List Data Structure

2.Non-linear data structure

- The data items are arranged in non-sequential order (hierarchical manner, where one element will be connected to one or more elements).
- The data items are present at different layers.
- It requires multiple runs. That is, if we start from the first element it might not be possible to traverse all the elements in a single pass.
- Different structures utilize memory in different efficient ways depending on the need.
- Time complexity remains the same.
- Example:

Graph Data Structure, Trees Data Structure, Map Data Structure

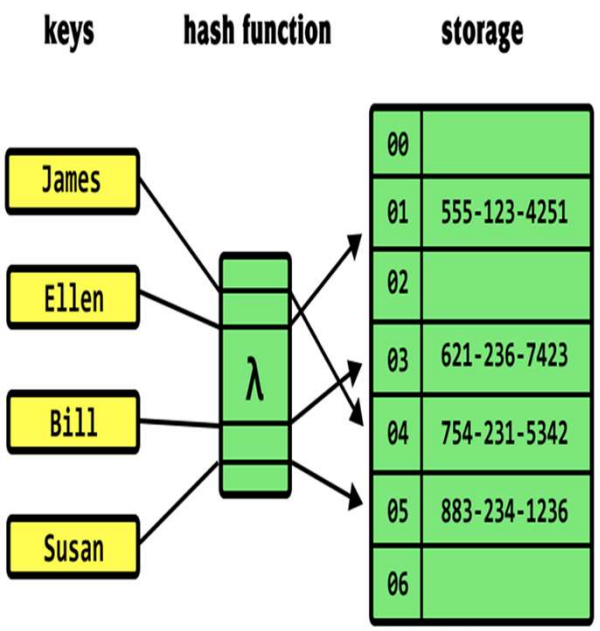
CASE STUDY

Social Media Application

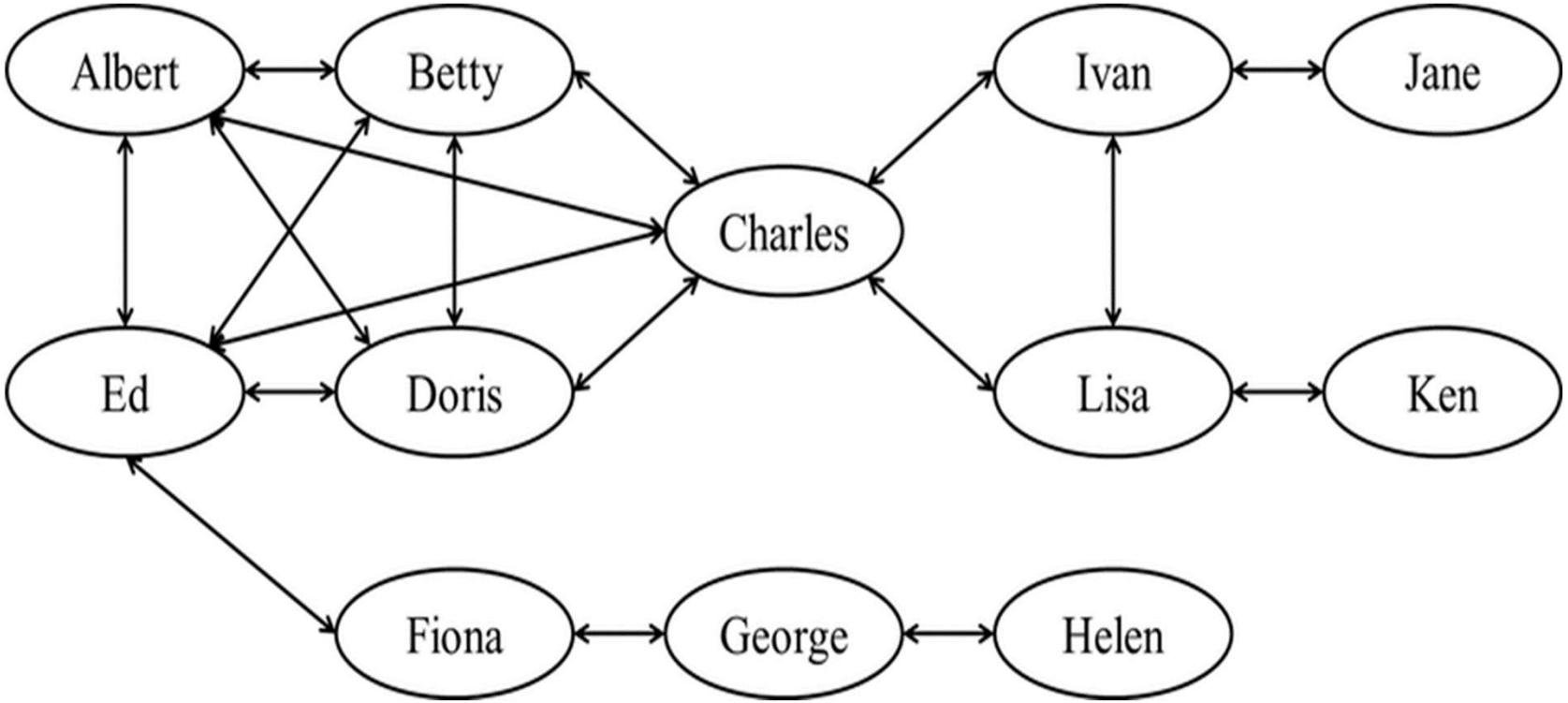
- Suppose we want to develop a social media application that allows users to connect with friends, post updates, and share media.
- One of the key features of the application is the ability to search for other users and their posts.
- To implement this feature, we need to use data structures to efficiently store and retrieve data.

Hash Table To Store User Profiles

- We can use a hash table to store user profiles, where each profile is identified by a unique username.
- The hash function will map each username to a unique index in the hash table, allowing for constant time access to a user's profile.
- We can also use a hash table to store posts, where each post is identified by a unique post ID.



Using Graphs To Connect User Profile



Binary tree to store media files

To store media files such as images and videos, we can use a binary tree.

Each node in the tree represents a media file, and the left and right child nodes represent smaller and larger media files respectively. This allows for efficient searching and retrieval of media files based on their size.

To store user connections (i.e., who they are friends with), we can use a graph. Each user is represented by a vertex, and an edge between two vertices indicates that the two users are friends.

This allows for efficient searching of a user's connections and for suggesting new connections based on mutual friends.

Storing Posts Using Linked List

To store user posts, we can use a linked list.

Each node in the list represents a post, and the next pointer points to the next post in the list.

This allows for efficient adding and deleting of posts.

Arrays

- Array is a collection of elements stored at contiguous memory locations.
- Each element in an array is identified by an index, which represents its position in the array.
- They are particularly useful for tasks such as sorting and searching because they provide fast access to individual elements.

Operations in array

- Accessing elements: You can access individual elements in an array by specifying their index. This allows you to read or modify the values stored in the array.

- Inserting elements: You can insert new elements into an array at a specific index. This involves shifting all the subsequent elements to make room for the new element.

- Deleting elements: You can remove elements from an array by specifying their index. This involves shifting all the subsequent elements to fill the gap left by the deleted element.

- Sorting elements: You can sort the elements in an array in ascending or descending order based on their value.

- Searching elements: You can search an array for a specific element or value. This can be done using linear search or binary search algorithms.

- Traversing elements: You can traverse all the elements in an array and perform some operation on each element. This can be done using loops or other iteration constructs.

- Copying elements: You can copy the elements of an array into another array or data structure.

- Merging arrays: You can combine the elements of two or more arrays into a single array.

Internal working of Arrays

- An array is typically implemented as a contiguous block of memory that stores a fixed-size collection of elements of the same data type. Each element is accessed using an index that represents the offset of the element from the start of the array.

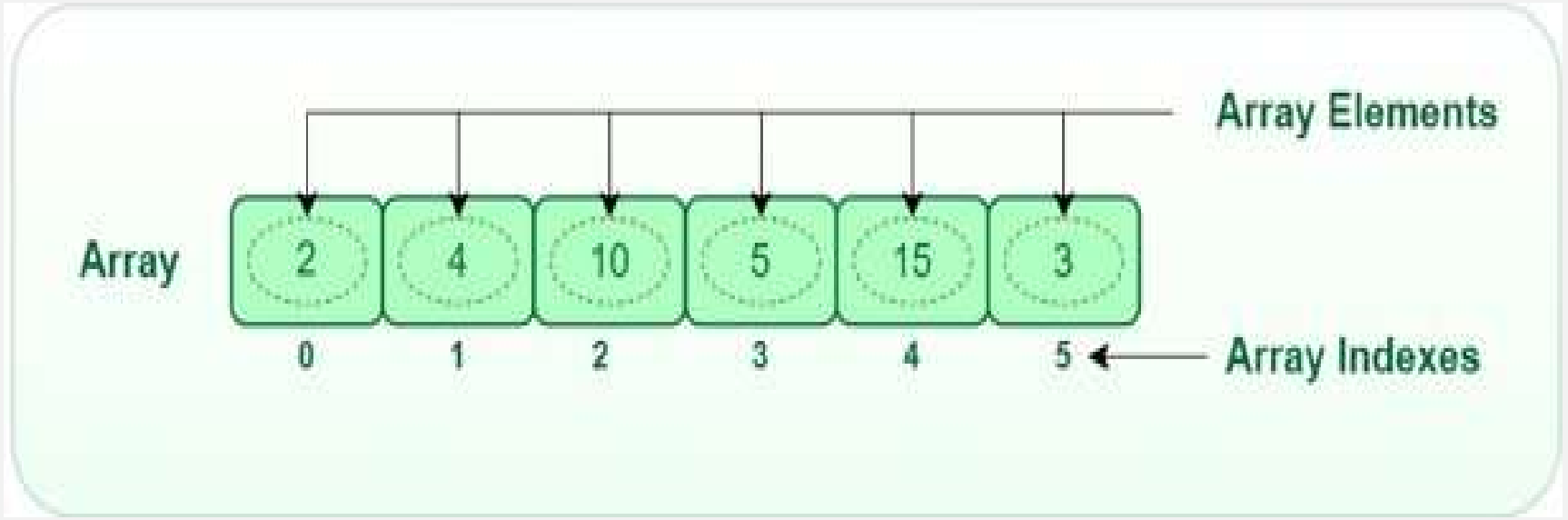
- When an array is created, a block of memory is allocated to store the elements of the array. The size of the block of memory is determined by multiplying the size of each element by the number of elements in the array.

- When an element is inserted into an array, all of the subsequent elements must be shifted to make room for the new element. This can be a time-consuming operation, especially for large arrays with many elements.

- when an element is deleted from an array, all of the subsequent elements must be shifted to fill the gap left by the deleted element.

- When accessing an element in an array, the index is used to calculate the offset of the element from the start of the array.

- Arrays provide constant-time access to individual elements, making them an efficient data structure for many applications.



Types of Arrays

- One-dimensional array: This is the simplest type of array, where the elements are arranged in a single row or a single column.

- Two-dimensional array: In a two-dimensional array, the elements are arranged in rows and columns, creating a grid-like structure.

- Multi-dimensional array: A multi-dimensional array has more than two dimensions and is used to store data in a higher-dimensional space.

Advantages of arrays

- Fast access: Arrays provide fast access to individual elements because they are stored in contiguous memory locations.
- Efficient memory usage: Arrays use memory efficiently because they store data in a contiguous block of memory.
- Random access: Arrays provide random access to elements, which means that you can access any element in the array directly using its index.
- Easy to manipulate: Arrays are easy to manipulate, and many programming languages provide built-in functions for performing common array operations, such as sorting, searching, and merging.

Disadvantages of arrays

- Fixed size: In most programming languages, arrays have a fixed size, which means that you cannot add or remove elements once the array has been created.

- Memory wastage: In cases where the size of the array is not known in advance, arrays can lead to memory wastage, since you may need to allocate more memory than you actually need.

- Indexing issues: Indexing errors can occur if you try to access an index that is out of bounds, or if you use the wrong data type for the index.

- Lack of flexibility: Since arrays have a fixed size, they are not very flexible, and cannot be easily adapted to changing data requirements or complex data structures.

Applications of Arrays

- Implementing sorting and searching algorithms: Many sorting and searching algorithms rely on arrays to store and manipulate data.

- Storing and manipulating collections of data: Arrays are often used to store and manipulate collections of data, such as lists of numbers, strings, or objects.

- Implementing hash tables and associative arrays: Hash tables and associative arrays use arrays to store and retrieve key-value pairs, which are commonly used in database applications and data processing.

- Implementing stacks and queues: Stacks and queues, which are used to implement algorithms and data structures such as depth-first search and breadth-first search, can be implemented using arrays.

One Dimensional Array

A one-dimensional array in Java is a collection of similar types of elements stored at contiguous memory locations. The data is stored in a continuous manner, which makes operations like search, delete, insert etc., much easier.

- **ArrayName** is the name of the array, which you can choose as per your preference. **size** represents the number of elements you want to store in the array, and it should be a positive integer value.

```
int[] arrayName = new int[size];
```

Introduction

It is clear from the name that a one-dimensional array in java must deal with only one parameter. Entities of similar types can be stored together using one-dimensional arrays. It can store primitive data types (int, float, char, etc.) or objects.

Declaration of one-dimensional array

- **Data-type:** The data type determines the data type of each element present in the array-like char, int, float, objects etc.
- **Var-name:** It is the name of the reference variable which points to the array object stored in the heap memory.

[ ] : It is called subscript.

General form:

data-type var-name[];

OR

data-type[] var-name;

OR

data-type []var-name;

Construction of one-dimensional array in Java

JVM(**Java Virtual Machine** that drives the Java Code, converts Java bytecode into machine language) will assign five contiguous memory locations to store the values assigned to the array.

JVM stores the elements in contiguous memory locations. Each element position can be easily accessed through the index number starting from 0 to n-1 where n is equal to the number of elements stored in the array.

Memory address for array

Declaration :

```
int marks[ ] = { 90, 97, 95, 99,
100 };
```

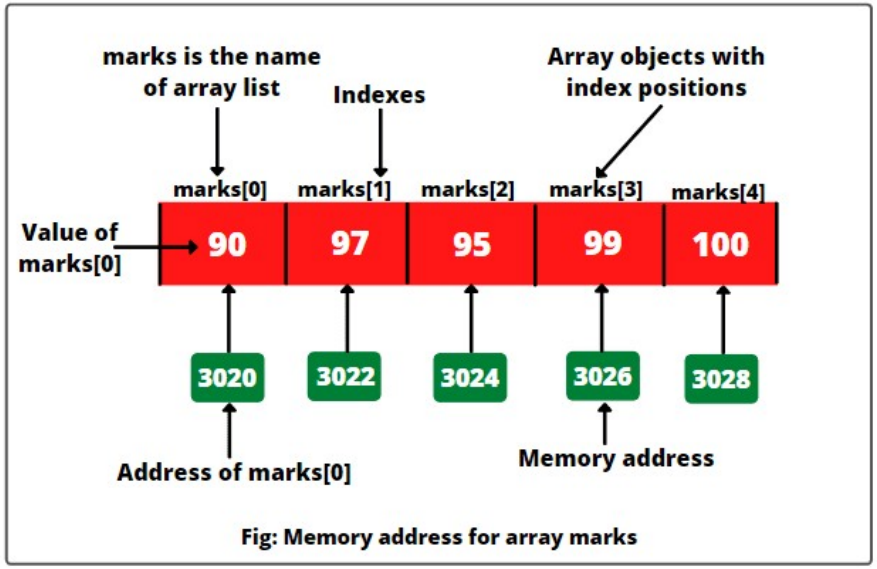
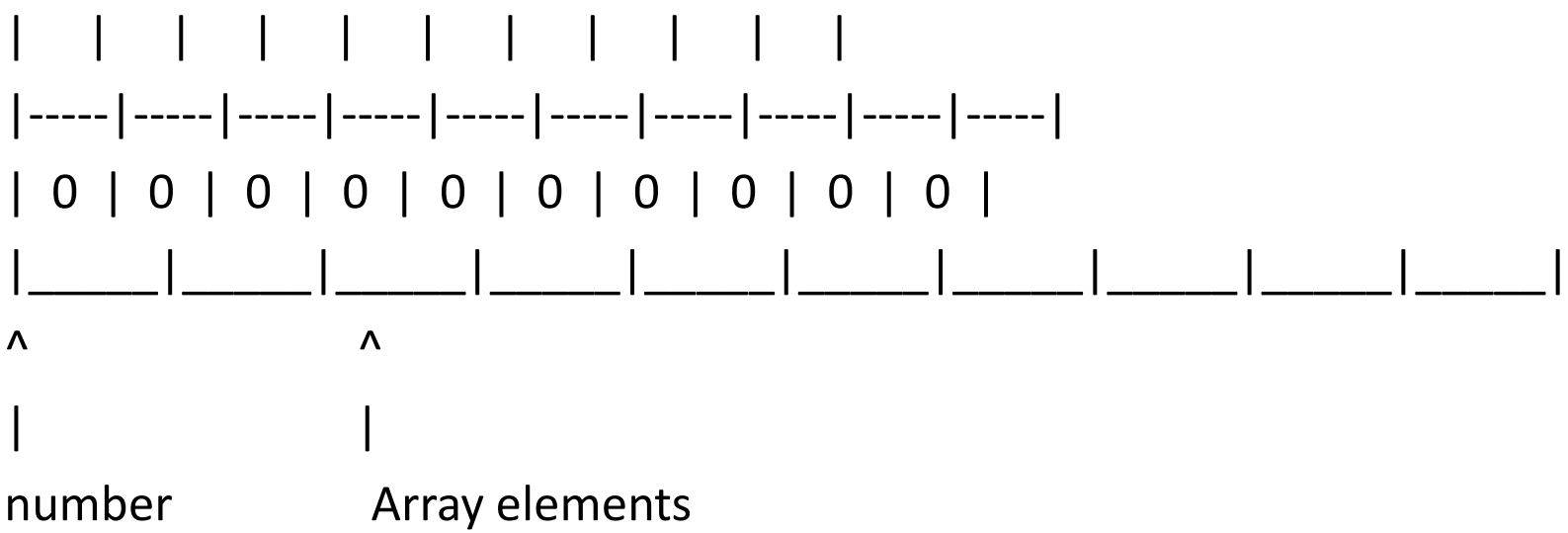


Fig: Memory address for array marks

Memory representation after construction

- JVM stores the elements in contiguous memory locations. Each element position can be easily accessed through the index number starting from 0 to n-1 where n is equal to the number of elements stored in the array.
- The second way of creating an array is by first declaring the array and then allocating the memory through the **new** keyword :
- var-name = new type[size];



- The memory representation consists of 10 contiguous blocks of memory, each representing an element of the array.
- Each block can store an integer value, initialized to 0 by default.
- The variable number is a reference to the array object, which is stored in the heap segment of the memory.
- The variable number points to the first element of the array, i.e., index 0.
- The arrow indicates that the variable number is referencing the array object, and the array elements are stored in consecutive memory locations.
- Each element in the array can be accessed using its index, e.g., number[0] for the first element, number[1] for the second element, and so on.

Advantages

- Simplicity: One-dimensional arrays are simple and easy to understand. They provide a basic way to store and access a fixed-size collection of elements of the same data type.
- Efficient memory usage: Arrays allocate contiguous memory locations for their elements, which allows for efficient memory usage and cache-friendly access patterns. This makes arrays suitable for applications that require fast and efficient data retrieval.
- Random access: Arrays allow for direct random access to any element using an index, which makes element retrieval and modification fast and efficient.
- Iteration: Arrays provide simple and efficient ways to iterate over the elements using loops, making them suitable for tasks that require sequential access to elements.

Disadvantages

- Fixed size: Arrays have a fixed size that is determined at the time of creation. Once an array is created, its size cannot be changed, which can be limiting in dynamic applications where the size of the data may change over time.
- No dynamic resizing: Unlike some other data structures, such as ArrayList in Java, arrays do not automatically resize themselves when elements are added or removed. Manual resizing and copying of elements may be required if the size of the array needs to be changed.
- Waste of memory: Arrays may waste memory if the allocated size is larger than the number of elements actually stored in the array. This can be inefficient in terms of memory usage, especially when dealing with large arrays.
- Lack of built-in functionality: Arrays in Java do not have built-in methods for common operations such as sorting, searching, or inserting elements. Additional programming effort may be required to implement such functionalities on top of arrays.

Uses of One-Dimensional Arrays

1. Grade Book: In a school or educational setting, a one-dimensional array can be used to store and manipulate grades of students. Each element in the array can represent the grade of a student for a particular subject or assignment. The array can be used to calculate the average, find the highest or lowest grade, and perform other statistical operations on the grades.
2. Music Playlist: a one-dimensional array can be used to store a playlist of songs. Each element in the array can represent a song in the playlist, and the array can be used to manage the order of songs, add or remove songs, and play songs sequentially.
3. Usernames and Passwords: In a login system, a one-dimensional array can be used to store usernames and passwords of registered users. Each element in the array can represent a username and its corresponding password. The array can be used to verify user credentials during the login process and manage user accounts.

Real-time examples

1. Stack.
2. Queue.
3. Dynamic Arrays.
4. Hash Tables.
5. Sorting Algorithms.

Two Dimensional Arrays

- In Java, a two-dimensional array is a data structure that allows you to store a matrix of elements, where each element is identified by two indices instead of one. You can think of a two-dimensional array as a table with rows and columns, where each cell holds a certain value.

- To declare a two-dimensional array in Java, you use the following **syntax**:

  type[][] arrayName = new type[rowSize][columnSize];

- Here, "type" is the data type of the array's elements (e.g., int, float, String), "arrayName" is the name you give to the array, "rowSize" is the number of rows in the array, and "columnSize" is the number of columns in the array.

Introduction

- Two-dimensional array, is a type of data structure that stores elements in a table-like format with rows and columns. It is a matrix-like structure, where each element is identified by its row and column indices.
- A 2D array is often used to represent matrices, tables, or grids in programming applications. For example, a 2D array can be used to represent a chessboard in a game, a spreadsheet in an application, or an image in image processing.
- A 2D array is often used to represent matrices, tables, or grids in programming applications. For example, a 2D array can be used to represent a chessboard in a game, a spreadsheet in an application, or an image in image processing.

- To access elements in a 2D array, you use the row and column indices of the element. The first index corresponds to the row number, and the second index corresponds to the column number. The syntax for accessing an element in a 2D array is as follows:

```
arrayName[rowIndex][colIndex]
```

- Here, "arrayName" is the name of the array, "rowIndex" is the index of the row where the element is located (starting from 0), and "colIndex" is the index of the column where the element is located (starting from 0).

- You can also use nested loops to iterate over all the elements in a 2D array. For example, to print out all the elements in a 2D array named "myArray", you can use the following code:

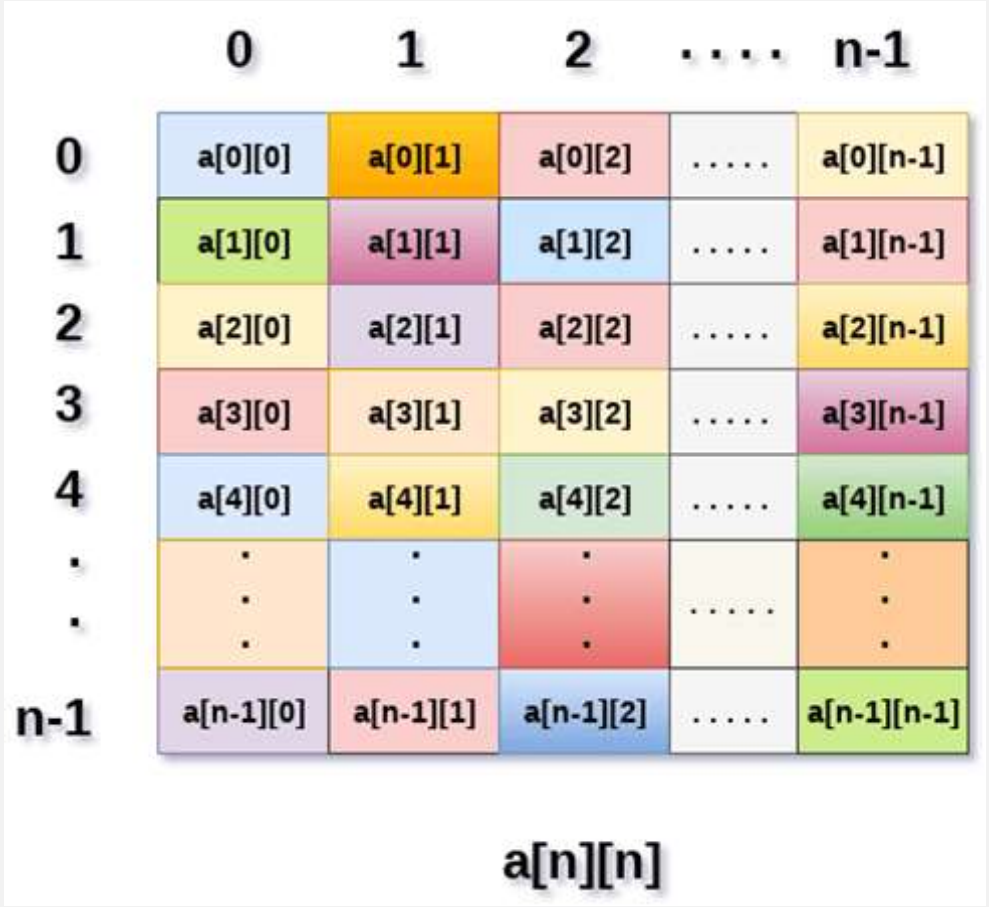
```
for (int i = 0; i < numRows; i++) {
    for (int j = 0; j < numCols; j++) {
        System.out.print(myArray[i][j] + " ");
    }
    System.out.println();
}
```

How to declare 2D Array

- In Java, you can declare a 2D array using the following syntax:

  datatype[][] arrayName = new datatype[rows][columns];

Here, '**datatype**' represents the data type of the elements in the array, '**arrayName**' is the name you choose for the array, '**rows**' is the number of rows in the array, and '**columns**' is the number of columns in the array.



- Above image shows the two dimensional array, the elements are organized in the form of rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

- For example, to declare a 2D integer array named myArray with 3 rows and 4 columns, you can use the following code:

```
int[][] myArray = new int[3][4];
```

- This will create a 2D integer array with 3 rows and 4 columns, where each element is initialized to the default value of 0.

- You can also initialize a 2D array with values by using an array literal. Here's an example:

```
int[][] myArray = {{1, 2, 3}, {4, 5, 6}, {7, 8, 9}};
```

- This will create a 2D integer array with 3 rows and 3 columns, where the first row contains the values 1, 2, and 3, the second row contains the values 4, 5, and 6, and the third row contains the values 7, 8, and 9.

Once you have declared a 2D array, you can access and modify its elements using the row and column indices. For example, to access the element at row 1 and column 2 of 'myArray', you can use the following code:

```
int element = myArray[1][2];
```

This will retrieve the value 6, which is located in the second row and third column of the array.

Note that 2D arrays can be initialized with any number of rows and columns, and the rows and columns do not have to be the same size.

Uses of Two-Dimensional Arrays

- 2D arrays are used in various programming applications where data needs to be organized in a matrix or grid-like structure.

  Here are some common applications of 2D arrays:

1. Game Development: In many games, a 2D array is used to represent a game board or game world. For example, in a chess game, a 2D array can be used to represent the chessboard.
2. Image Processing: In image processing, a 2D array is often used to represent an image, where each element in the array represents a pixel in the image.

1. Spreadsheet Applications: Spreadsheet applications like Microsoft Excel or Google Sheets use a 2D array to represent the cells in a spreadsheet.
2. Data Mining and Analysis: In data mining and analysis, a 2D array can be used to represent a dataset, where each row represents a sample, and each column represents a feature.
3. Graphical User Interfaces (GUIs): In GUIs, a 2D array can be used to represent a grid of buttons, checkboxes, or other GUI components.

Time Complexity

- Accessing a specific element in a 2D array takes constant time O(1), as the JVM can directly calculate the memory address of the element using its row and column index. For example, to access the element in the second row and third column, we can use arr[1][2], which takes constant time.
- Traversing the entire 2D array using nested loops takes O(n^2) time, where n is the number of elements in the array. In this case, n = 3 x 4 = 12, so the time complexity of traversing the entire array would be O(12^2) = O(144).

```
// Accessing a specific element in a 2D array
int element = arr[1][2]; // O(1)

// Traversing the entire 2D array using nested loops
for (int i = 0; i < arr.length; i++) {
  for (int j = 0; j < arr[i].length; j++) {
    int value = arr[i][j];
    // Do something with value
  }
} // O(n^2)
```

Space Complexity

- The space complexity of a 2D array is proportional to the product of its dimensions, or O(n * m), where n is the number of rows and m is the number of columns. In this case, the space complexity would be O(3 * 4) = O(12), as the array has 3 rows and 4 columns.
- The space complexity also depends on the size of the data type used to store the elements in the array. For example, if we used double instead of int, the space complexity would be O(3 * 4 * 8) = O(96), as each double element takes up 8 bytes of memory.

```
// Declaring a 2D array of size 3 x 4
    int[][] arr = new int[3][4]; // O(n * m)
```

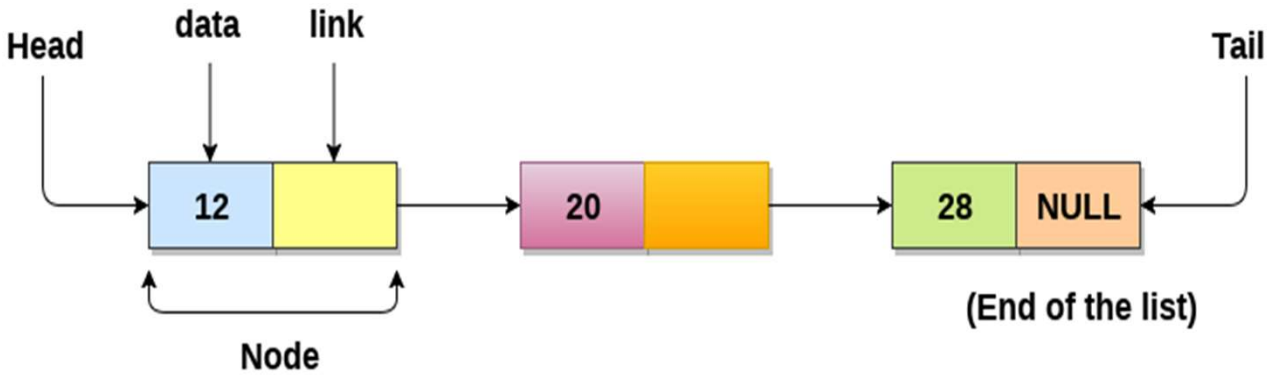
LINKED LIST

LINKED LIST

• Linked List can be defined as collection of objects called **nodes** that are randomly stored in the memory.

• A node contains two fields i.e data stored at that particular address and the pointer which contains the address of the next node in the memory.

• The last node of the list contains pointer to the null.



Uses of Linked List

- Graphs and trees: Linked lists can be used to represent nodes in a graph or tree data structure.
- Dynamic memory allocation: Linked lists are useful for dynamic memory allocation, which is when you need to allocate and deallocate memory during runtime.
- Implementing other data structures: Linked lists are a fundamental data structure that can be used to implement other data structures, such as stacks, queues, and hash tables.
- File systems: Many file systems use linked lists to keep track of files and directories.

Why use linked list

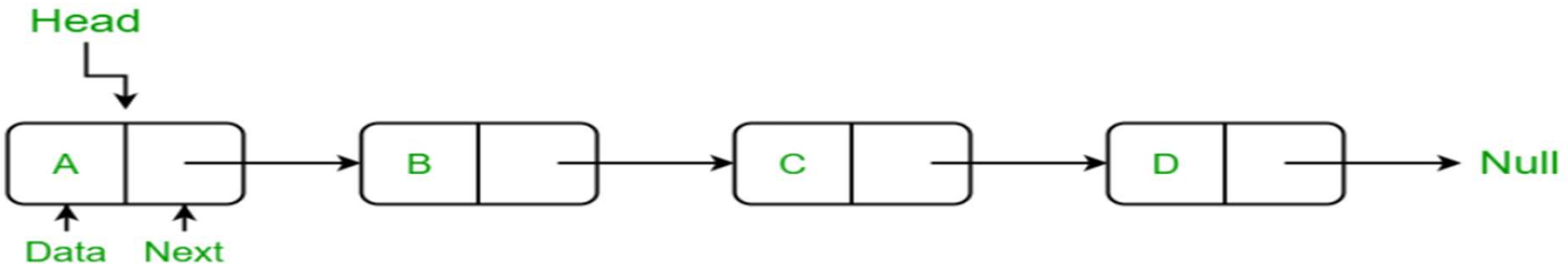
- It allocates the memory dynamically.
- All the nodes of linked list are non-contiguously stored in the memory and linked together with the help of pointers.
- Ease of Insertion/Deletion.
- Insertion at the beginning is a constant time operation and takes O(1) time.
- Memory efficiency

PURPOSES OF LINKEDLISTS

- Linked Lists can be used to implement useful data structures like stacks and queues.
- Linked Lists can be used to implement hash tables, each bucket of the hash table can be a linked list.
- Linked Lists can be used to implement graphs (Adjacency List representation of graph).
- Linked Lists can be used in a refined way in implementing different file systems in one form or another.

Singly Linked List

- A singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and each element is connected only to its next element using a pointer.
- A node in the singly linked list consist of two parts: data part and link part. Data part of the node stores actual information that is to be represented by the node while the link part of the node stores the address of its immediate successor.
- One way chain or singly linked list can be traversed only in one direction.



Operations on Singly Linked List

1. **Insertion**:

- **Insertion at the beginning (prepend)**: In this operation, a new node is added to the beginning of the list, making it the new head of the list. This operation has a time complexity of O(1).

- **Insertion at the end (append)**: In this operation, a new node is added to the end of the list, making it the new tail of the list. This operation has a time complexity of O(n), where n is the number of nodes in the list.

- **Insertion at a specific position (insert)**: In this operation, a new node is added at a specific position in the list, shifting the existing nodes as necessary. This operation has a time complexity of O(n), where n is the number of nodes in the list.

2. **Deletion**:

• **Deletion at the beginning (pop)**: In this operation, the first node in the list (the head) is removed. This operation has a time complexity of O(1).

• **Deletion at the end (delete_last)**: In this operation, the last node in the list (the tail) is removed. This operation has a time complexity of O(n), where n is the number of nodes in the list.

• **Deletion at a specific position (delete)**: In this operation, a node at a specific position in the list is removed, shifting the remaining nodes as necessary. This operation has a time complexity of O(n), where n is the number of nodes in the list.

3. **Traversal:**

- **Iterative traversal**: In this operation, each node in the list is visited in turn, starting from the head and ending at the tail. This operation has a time complexity of O(n), where n is the number of nodes in the list.

- **Recursive traversal**: In this operation, a recursive function is used to visit each node in the list, starting from the head and ending at the tail. This operation also has a time complexity of O(n).

4**. Searching:**

- Searching for a node with a specific value: In this operation, each node in the list is checked to see if it contains the specified value. This operation has a time complexity of O(n), where n is the number of nodes in the list.

Operation	Time Complexity	Space Complexity
Accessing an element by index	O(n)	O(1)
Accessing an element by value	O(n)	O(1)
Inserting an element at the beginning	O(1)	O(1)
Inserting an element at the end	O(n)	O(1)
Inserting an element in the middle	O(n)	O(1)
Removing an element from the beginning	O(1)	O(1)
Removing an element from the end	O(n)	O(1)
Removing an element from the middle	O(n)	O(1)
Traversing the list	O(n)	O(1)

Case Study: Music Player

- Singly linked lists can be used to implement a playlist feature in a music player, allowing users to create and manage their own playlists.

- In this application, each song can be represented as a node in a singly linked list. Each node contains information about the song, such as its title, artist, and file location. Each node also contains a reference to the next node in the list, allowing the songs to be linked together in a specific order.

- The playlist feature can be implemented using a separate singly linked list that contains references to the nodes in the song list. Each node in the playlist list contains a reference to a node in the song list, allowing users to create custom playlists by selecting specific songs in the song list and adding them to the playlist.

- The playlist feature can support the following operations:

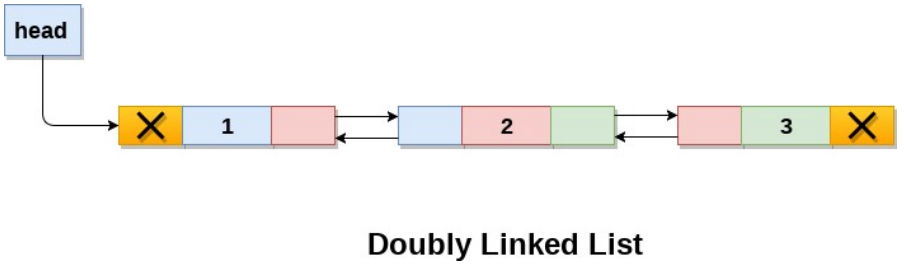
1. Insertion: adding a new song to the playlist
2. Deletion: removing a song from the playlist
3. Traversal: playing each song in the playlist in order

- Additionally, the playlist feature can be enhanced with additional features, such as the ability to shuffle the songs in the playlist or repeat the playlist. These features can be implemented by modifying the traversal algorithm to randomly select the next song in the list for shuffling, or by using a loop to repeat the playlist.

- In conclusion, the use of singly linked lists in a music player application can provide an efficient and flexible way to manage playlists, allowing users to create custom playlists and manage their music collections with ease.

Doubly linked list

- Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
- In a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer).



Doubly Linked List

Structure of a node in doubly linked list can be given as :

```
struct node
{
   struct node *prev;
   int data;
   struct node *next;
}
```

Operations on Doubly Linked Lists

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list

5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

Advantages

- Bidirectional traversal
- Efficient insertion and deletion
- Versatility
- Space efficiency
- Flexibility

Time and Space Complexity

Operation	Time Complexity	Space Complexity
Access	O(n)	O(1)
Search	O(n)	O(1)
Insertion at beginning	O(1)	O(1)
Insertion at end	O(1)	O(1)
Insertion at index i	O(n)	O(1)
Deletion at beginning	O(1)	O(1)
Deletion at end	O(1)	O(1)
Deletion at index i	O(n)	O(1)

Case Study

Suppose we have a library management system that allows librarians to add, remove, and search for books in the library. To represent the list of books, we can use a doubly linked list, where each node represents a book in the list, and each node has a reference to the previous and next nodes in the list.

Here's an example list of books represented as a doubly linked list:

HEAD <-> Book 1 <-> Book 2 <-> Book 3 <-> Book 4 <-> Book 5 <-> TAIL

In this example, the **HEAD** and **TAIL** nodes represent the start and end of the list, respectively. The **Book 1** node represents the first book in the list, and it has a reference to the **HEAD** node as its previous node, and the **Book 2** node as its next node. Similarly, the **Book 2** node has a reference to the **Book 1** node as its previous node, and the **Book 3** node as its next node, and so on.

Suppose a librarian wants to add a new book to the list between **Book 2** and **Book3**. To do this, we can create a new node for the new book and update the references of the adjacent nodes to include the new node. Here's what the list would look like after adding the new book:

HEAD <-> Book 1 <-> Book 2 <-> New Book <-> Book 3 <-> Book 4 <-> Book 5 <-> TAIL

In this updated list, the **New Book** node has a reference to the **Book 2** node as its previous node, and the **Book 3** node as its next node, and the references of the adjacent nodes have been updated accordingly.

Suppose a librarian wants to remove **Book 4** from the list. To do this, we can update the references of the adjacent nodes to skip over the **Book 4** node, and then delete the node from the list. Here's what the list would look like after removing **Book 4**:

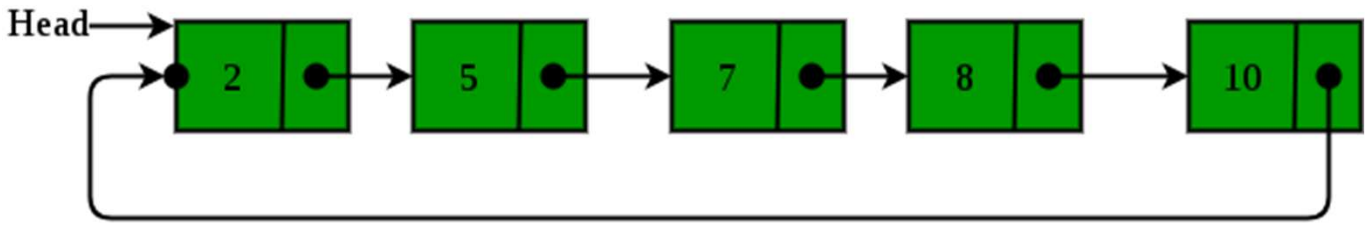
HEAD <-> Book 1 <-> Book 2 <-> New Book <-> Book 3 <-> Book 5 <-> TAIL

In this updated list, the **New Book** node has a reference to the **Book 2** node as its previous node, and the **Book 3** node as its next node, and the references of the adjacent nodes have been updated to skip over the **Book 4** node, which has been deleted from the list.

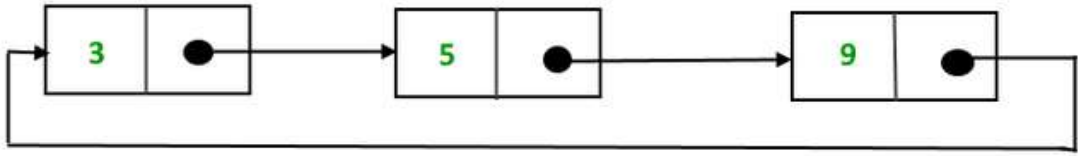
This is just a simple example, but it demonstrates how a doubly linked list can be used to represent a list of books in a library management system. By using a doubly linked list, the system can efficiently navigate the list in both forward and backward directions, as well as support adding or deleting books from any position in the list.

Circular Linked List

- Type of linked list where the last node points to the first node
- Creates a circular loop.
- The first node and the last node are connected to each other.
- There is no null at the end



Circular Linked List: 2 Types

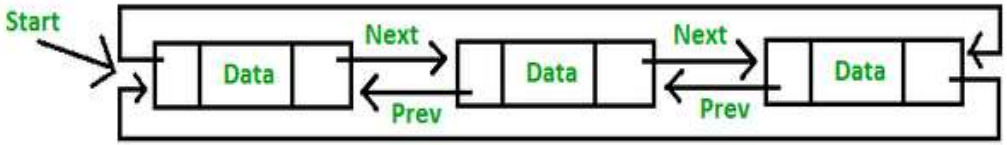


Circular Singly Linked List

- The last node of the list contains a pointer to the first node of the list.
- We traverse the circular singly linked list until we reach the same node where we started.
- Has no beginning or end.
- No null value is present in the next part of any of the nodes.

U ▪
S T

Circular Linked List: 2 Types



Circular Doubly Linked List

- Has properties of both doubly linked list and circular linked list
- Two consecutive elements are linked or connected by the previous and next pointer
- The last node points to the first node by the next pointer
- The first node points to the last node by the previous pointer

U ▪
S T

Circular Linked List: Usage

- **Dynamic memory allocation**: Nodes can be added or removed from the list dynamically.
- **Efficient Traversal**: Any node can be set as the starting point.
- **Memory management**: Memory can be efficiently managed by reusing nodes that have been deleted from the list. This can help to reduce memory fragmentation and improve overall system performance.
- Can be used in a wide range of applications, from computer graphics to operating system design.
- A node always points to another node, so NULL assignment is not necessary.
- Nodes are traversed quickly from the first to the last.

U ▪
S T

Circular Linked List: Time and Space Complexity

Operation	Best Case Time Complexity	Worst Case Time Complexity	Space Complexity
Accessing a Node	O(1)	O(n)	O(1)
Inserting at Beginning	O(1)	O(1)	O(1)
Inserting at End	O(1)	O(n)	O(1)
Inserting in Middle	O(n/2)	O(n)	O(1)
Deleting a Node	O(n)	O(n)	O(1)
Traversing the List	O(n)	O(n)	O(1)

Case Study: Circular Playlist in Music Players

- Circular playlist is a list of music tracks that is played in a continuous loop, meaning that the last track is connected to the first track to form a circular structure.
- Here circular linked list can be used to represent the circular playlist, where each node in the list represents a music track.
- The "next" pointer of each node points to the next track in the playlist.
- Final node pointing back to the first node to complete the circular structure.
- Provides an efficient and seamless User Experience for music playback, and is a common technique used in many music player applications.

Case Study: Benefits

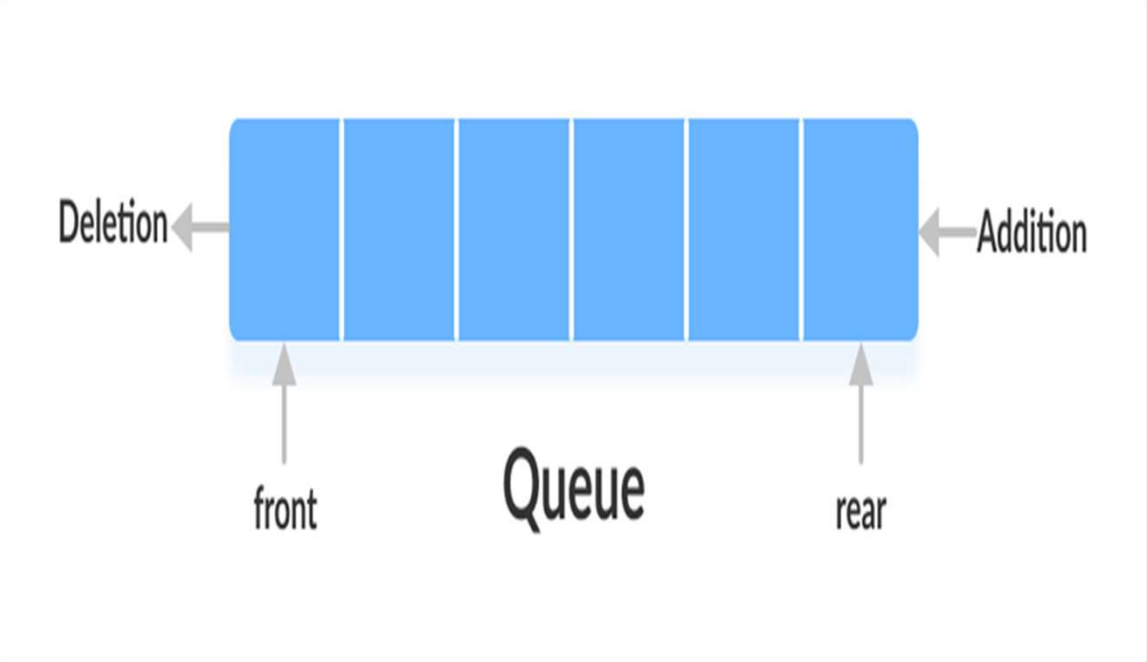
- **Seamless Playback**: As the playlist is circular in nature, the music player can seamlessly transition from the last track to the first track without any interruptions or pauses.
- **Dynamic Playlist Size**: The circular linked list implementation allows for the playlist size to be dynamically adjusted during runtime, which is useful in applications where the playlist needs to be optimized based on the user's preferences.
- **Efficient Navigation**: As the playlist is implemented as a circular linked list, navigating to the next or previous track in the playlist has a constant time complexity, regardless of the position of the track in the playlist.

QUEUE

QUEUE DATA STRUCTURE

- queue is a type of data structure in the Java programming language that stores elements of the same kind
- linear data structure
- The components in a queue are stored in a FIFO (First In, First Out) behavior
- Queues are useful in situations where elements need to be processed in the order they were added, such as handling requests in a web server or processing messages in a messaging system.

- Two ends - Front and Rear



Basic Operations for Queue

- Enqueue() - Insertion of elements to the queue.
- Dequeue() - Removal of elements from the queue.
- Peek() - Acquires the data element available at the front node of the queue without deleting it.
- isFull() - Validates if the queue is full.
- isNull() - Checks if the queue is empty.

Applications of Queue

- **Task Scheduling**: Queues can be used to schedule tasks based on priority or the order in which they were received.
- **Resource Allocation:** Queues can be used to manage and allocate resources, such as printers or CPU processing time.
- **Batch Processing**: Queues can be used to handle batch processing jobs, such as data analysis or image rendering.
- **Message Buffering**: Queues can be used to buffer messages in communication systems, such as message queues in messaging systems or buffers in computer networks.
- **Event Handling**: Queues can be used to handle events in event-driven systems, such as GUI applications or simulation systems.

Advantages of Queue

- large amount of data can be managed efficiently with ease
- Operations such as insertion and deletion can be performed with ease
- useful when a particular service is used by multiple consumers.
- fast in speed for data inter-process communication.
- used in the implementation of other data structures.

Disadvantages of Queue

- The operations such as insertion and deletion of elements from the middle are time consuming.
- Limited Space.
- Searching an element takes O(N) time.
- Maximum size of a queue must be defined
- prior

Implementations of Queue Data Structure

- Queue data structure can be implemented using arrays, linked lists and priority queue.
- Array-based queue: A queue implemented using an array as the underlying data structure. Elements are added to the back of the queue and removed from the front. This implementation is efficient for randomly accessing elements, but less efficient for inserting or removing elements from the front or middle of the queue.

- Linked list-based queue: A queue implemented using a linked list as the underlying data structure. Elements are added to the back of the queue and removed from the front. This implementation is efficient for inserting or removing elements from the front or middle of the queue, but less efficient for randomly accessing elements.
- Priority queue: A queue where each element has a priority assigned to it, and elements are removed from the queue in order of priority. This implementation is useful when elements need to be processed in a specific order, and is efficient for accessing the highest-priority element, but less efficient for accessing elements in arbitrary order.

Advantages of Array-based implementation

- Simple Implementation
- Fast access time

Disadvantages of Array-based implementation

- Fixed size
- Dynamic resizing

Advantages of Linked list-based implementation

- Dynamic Size
- Easy Insertion and Deletion
- Flexibility

Disadvantages of Linked list-based implementation

- Random Access
- Extra Memory
- Traversal
- Fragmentation

Advantages of Priority Queue Based Implementation

Efficient Insertion and Removal

Flexible

Easy to Implement

Provides Efficient Search

Disadvantages of Priority Queue Based Implementation

Limited Functionality

Extra Overhead

Limited Sorting Options

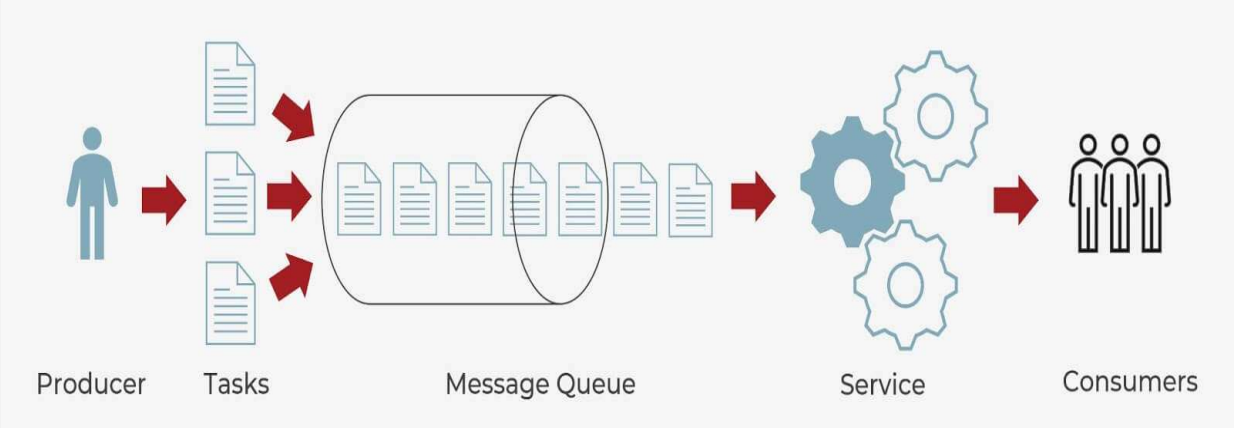
Memory Usage

Time and Space Complexity of Queue Data Structure

Data Structure	Complexity	Time Complexity			Space Complexity
	Operations	enqueue	dequeue	peek	
Queue	Array	O(1)	O(n)	O(1)	O(n)
	Linked list	O(1)	O(1)	O(1)	O(n)
Circular queue		O(1)	O(1)	O(1)	O(n)
Deque		O(1)	O(1)	O(1)	O(n)
Priority queue		O(log n)	O(log n)	O(1)	O(n)

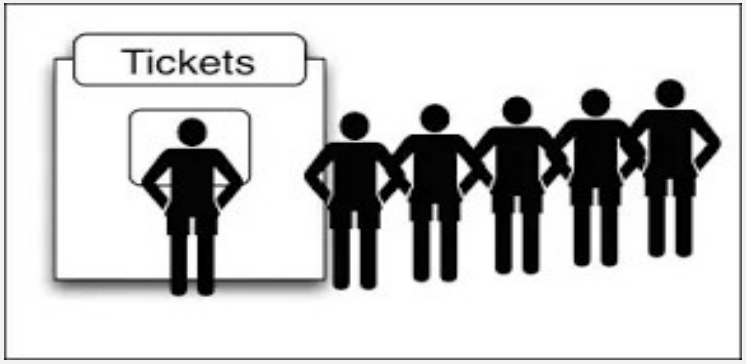
Case study: Messaging Queues

- When it comes to modern cloud architecture, queues are used typically for cross-service communication. Messages are stored on the queue data structure until they are processed by a consumer and deleted, allowing different parts of a system to communicate and process operations in an asynchronous manner.



Case study: Waiting List for a Ride or Service

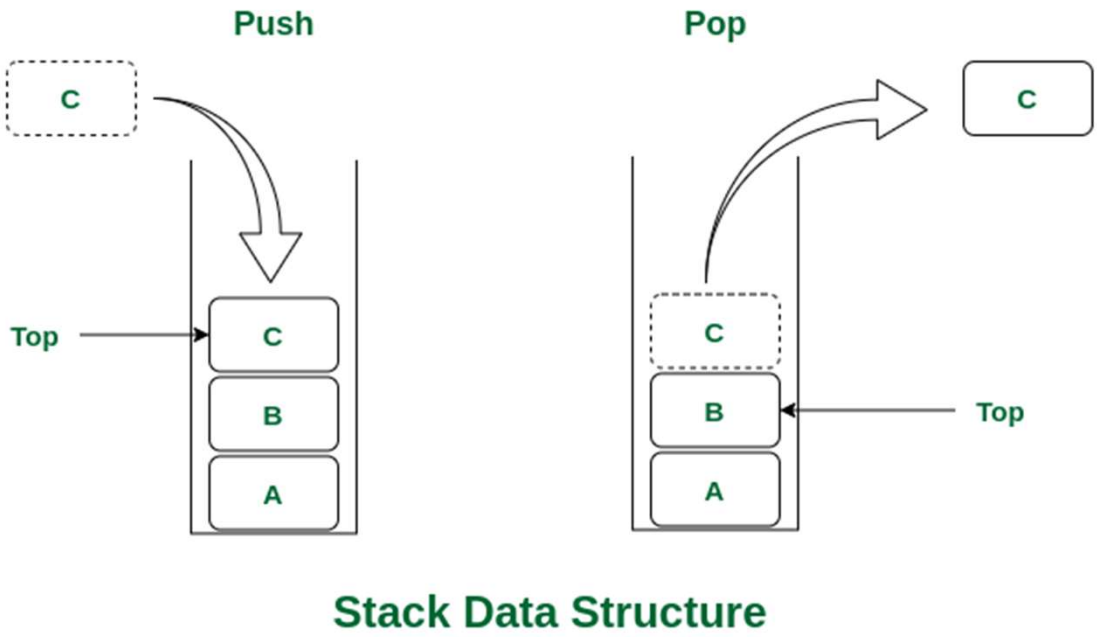
- A waiting list is a common use case for a queue data structure, especially in scenarios where people need to wait their turn for a ride or a service. Queues are commonly used to manage waiting lists for services like amusement park rides or customer support. As customers arrive, they are added to the end of the queue and served in the order they arrived.



Stack Data Structure

- A linear data structure that follows the Last-In-First-Out (LIFO) principle.
- The last element added to the stack will be the first one to be removed.
- To implement the stack, it is required to maintain the pointer to the top of the stack, which is the last element to be inserted because we can access the elements only on the top of the stack.

- Two main operations: push and pop.
- Push adds an element to the top of the stack.
- Pop removes the top element from the stack.



Stack Data Structure

Other Operations on Stack

- Peek: This operation returns the top element from the stack without removing it. This is useful when you want to look at the top element without modifying the stack.
- isEmpty: This operation checks if the stack is empty. It returns true if the stack is empty, and false otherwise.
- Size: This operation returns the number of elements currently in the stack.

Applications of Stack Data Structure

- Implementation of function calls and recursion in programming languages.
- Another application is in the evaluation of arithmetic expressions, where stack is used to store operators and operands and perform the required operations in the correct order.
- Infix to Postfix /Prefix conversion.

Advantages of Stack

- **Easy implementation:** Stack data structure is easy to implement using arrays or linked lists, and its operations are simple to understand and implement.
- **Efficient memory utilization**: Stack uses a contiguous block of memory, making it more efficient in memory utilization as compared to other data structures.
- **Fast access time:** Stack data structure provides fast access time for adding and removing elements as the elements are added and removed from the top of the stack.
- **Helps in function calls:** Stack data structure is used to store function calls and their states, which helps in the efficient implementation of recursive function calls.

- **Supports backtracking:** Stack data structure supports backtracking algorithms, which are used in problem-solving to explore all possible solutions by storing the previous states.
- **Used in Compiler Design:** Stack data structure is used in compiler design for parsing and syntax analysis of programming languages.
- **Enables undo/redo operations**: Stack data structure is used to enable undo and redo operations in various applications like text editors, graphic design tools, and software development environments.

Disadvantages of Stack

- **Limited capacity:** Stack data structure has a limited capacity as it can only hold a fixed number of elements. If the stack becomes full, adding new elements may result in stack overflow, leading to the loss of data.
- **No random access:** Stack data structure does not allow for random access to its elements, and it only allows for adding and removing elements from the top of the stack. To access an element in the middle of the stack, all the elements above it must be removed.
- **Memory management:** Stack data structure uses a contiguous block of memory, which can result in memory fragmentation if elements are added and removed frequently.

- **Not suitable for certain applications:** Stack data structure is not suitable for applications that require accessing elements in the middle of the stack, like searching or sorting algorithms.
- **Stack overflow and underflow**: Stack data structure can result in stack overflow if too many elements are pushed onto the stack, and it can result in stack underflow if too many elements are popped from the stack.
- **Recursive function calls limitations:** While stack data structure supports recursive function calls, too many recursive function calls can lead to stack overflow, resulting in the termination of the program.

Implementations of Stack Data Structure

- Stack data structure can be implemented using arrays and linked lists. Array implementation is simpler and faster, but has a fixed size and may cause overflow or underflow errors.
- Linked list implementation is more flexible and dynamic, but requires extra memory allocation for each element and may have slower access times.

Advantages of array implementation:

- Easy to implement.
- Memory is saved as pointers are not involved.

Disadvantages of array implementation:

- It is not dynamic i.e., it doesn’t grow and shrink depending on needs at runtime.
- The total size of the stack must be defined beforehand.

Advantages of Linked List implementation:

- The linked list implementation of a stack can grow and shrink according to the needs at runtime.
- It is used in many virtual machines like JVM.

Disadvantages of Linked List implementation:

- Requires extra memory due to the involvement of pointers.
- Random accessing is not possible in stack.

Time and Space Complexity of Stack Data Structure

- The time complexity of push, pop, peek, and isEmpty operations on stack data structure is O(1)

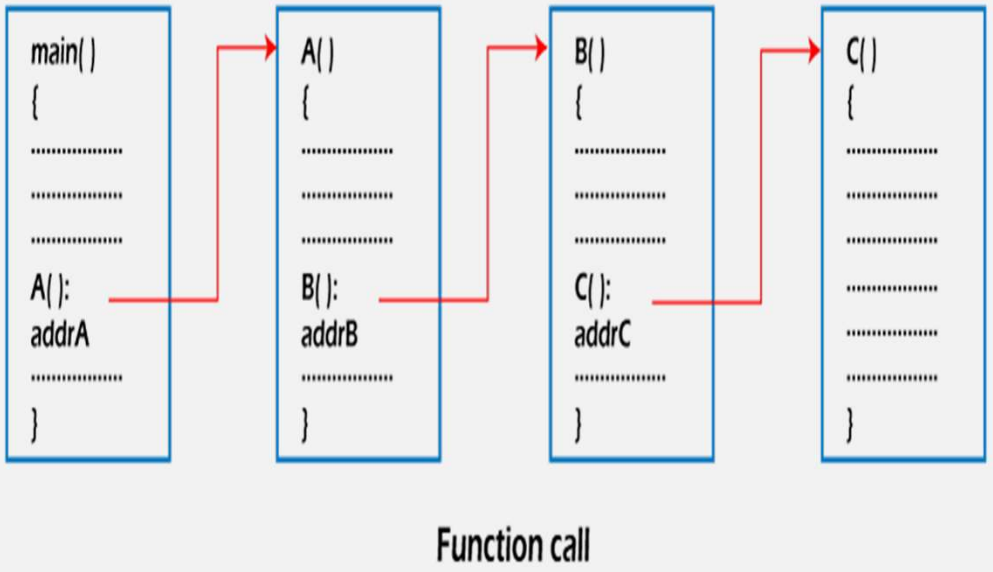
- The space complexity of stack data structure is also O(n)

Operations	Complexity
push()	O(1)
pop()	O(1)
isEmpty()	O(1)
size()	O(1)

Case Study: Function Calls

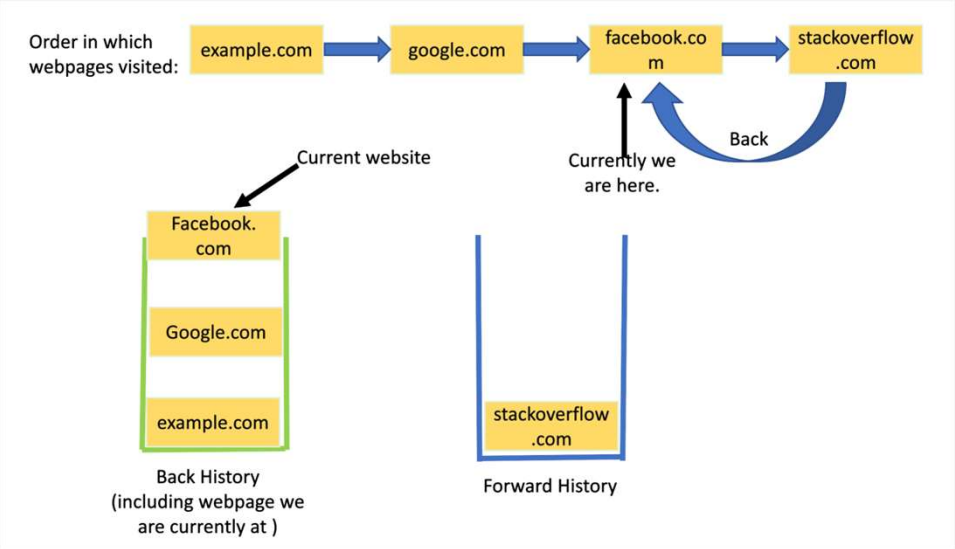
- Another example of using a stack data structure is in programming languages. When a function is called, its parameters and local variables are pushed onto the call stack. When the function returns, these values are popped off the stack.

- This allows for nested function calls and ensures that each function has its own set of parameters and local variables. The stack data structure also ensures that functions are executed in the correct order.



Case Study: Browser History

- One example of using a stack data structure is in web browsers. When you visit a website, the URL is added to the top of the browser history stack. If you click the back button, the most recent URL is popped off the stack and displayed.
- This allows users to easily navigate through their browsing history and go back to previously visited websites. The stack data structure ensures that the most recently visited website is always at the top of the stack.



TREE DATA STRUCTURE

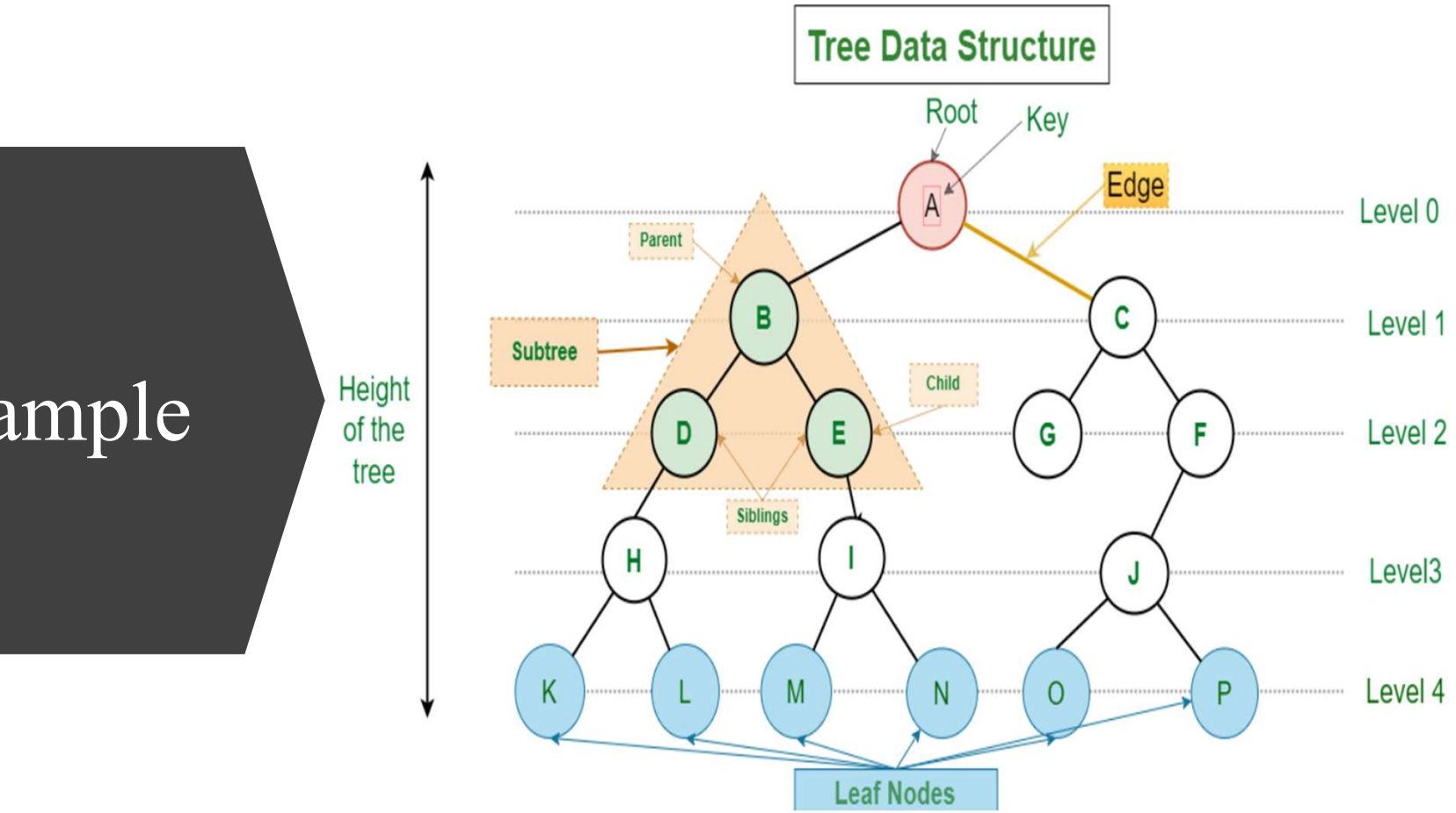
What is Tree?

- A tree data structure is a hierarchical data structure that consists of nodes connected by edges. It is a type of graph that does not contain any cycles (loops), meaning that there is only one path from any node to any other node.

- Top of the tree is the root node, which has zero or more child nodes. Each child node can have its own children, forming sub-trees. Nodes with no children are called leaf nodes.

- In a tree data structure, each node can have any number of child nodes, but each child node can only have one parent. This one-to-many relationship between nodes makes trees a useful way to represent hierarchical relationships.

Example



Why Tree Data Structure?

- One reason to use trees might be because you want to store information that naturally forms a hierarchy.
- Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).
- Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).
- Like Linked Lists and unlike Arrays, Trees don't have an upper limit on the number of nodes as nodes are linked using pointers.

Basic Operation Of Tree

- **Create** – create a tree in data structure.
- **Insert** − Inserts data in a tree.
- **Search** − Searches specific data in a tree to check it is present or not.
- **Preorder Traversal** – perform Traveling a tree in a pre-order manner data structure .
- **In order Traversal** – perform Traveling a tree in an in-order manner.
- **Post order Traversal** –perform Traveling a tree in a post-order manner.

Type of Tree

- Binary Tree
- Binary Search Tree
- AVL Tree
- B Tree
- B+ Tree

Application of Tree

- **Artificial intelligence**: Decision trees are used in machine learning algorithms for decision making and classification tasks.
- **Network routing**: Trees are used to model the hierarchy of a network routing system, such as the internet.
- **Game theory**: Game trees are used to represent the possible outcomes of a game and help players make strategic decisions.
- **Image processing**: Trees are used in image compression algorithms, such as wavelet compression, to efficiently represent the image data.
- **Finance**: Trees are used in financial modeling to represent the possible outcomes of investment decisions and help with risk analysis.

Binary tree

A binary tree is a tree data structure composed of nodes, each of which has at most, two children, referred to as left and right nodes. The tree starts off with a single node known as the root.

Each node in the tree contains the following:

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

Properties of Binary Tree

- At each level of i, the maximum number of nodes is 2^i.
- The height of the tree is defined as the longest path from the root node to the leaf node. The tree which is shown above has a height equal to 3. Therefore, the maximum number of nodes at height 3 is equal to (1+2+4+8) = 15. In general, the maximum number of nodes possible at height h is $(2^0 + 2^1 + 2^2 + \ldots 2^h) = 2^{h+1} - 1$.
- The minimum number of nodes possible at height h is equal to **h+1**.
- If the number of nodes is minimum, then the height of the tree would be maximum. Conversely, if the number of nodes is maximum, then the height of the tree would be minimum.

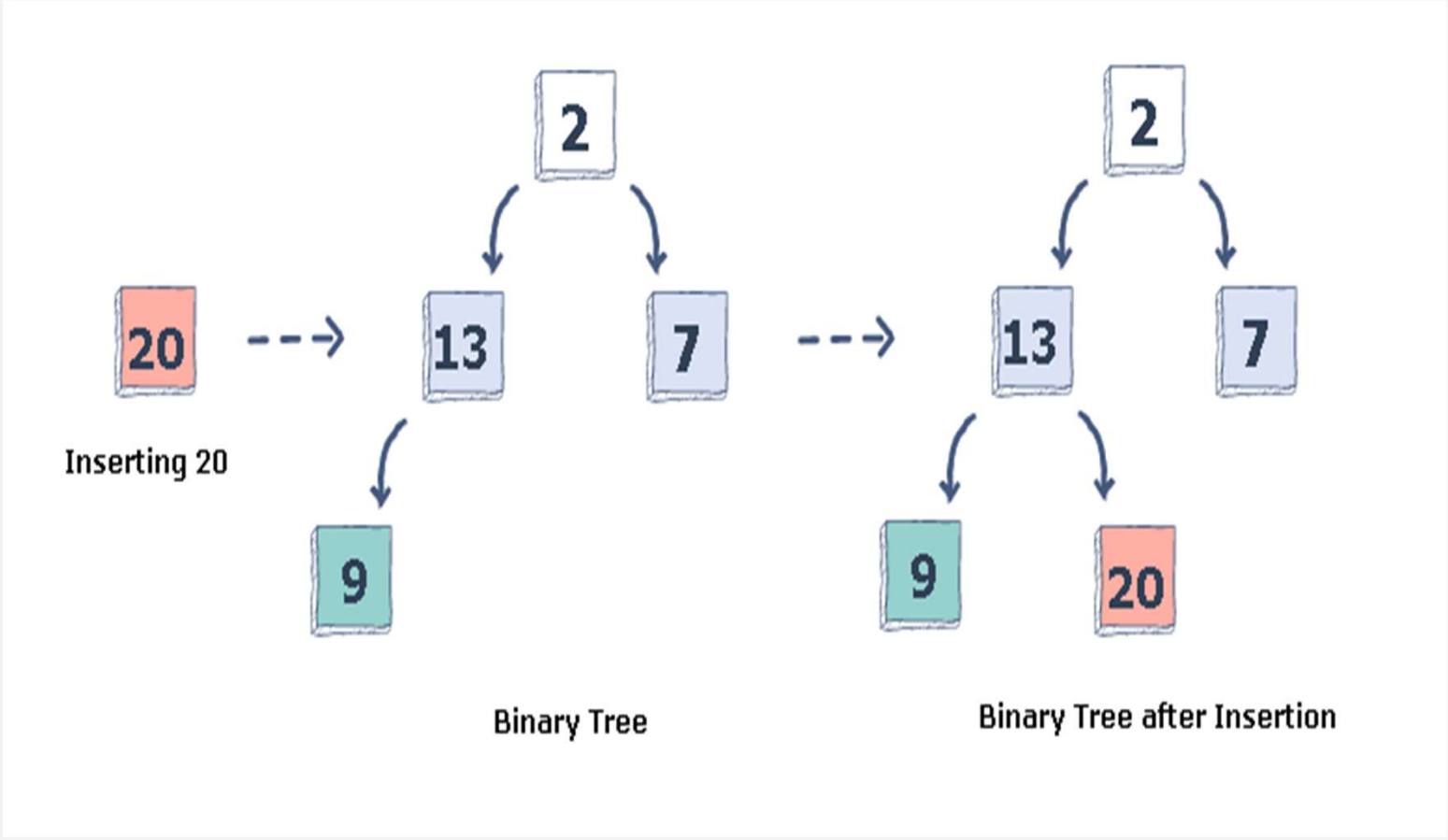
Common operations

Following is a list of common operations that can be performed on a binary tree:

1. Insertion

Elements may be inserted into a binary tree in any order. The very first insertion operation creates the root node. Each insertion that follows iteratively searches for an empty location at each level of the tree.

Upon finding an empty left *or* right child, the new element is inserted. By convention, the insertion always begins from the *left* child node.


2
20
13
7
Inserting 20
9
Binary Tree
2
13
7
9
20
Binary Tree after Insertion

Insertion algorithm

1.Start at the root of the binary tree.

2.If the root is empty, create a new node with the key value of the new node and make it the root of the tree.

3.If the root is not empty, compare the key value of the new node to the key value of the current node.

4.If the key value of the new node is less than the key value of the current node, go to the left subtree of the current node.

5.If the key value of the new node is greater than the key value of the current node, go to the right subtree of the current node.

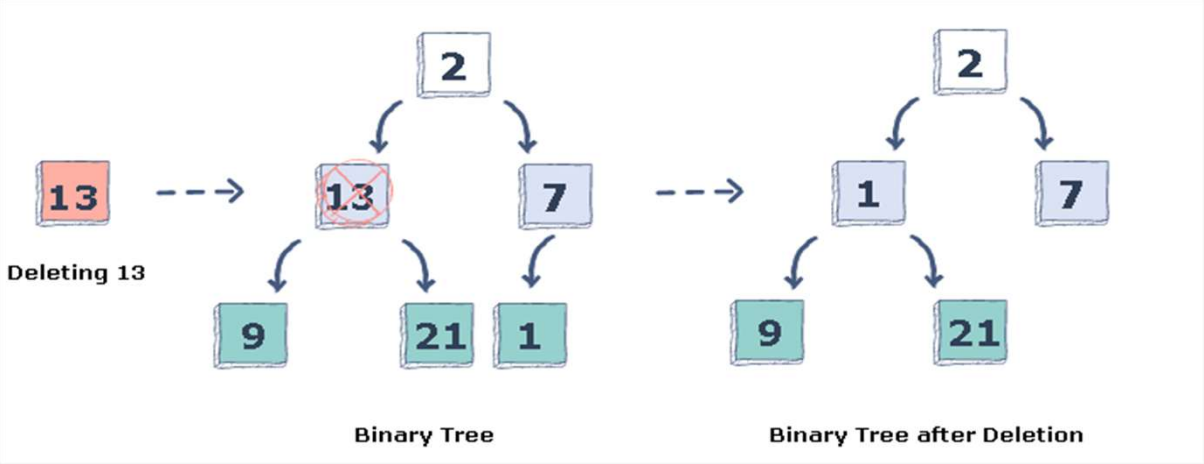
6.Repeat steps 3-5 until an empty position is found.

7.Create a new node with the key value of the new node at the empty position.

2. Deletion

An element may also be removed from the binary tree. Since there is no particular order among the elements, upon deletion of a particular node, it is replaced with the right-most element.

Let's look at an example to get a better idea of how the deletion process works.



Deletion Algorithm

1 Start at the root node.

2 Traverse the tree to find the node to be deleted.

- If the node is not found, exit the function.

3 If the node has no children, simply delete it.

4 If the node has one child, replace the node with its child.

5 If the node has two children, find the node with the smallest value in the right subtree (the successor).

- Replace the node to be deleted with the successor.
- Delete the successor (which will have at most one child).

3. Tree traversal

Another frequently used tree operation is traversal.

Tree traversal is the process of visiting each node present in a tree.

There are three methods of tree traversal:

In-order traversal

Post-order traversal

Pre-order traversal

Types of Binary Tree

- Full/ proper/ strict Binary tree
- Complete Binary tree
- Perfect Binary tree
- Degenerate Binary tree
- Balanced Binary tree

Full/ proper/ strict Binary tree

- The full binary tree is also known as a strict binary tree. The tree can only be considered as the full binary tree if each node must contain either 0 or 2 children. The full binary tree can also be defined as the tree in which each node must contain 2 children except the leaf nodes

Properties of Full Binary Tree

- The number of leaf nodes is equal to the number of internal nodes plus 1. In the above example, the number of internal nodes is 5; therefore, the number of leaf nodes is equal to 6.
- The maximum number of nodes is the same as the number of nodes in the binary tree, i.e., 2^{h+1} -1.
- The minimum number of nodes in the full binary tree is 2*h-1.
- The minimum height of the full binary tree is **$\log_2(n+1) - 1$.**

Complete Binary Tree

The complete binary tree is a tree in which all the nodes are completely filled except the last level. In the last level, all the nodes must be as left as possible. In a complete binary tree, the nodes should be added from the left.

Properties of Complete Binary Tree

The maximum number of nodes in complete binary tree is $2^{h+1} - 1$.

The minimum number of nodes in complete binary tree is 2^{h}.

The minimum height of a complete binary tree is **$\log_2(n+1) - 1$.**

Perfect Binary Tree

A tree is a perfect binary tree if all the internal nodes have 2 children, and all the leaf nodes are at the same level.

Degenerate Binary Tree

The degenerate binary tree is a tree in which all the internal nodes have only one children.

Balanced Binary Tree

The balanced binary tree is a tree in which both the left and right trees differ by atmost 1. For example, ***AVL*** and ***Red-Black trees*** are balanced binary tree.

Binary Search Tree

- A binary search tree follows some order to arrange the elements. In a Binary search tree, the value of left node must be smaller than the parent node, and the value of right node must be greater than the parent node. This rule is applied recursively to the left and right subtrees of the root.
- Let's understand the concept of Binary search tree with an example.
- In the above figure, we can observe that the root node is 40, and all the nodes of the left subtree are smaller than the root node, and all the nodes of the right subtree are greater than the root node.
- Similarly, we can see the left child of root node is greater than its left child and smaller than its right child. So, it also satisfies the property of binary search tree. Therefore, we can say that the tree in the above image is a binary search tree.

Suppose if we change the value of node 35 to 55 in the above tree, check whether the tree will be binary search tree or not.

In the above tree, the value of root node is 40, which is greater than its left child 30 but smaller than right child of 30, i.e., 55. So, the above tree does not satisfy the property of Binary search tree. Therefore, the above tree is not a binary search tree.

Advantages of Binary search tree

Searching an element in the Binary search tree is easy as we always have a hint that which subtree has the desired element.

As compared to array and linked lists, insertion and deletion operations are faster in BST.

Algorithm to search

- Search (root, item)
- Step 1 - if (item = root → data) or (root = NULL)
- return root
- else if (item **< root** → data)
- return Search(root → left, item)
- else
- return Search(root → right, item)
- END if
- Step 2 - END

Insertion in Binary Search tree

A new key in BST is always inserted at the leaf. To insert an element in BST, we have to start searching from the root node; if the node to be inserted is less than the root node, then search for an empty location in the left subtree. Else, search for the empty location in the right subtree and insert the data. Insert in BST is similar to searching, as we always have to maintain the rule that the left subtree is smaller than the root, and right subtree is larger than the root.

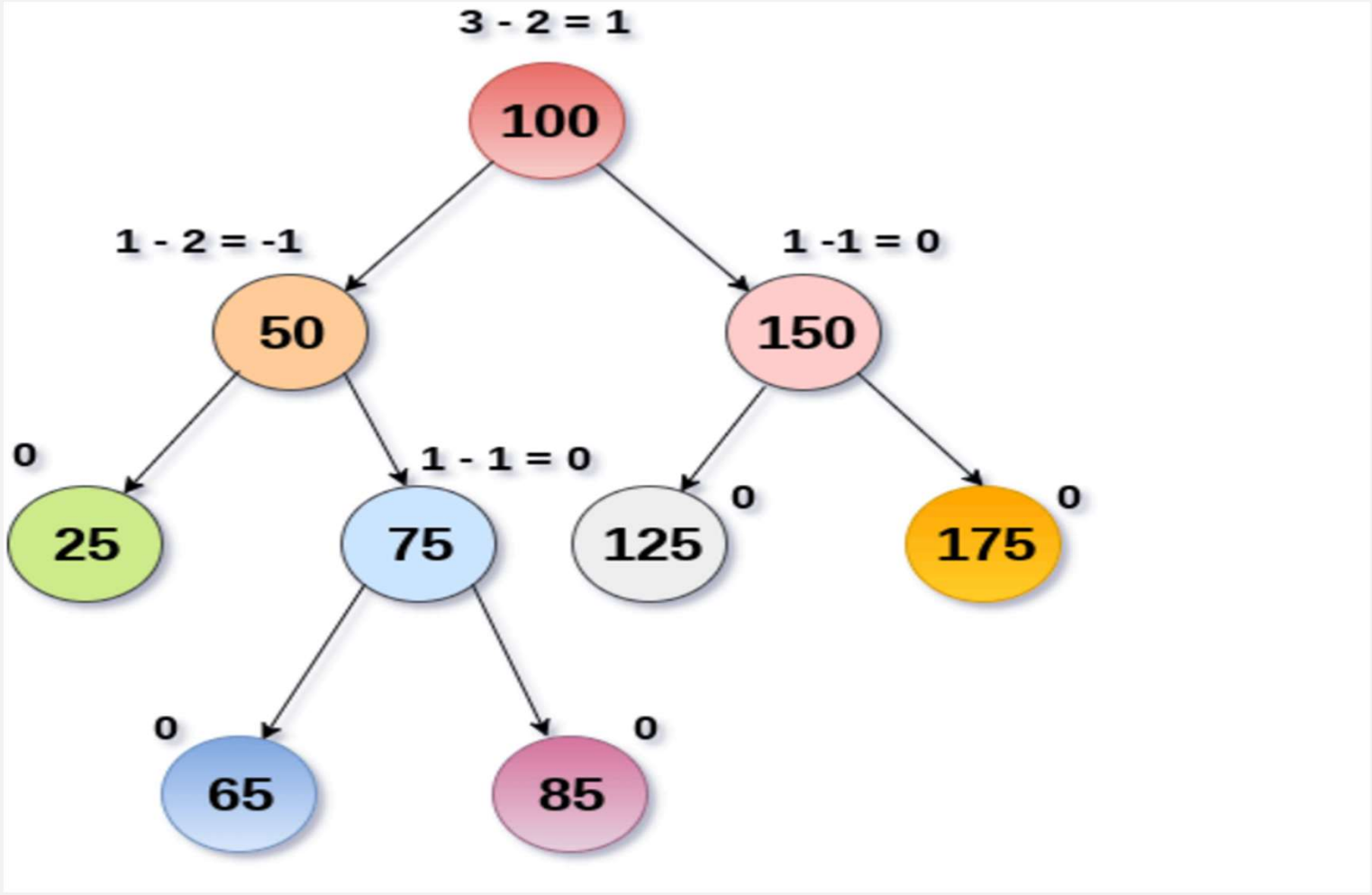
AVL Tree

- AVL Tree is invented by GM Adelson- Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors.
- AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree.
- Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Why AVL Tree?

- AVL tree controls the height of the binary search tree by not letting it to be skewed. The time taken for all operations in a binary search tree of height h is **O(h)**. However, it can be extended to **O(n)** if the BST becomes skewed (worst case). By limiting this height to log n, AVL tree imposes an upper bound on each operation to be **O(log n)** where n is the number of nodes.
- Balance Factor (k) = height (left(k)) - height (right(k))

  If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree .If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height .If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

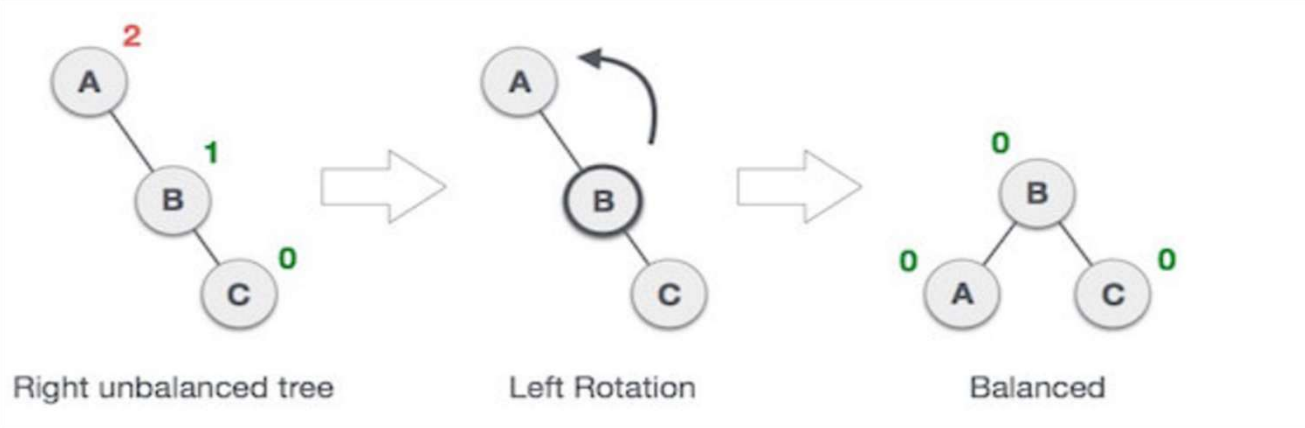

3 - 2 = 1
100
1 - 2 = -1
1 -1 = 0
50
150
0
1 - 1 = 0
0
0
25
75
125
175
0
0
65
85

AVL Rotations

- We perform rotation in AVL tree only in case if Balance Factor is other than **-1, 0, and 1**. There are basically four types of rotations which are as follows:
- L L rotation: Inserted node is in the left subtree of left subtree of A
- R R rotation : Inserted node is in the right subtree of right subtree of A
- L R rotation : Inserted node is in the right subtree of left subtree of A
- R L rotation : Inserted node is in the left subtree of right subtree of A
- The first two rotations LL and RR are single rotations and the next two rotations LR and RL are double rotations. For a tree to be unbalanced, minimum height must be at least 2, Let us understand each rotation

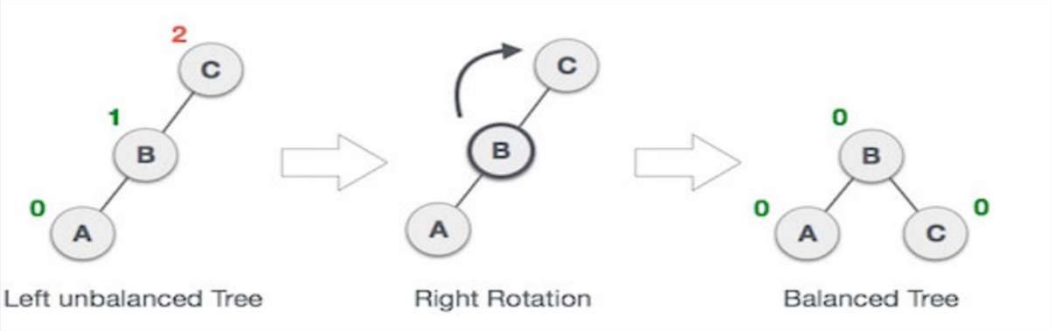
RR Rotation

- When BST becomes unbalanced, due to a node is inserted into the right subtree of the right subtree of A, then we perform RR rotation, RR rotation is an anticlockwise rotation, which is applied on the edge below a node having balance factor –2.



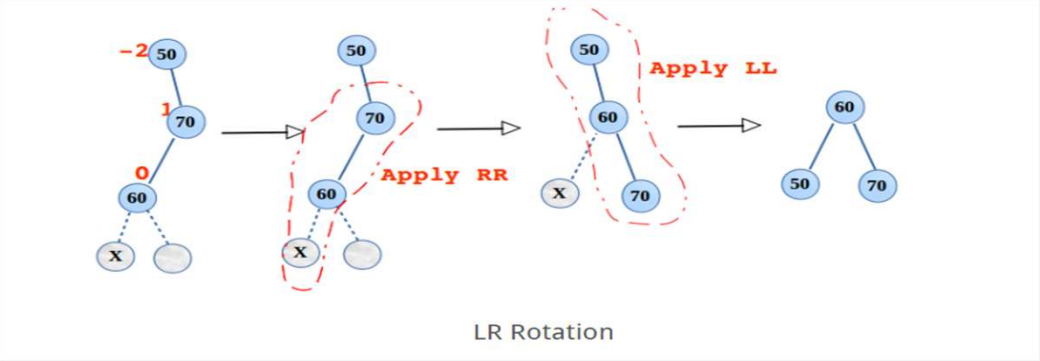
LL Rotation

- When BST becomes unbalanced, due to a node is inserted into the left subtree of the left subtree of C, then we perform LL rotation, LL rotation is clockwise rotation, which is applied on the edge below a node having balance factor 2.



LR Rotation

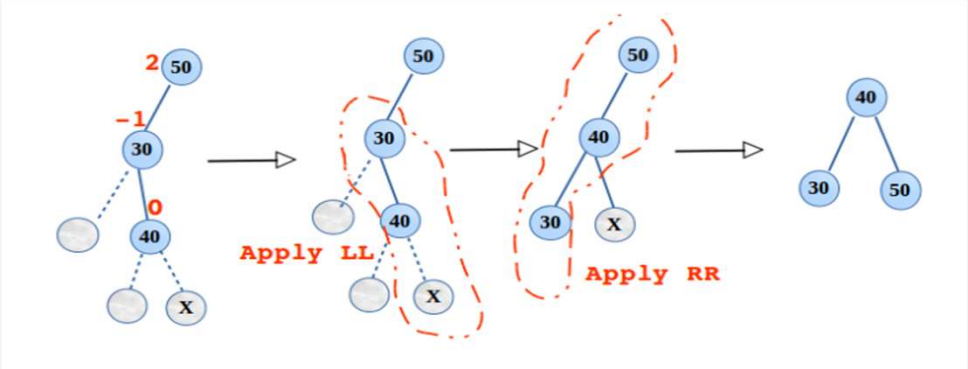
- Double rotations are bit tougher than single rotation which has already explained above. LR rotation = RR rotation + LL rotation, i.e., first RR rotation is performed on subtree and then LL rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.



LR Rotation

RL Rotation

RL ROTATION= LL rotation + RR rotation, i.e., first LL rotation is performed on subtree and then RR rotation is performed on full tree, by full tree we mean the first node from the path of inserted node whose balance factor is other than -1, 0, or 1.



Applications of AVL TREE

- Databases: As mentioned earlier, AVL trees are often used to index databases to support fast searching, insertion, and deletion operations.
- Computer graphics: AVL trees can be used in computer graphics applications, such as image rendering, to quickly find the closest points or to search for objects that intersect with a particular region.
- Network routers: Network routers often use AVL trees to store routing tables for efficient packet routing in computer networks.
- Compiler design: AVL trees can be used in compiler design to optimize the organization and storage of symbol tables, which contain information about the variables and functions used in a program.

Advantages

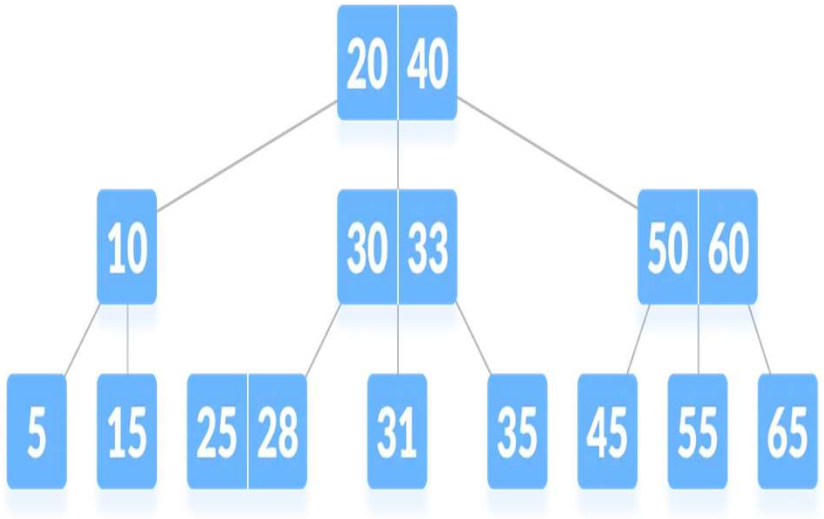
- **AVL trees can self-balance:** One of the primary concerns of computer science professionals is ensuring that their trees are balanced, and AVL trees have a higher likelihood of being balanced. An unbalanced tree means operations will take longer to complete, resulting in time-consuming lookup applications.
- It is not skewed in any way.
- AVL tree also have the Balancing capabilities with a different type of rotation
- Better searching time complexity than other trees, such as the binary Tree.

Disadvantages

- AVL trees are difficult to implement.
- In addition, AVL trees have high constant factors for some operations. For example, restructuring is an expensive operation, and an AVL tree may have to re-balance itself log 2 *n* in the worst case during a removal of a node.
- Most STL implementations of the ordered associative containers (sets, multisets, maps and multimaps) use red-black trees instead of AVL trees. Unlike AVL trees, red-black trees require only one restructuring for a removal or an insertion.

B Tree

- B-tree is a special type of self-balancing search tree in which each node can contain more than one key and can have more than two children.
- It is a generalized form of the binary search tree.
- It is also known as a height-balanced m-way tree.



PROPERTIES OF B-TREE

- For each node x, the keys are stored in increasing order.
- In each node, there is a boolean value x.leaf which is true if x is a leaf.
- If n is the order of the tree, each internal node can contain at most n - 1 keys along with a pointer to each child.

PROPERTIES OF B-TREE

- Each node except root can have at most n children and at least n/2 children.
- All leaves have the same depth (i.e. height-h of the tree).
- The root has at least 2 children and contains a minimum of 1 key.
- If $n \geq 1$, then for any n-key B-tree of height h and minimum degree $t \geq 2$, $h \geq \log_t (n+1)/2$.

ADVANTAGES OF B-TREE

- B-Trees have a guaranteed time complexity of O(log n) for basic operations like insertion, deletion, and searching, which makes them suitable for large data sets and real-time applications.
- B-Trees are self-balancing.
- High-concurrency and high-throughput.
- Efficient storage utilization.

OPERATIONS ON B-TREE

- ***Searching Algorithm***

```
BtreeSearch(x, k)
 i = 1
while i ≤ n[x] and k ≥ key i[x]
   do i = i + 1
 if i  n[x] and k = keyi[x]
    then return (x, i)
 if leaf [x]
    then return NIL
 else
   return BtreeSearch(ci[x], k)
```

- ***Insertion Algorithm***

```
function insert(B-tree T, element x):
    if T is empty:
        create new leaf node L
        insert x into L in sorted order
        make L the root of T
    else:
        let N be the node pointed to by the root of T
        if N is not a leaf node:
            let C be the appropriate child node of N to follow
            insert(T, x) into C
```

```
if C was split:
        promote the middle element of C to N
        if N is full:
            recursively split and promote as necessary
  else:  // N is a leaf node
      if N has room for x:
          insert x into N in sorted order
      else:
          split N into two new leaf nodes L and R
          insert x into the appropriate node (L or R)
          promote the middle element of L to N
          if N is full:
              recursively split and promote as necessary
```

- ***Deletion Algorithm***

```
function delete(B-tree T, element x):
    let N be the node containing x (or the leaf node where x should be)
    if N is a leaf node:
        if N has more than the minimum number of keys:
            remove x from N
        else:
            let P be the parent node of N
            let S be the sibling of N (to the left or right)
            if S has more than the minimum number of keys:
                move an element from P to N
```

```
            move an element from S to P
        else:
            merge N, P, and S into a single node N'
            delete the element from N'
            delete(P, element that was moved to P) recursively
            if T has become empty:
                set T to be the only child of the original root
else:  // N is an internal node
    let C be the appropriate child node of N to follow
    if C has the minimum number of keys:
        let L and R be the left and right siblings of C, respectively
```

```
if L has more than the minimum number of keys:
        move an element from L to C
        move the largest element from C to N
    else if R has more than the minimum number of keys:
        move an element from R to C
        move the smallest element from C to N
    else:
        merge C, N, and one of its siblings into a single node C'
        delete(N, element that was moved to N) recursively from C'
        if T has become empty:
            set T to be the only child of the original root
delete(C, x) recursively
```

DISADVANTAGES OF B-TREE

01

B-Trees are based on disk-based data structures and can have a high disk usage.

02

Not the best for all cases.

03

Slow in comparison to other data structures.

APPLICATION OF B-TREE

- It is used in large databases to access data stored on the disk
- Searching for data in a data set can be achieved in significantly less time using the B-Tree
- With the indexing feature, multilevel indexing can be achieved.
- Most of the servers also use the B-tree approach.
- B-Trees are used in CAD systems to organize and search geometric data.
- B-Trees are also used in other areas such as natural language processing, computer networks, and cryptography.

B+ TREE

- B + tree is a variation of B-tree data structure. In a B + tree, data pointers are stored only at the leaf nodes of the tree. In a B+ tree structure of a leaf node differs from the structure of internal nodes.
- The leaf nodes have an entry for every value of the search field, along with a data pointer to the record (or to the block that contains this record).
- The leaf nodes of the B+ tree are linked together to provide ordered access on the search field to the records.

ADVANTAGES OF B+TREE

- A B+ tree with ‘l’ levels can store more entries in its internal nodes compared to a B-tree having the same ‘l’ levels. This accentuates the significant improvement made to the search time for any given key. Having lesser levels and the presence of P_{next} pointers imply that the B+ trees is very quick and efficient in accessing records from disks.
- Data stored in a B+ tree can be accessed both sequentially and directly.
- It takes an equal number of disk accesses to fetch records.
- B+ trees have redundant search keys, and storing search keys repeatedly is not possible.

DISADVANTAGE OF B+TREE

- The major drawback of B-tree is the difficulty of traversing the keys sequentially. The B+ tree retains the rapid random access property of the B-tree while also allowing rapid sequential access.

APPLICATIONS OF B+TREE

- Multilevel Indexing
- Faster operations on the tree (insertion, deletion, search)
- Database indexing

Data Structure-Graph

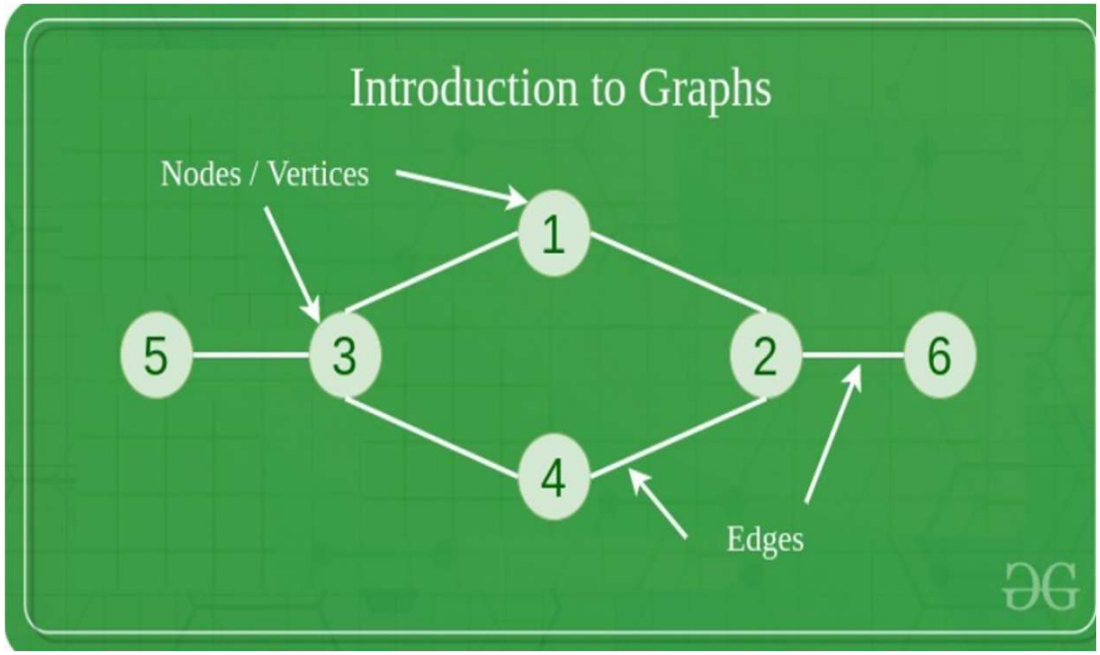
Graph- Introduction

- Non-linear data structure consisting of vertices and edges.
- Consists of a set of nodes and a set of edges that connect pairs of nodes.
- Represent the relationships or connections between these entities.
- Used to model a wide range of real-world scenarios, including social networks, transportation networks etc.

Components of a Graph

- **Vertices:** Vertices are the fundamental units of the graph. Sometimes, vertices are also known as vertex or nodes. Every node/vertex can be labeled or unlabelled.

- **Edges:** Edges are used to connect two nodes of the graph. It can be ordered pair of nodes in a directed graph.Sometimes, edges are also known as arcs. Every edge can be labelled/unlabelled.



Graph – Significance

- Representation of Relationships
- Problem solving
- Data Analysis
- Modeling Complex Systems
- Visualization
- Communication and Collaboration

Graph–Types

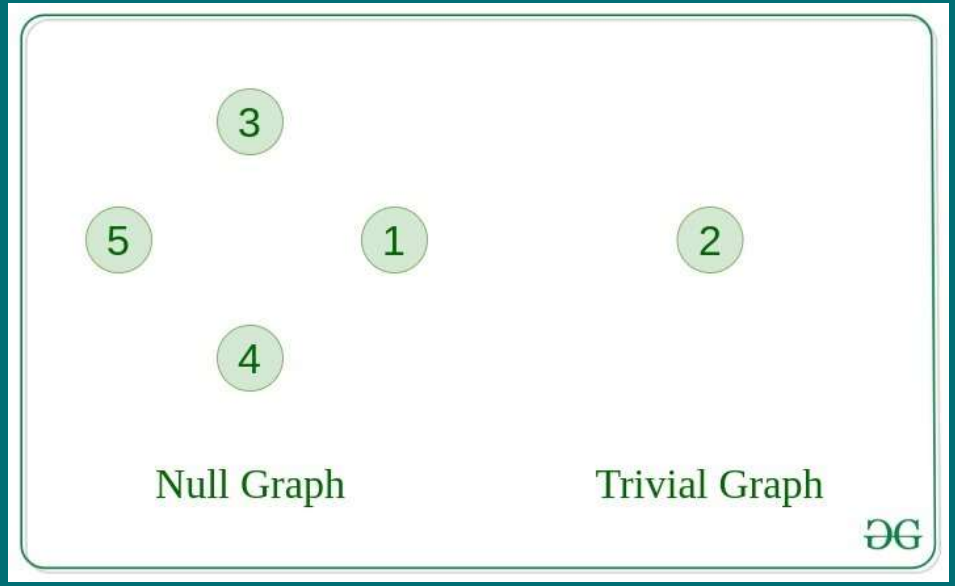
Main types of graphs are:

1. Null Graph

A graph is known as a null graph if there are no edges in the graph.

2. Trivial Graph

Graph having only a single vertex, it is also the smallest graph possible



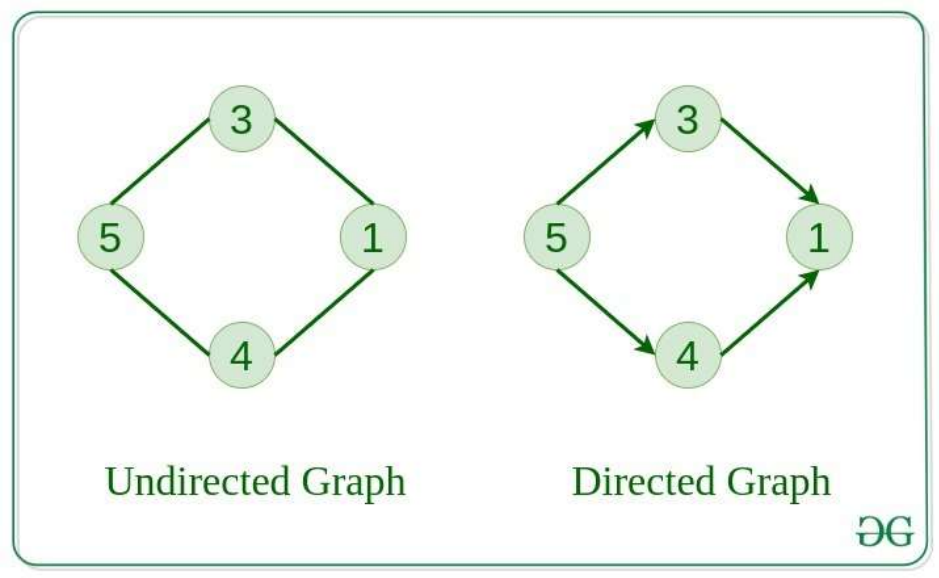
Graph–Types(Cont)

3. Undirected Graph

A graph in which edges do not have any direction. That is the nodes are unordered pairs in the definition of every edge.

4. Directed Graph

A graph in which edge has direction. That is the nodes are ordered pairs in the definition of every edge.



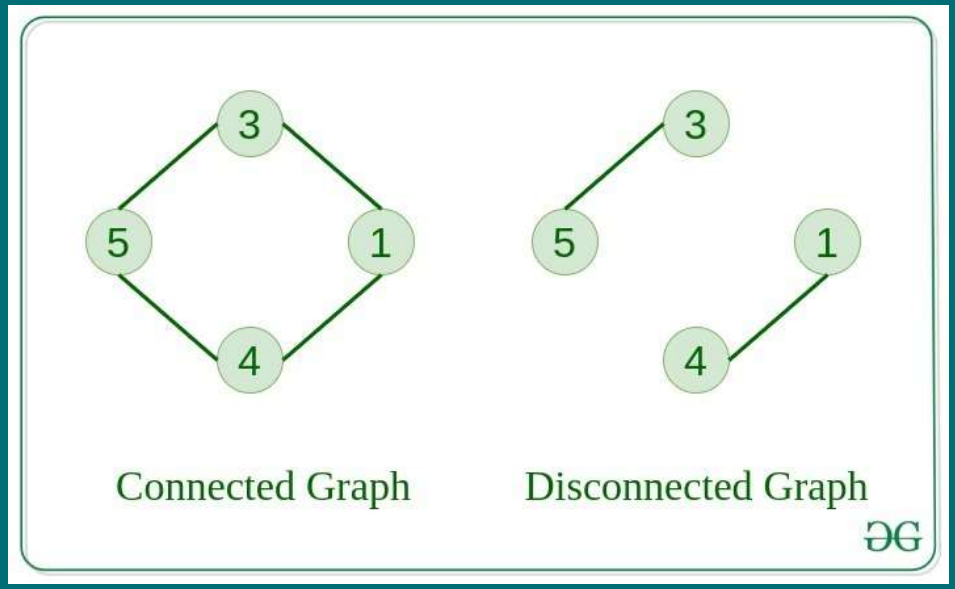
Graph–Types(Cont)

5. Connected Graph

The graph in which from one node we can visit any other node in the graph is known as a connected graph.

6. Disconnected Graph

The graph in which at least one node is not reachable from a node is known as a disconnected graph.



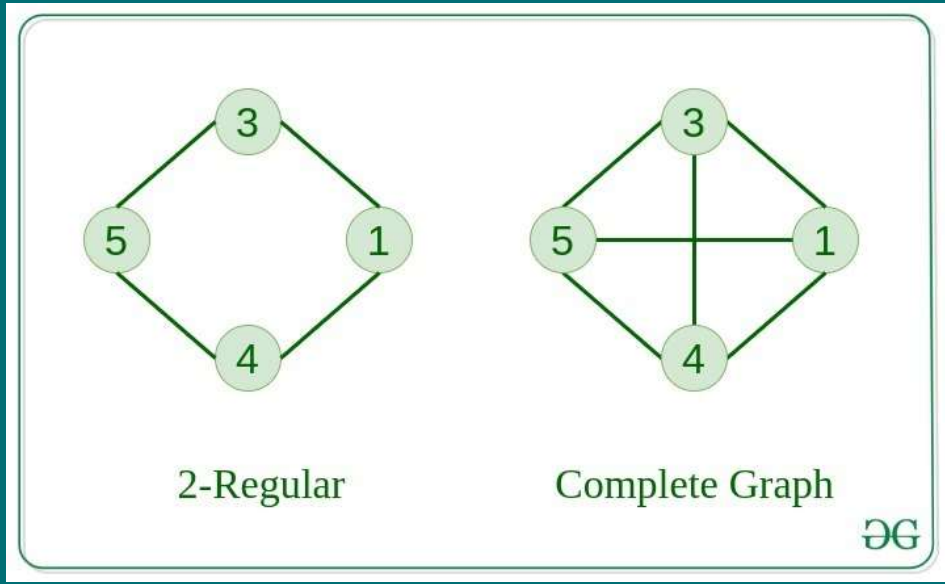
Graph–Types(Cont)

7. Regular Graph

The graph in which the degree of every vertex is equal to K is called K regular graph.

8. Complete Graph

The graph in which from each node there is an edge to each other node.



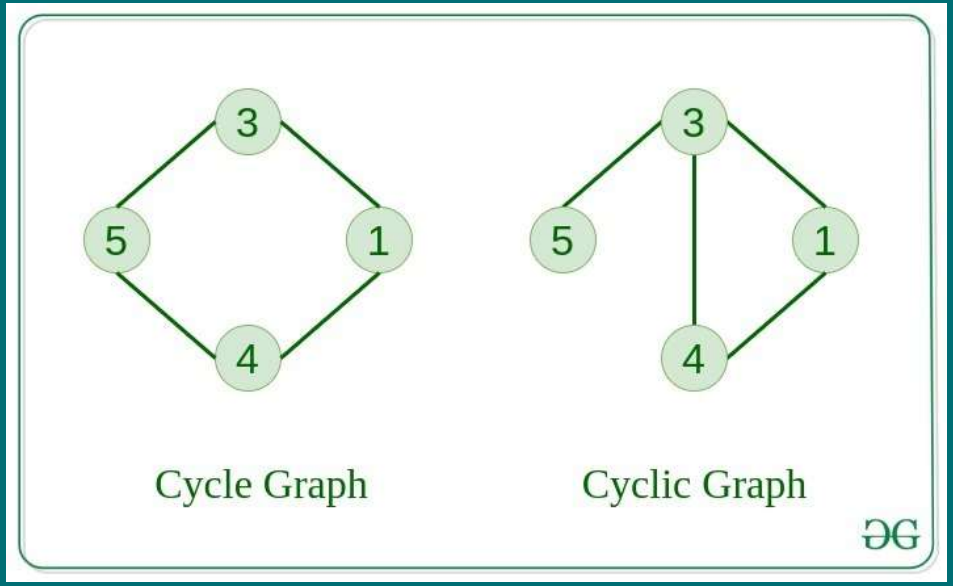
Graph–Types(Cont)

9. Cycle Graph

The graph in which the graph is a cycle in itself, the degree of each vertex is 2.

10. Cyclic Graph

A graph containing at least one cycle is known as a Cyclic graph.



U ▪
S T

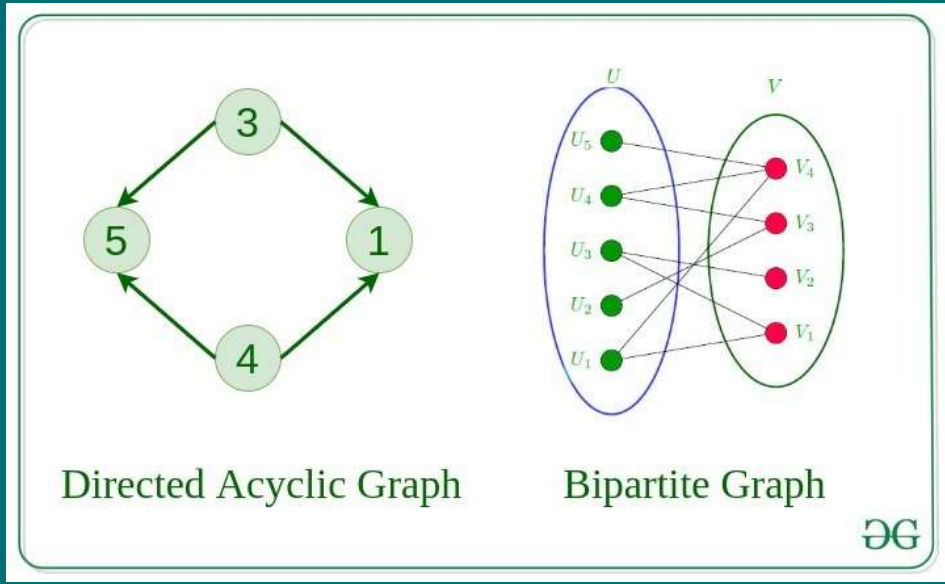
Graph–Types(Cont)

11. Directed Acyclic Graph

A Directed Graph that does not contain any cycle.

12. Bipartite Graph

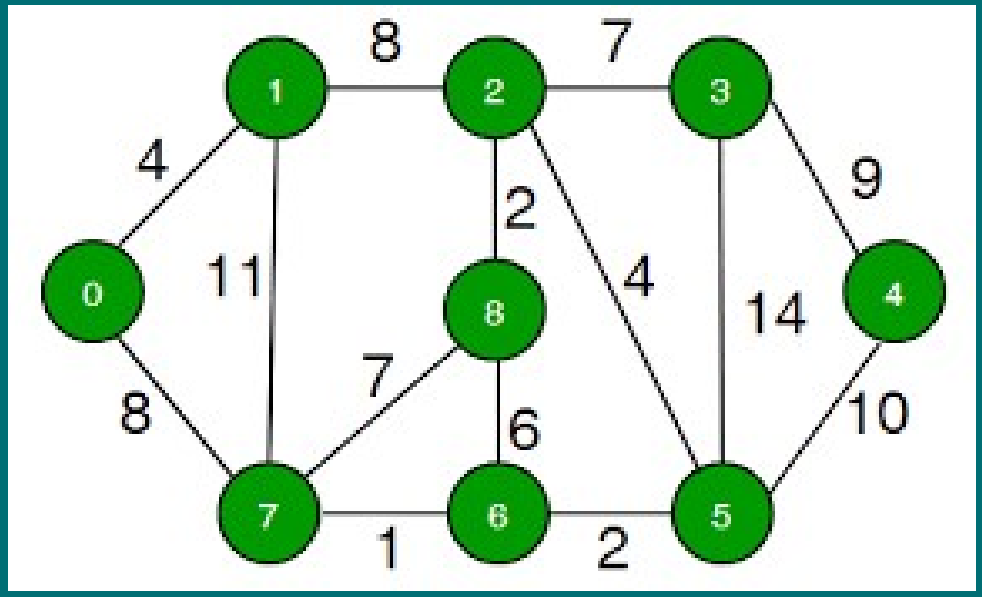
A graph in which vertex can be divided into two sets such that vertex in each set does not contain any edge between them.



Graph–Types(Cont)

13. Weighted Graph

- A graph in which the edges are already specified with suitable weight is known as a weighted graph.
- Weighted graphs can be further classified as directed weighted graphs and undirected weighted graphs.

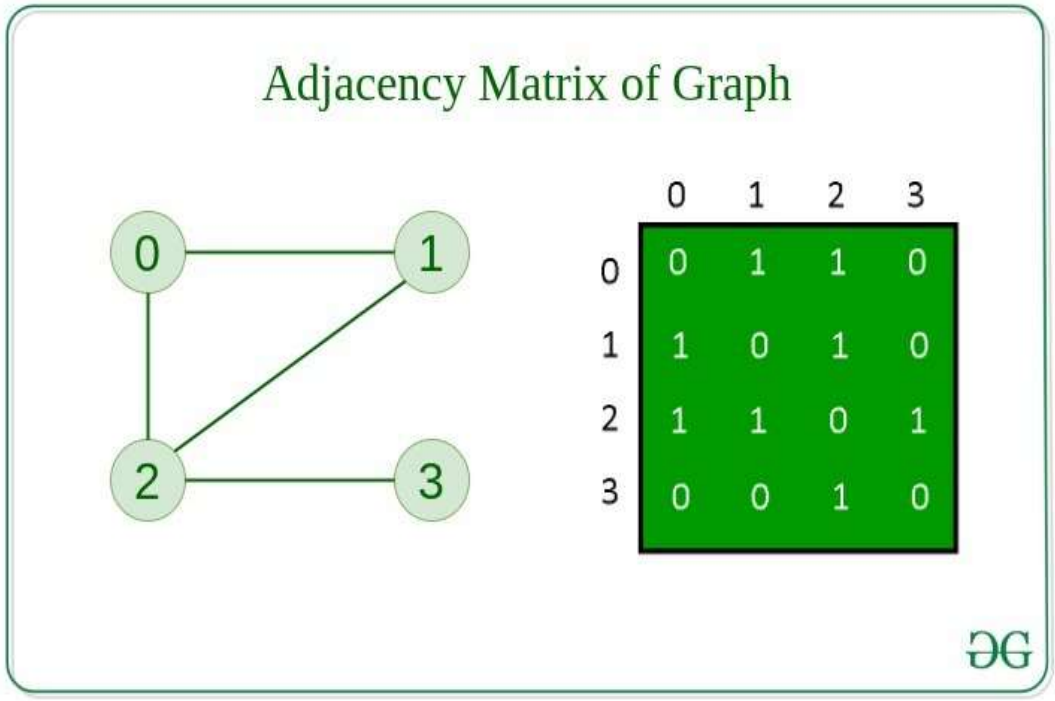


U · S T

Graphs-Representation

1. **Adjacency Matrix**

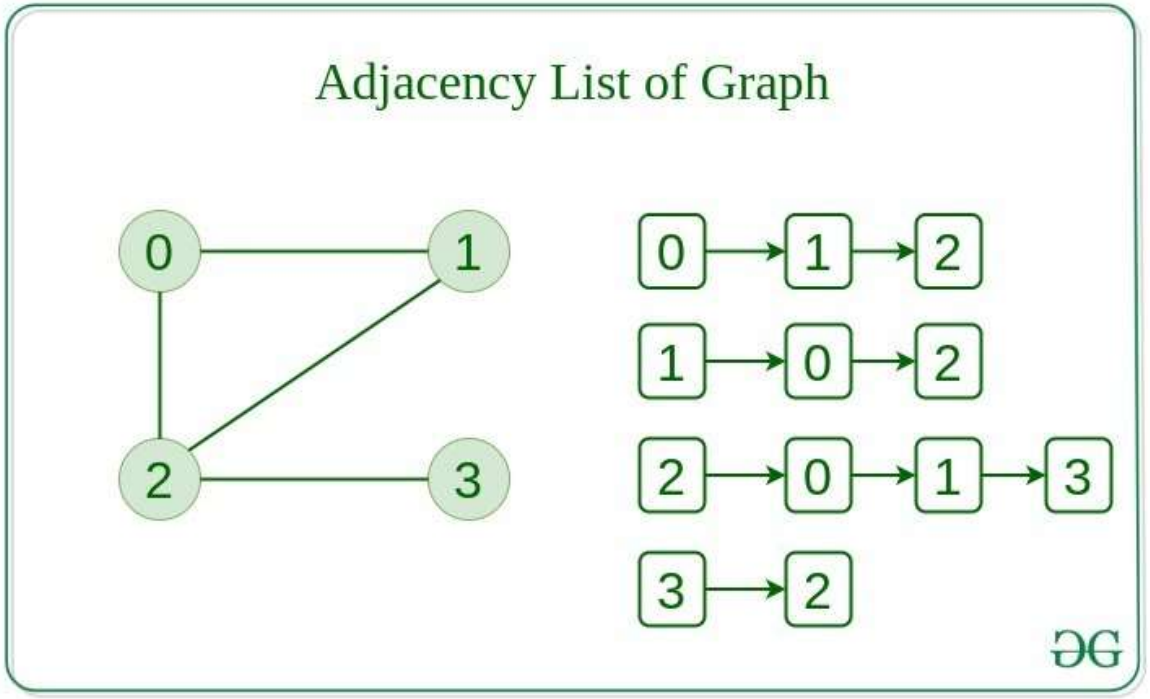
- Graph is stored in the form of the 2D matrix where rows and columns denote vertices.
- Each entry in the matrix represents the weight of the edge between those vertices.



Graphs- Representation

2. **Adjacency List**

- Graph is represented as a collection of linked lists.
- There is an array of pointer which points to the edges connected to that vertex.



Graph-Approaches

Graph approaches can be applied in different ways, depending on the problem at hand. Some common graph-based algorithms and techniques include:

- **Graph traversal algorithms**: These algorithms explore a graph by visiting its nodes and edges in a systematic manner, such as breadth-first search (BFS) and depth-first search (DFS).
- **Shortest path algorithms**: These algorithms find the shortest path between two nodes in a graph, such as Dijkstra's algorithm and Floyd-Warshall algorithm.

U ▪
S T

Graph-Approaches(Cont)

BFS(Breadth First Search)

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Enqueue the starting node A and set its STATUS = 2 (waiting state)

Step 3: Repeat Steps 4 and 5 until QUEUE is empty

Step 4: Dequeue a node N. Process it and set its STATUS = 3 (processed state).

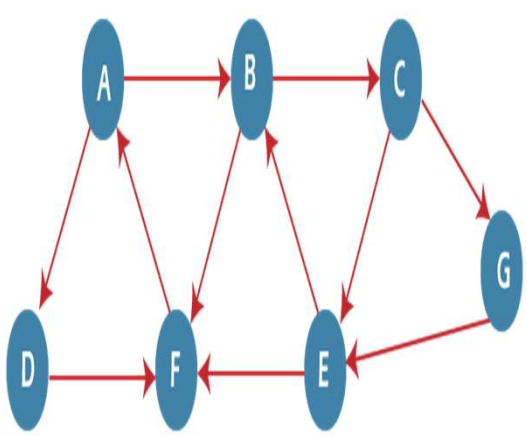
Step 5: Enqueue all the neighbours of N that are in the ready state (whose STATUS = 1) and set

their STATUS = 2

(waiting state)

[END OF LOOP]

Step 6: EXIT



Adjacency Lists

A:B,D
B:C,F
C:E,G
G:E
E:B,F
F:A
D:F

U ▪
S T

Graph-Approaches(Cont)

DFS(Depth First Search)

Step 1: SET STATUS = 1 (ready state) for each node in G

Step 2: Push the starting node A on the stack and set its STATUS = 2 (waiting state)

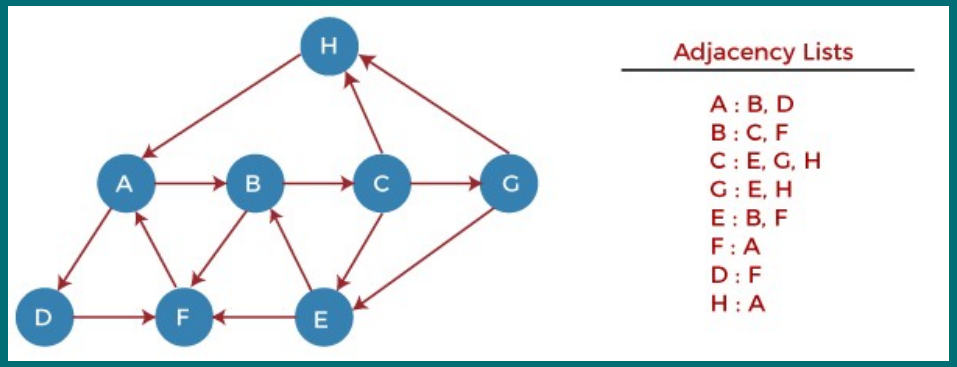
Step 3: Repeat Steps 4 and 5 until STACK is empty

Step 4: Pop the top node N. Process it and set its STATUS = 3 (processed state)

Step 5: Push on the stack all the neighbors of N that are in the ready state (whose STATUS = 1) and set their STATUS = 2 (waiting state)

[END OF LOOP]

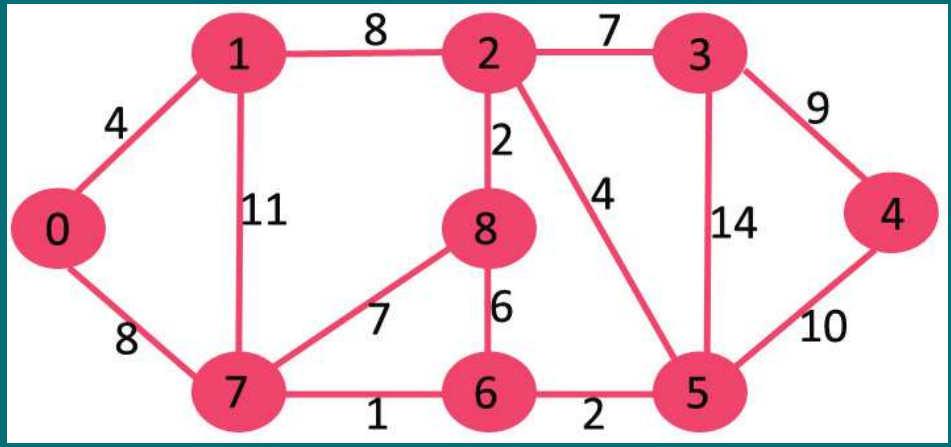
Step 6: EXIT



Graph-Approaches(Cont)

Dijkstra's algorithm

- Step 1:Assign infinite distances to all nodes except the starting node which gets a distance of 0.
- Step 2:Choose the node with the smallest distance and visit it.
- Step 3:Update the distances of all the neighboring nodes by adding the weight of the edge to the smallest distance.
- Step 4:Repeat step 2 and 3 until all nodes have been visited or the smallest tentative distance among the unvisited nodes is infinity.

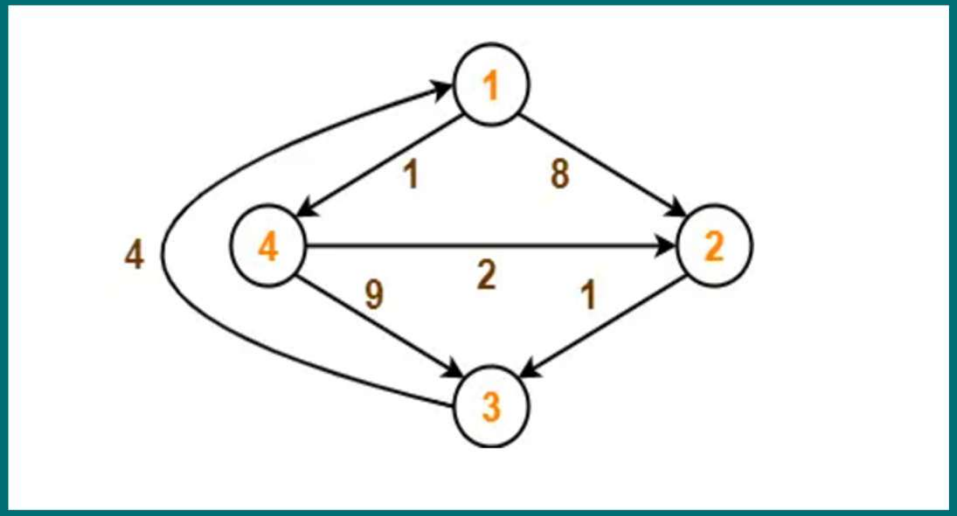


U ▪
S T

Graph-Approaches(Cont)

Floyd-Warshall algorithm

- Initialize a matrix of size n x n, where n is the number of nodes in the graph. The matrix is used to store the distances between all pairs of nodes.
- Initialize the matrix such that the distance between every node i and j is either the weight of the edge (i, j) if it exists, or infinity if it does not.
- For each node k from 1 to n, calculate the distance between all pairs of nodes (i, j) passing through k. If the distance from i to k plus the distance from k to j is less than the current distance from i to j, update the distance.
- After the above calculation, the matrix will contain the shortest distance between all pairs of nodes in the graph.



Graph-Approaches(Cont)

- **Clustering algorithms**: These algorithms group nodes in a graph based on their similarity or proximity, such as k-means clustering and spectral clustering.
- **Centrality measures**: These measures identify important nodes or vertices in a graph, such as degree centrality, betweenness centrality, and closeness centrality, which can be used for identifying influential individuals in a social network or critical nodes in a transportation network.
- **Graph-based machine learning algorithms**: These algorithms utilize graph structures to model and learn from data, such as graph neural networks (GNNs) and random walk-based methods.

Graph-Approaches(Cont)

- **Network flow algorithms**: These algorithms model the flow of resources or information in a network, such as the max-flow min-cut theorem and Ford-Fulkerson algorithm.
- **Graph-based data visualization**: These techniques use graphs to visually represent data, such as node-link diagrams, matrix representations, and force-directed layouts, which can be used for visualizing complex relationships in data.

Graph-Case Study

Here is real-time example of how graph is used:

- **Social Networks**: Social networks such as Facebook, LinkedIn, and Twitter utilize graphs to represent relationships between users.
  - Each user can be represented as a node, and the friendships or connections between users can be represented as edges in an undirected graph.
  - This allows for various graph-based algorithms to be applied for tasks such as finding friends of friends, recommending connections, or identifying communities within the network.

Graph-Benefits

Graphs have several benefits that make them suitable for various applications:

- **Representation of complex relationships**:

  Graphs are a flexible data structure that can represent complex relationships between entities.

- **Efficient data retrieval and traversal**:

  Graphs allow for efficient data retrieval and traversal operations.

Graph-Benefits(Cont)

- **Flexibility and extensibility**:

Graphs are flexible and extensible, allowing for easy addition, deletion, or modification of nodes and edges.

- **Scalability and performance**:

Graph databases and graph-based algorithms are designed for handling large-scale data and can scale horizontally to accommodate growing data volumes.

SEARCHING ALGORITHMS

Searching

- Searching is a process of finding a particular element among several given elements.
- The search is successful if the required element is found.
- Otherwise, the search is unsuccessful.

Searching Algorithms

- Searching Algorithms are a family of algorithms used for the purpose of searching.
- The searching of an element in the given array may be carried out in the following two ways-

Binary Serach

- Binary Search is one of the fastest searching algorithms.
- It is used for finding the location of an element in a linear array.
- It works on the principle of divide and conquer technique.
- Binary Search Algorithm can be applied only on **Sorted arrays**.

- The elements must be arranged in-
  - Either ascending order if the elements are numbers.
  - Or dictionary order if the elements are strings.
  - To apply binary search on an unsorted array,
  - First, sort the array using some sorting technique.
  - Then, use binary search algorithm.

Binary Search Algorithm

- There is a linear array ‘a’ of size ‘n’.
- Binary search algorithm is being used to search an element ‘item’ in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.
- Variables beg and end keeps track of the index of the first and last element of the array or sub array in which the element is being searched at that instant.
- Variable mid keeps track of the index of the middle element of that array or sub array in which the element is being searched at that instant.

Binary Search Algorithm

Begin

Set beg = 0

Set end = n-1

Set mid = (beg + end) / 2

while ((beg <= end) and (a[mid] ≠ item)) do

    if (item < a[mid]) then

```
else
        Set beg = mid + 1
    endif
Set mid = (beg + end) / 2
endwhile
```

```
if (beg > end) then
    Set loc = -1
else
    Set loc = mid
endif
End
```

- Binary Search Algorithm searches an element by comparing it with the middle most element of the array.
- Then, following three cases are possible-

Case-01

- If the element being searched is found to be the middle most element, its index is returned.

Case-02

- If the element being searched is found to be greater than the middle most element,
- then its search is further continued in the right sub array of the middle most element.

Case-03

- If the element being searched is found to be smaller than the middle most element,
- then its search is further continued in the left sub array of the middle most element.
- This iteration keeps on repeating on the sub arrays until the desired element is found
- or size of the sub array reduces to zero.

Time Complexity Analysis

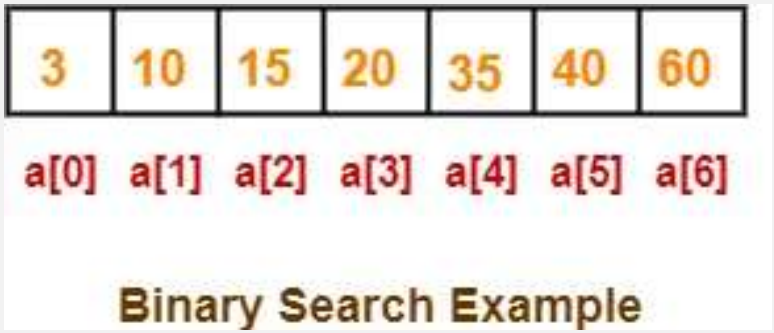
- In each iteration or in each recursive call, the search gets reduced to half of the array.
- So, for n elements in the array, there are $\log_2 n$ iterations or recursive calls.
- **Time Complexity of Binary Search Algorithm is $O(\log_2 n)$.**
- Here, n is the number of elements in the sorted linear array.
- This time complexity of binary search remains unchanged irrespective of the element position even if it is not present in the array.

Space Complexity Analysis

- No extra space is needed, the space complexity of the binary search is O(1).

Example

- Consider-
- We are given the following sorted linear array.
- Element 15 has to be searched in it using Binary Search Algorithm.
- Binary Search Algorithm works in the following steps-



Binary Search Example

- **Step-01:**

- To begin with, we take beg=0 and end=6.
- We compute location of the middle element as-
- Mid
  = (beg + end) / 2
  = (0 + 6) / 2
  = 3
- Here, a[mid] = a[3] = 20 ≠ 15 and beg < end.
- So, we start next iteration.

- **Step-02:**

- Since a[mid] = 20 > 15, so we take end = mid – 1 = 3 – 1 = 2 whereas beg remains unchanged.
- We compute location of the middle element as-
- Mid

  = (beg + end) / 2

  = (0 + 2) / 2

  = 1
- Here, a[mid] = a[1] = 10 ≠ 15 and beg < end.
- So, we start next iteration.

- **Step-03:**

- Since a[mid] = 10 < 15, so we take beg = mid + 1 = 1 + 1 = 2 whereas end remains unchanged.
- We compute location of the middle element as-
- Mid

  = (beg + end) / 2

  = (2 + 2) / 2

  = 2
- Here, a[mid] = a[2] = 15 which matches to the element being searched.
- So, our search terminates in success and index 2 is returned.

Advantages

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

Disadvantages

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.(because of it
  random access nature)

Linear Search

- Linear Search is the simplest searching algorithm.
- It traverses the array sequentially to locate the required element.
- It searches for an element by comparing it with each element of the array one by one.
- So, it is also called as **Sequential Search**.

- Linear Search Algorithm is applied when-
- No information is given about the array.
- The given array is unsorted or the elements are unordered.
- The list of data items is smaller.

Linear Search Algorithm

- Consider
- There is a linear array ‘a’ of size ‘n’.
- Linear search algorithm is being used to search an element ‘item’ in this linear array.
- If search ends in success, it sets loc to the index of the element otherwise it sets loc to -1.

```
Begin

for i = 0 to (n - 1) by 1 do

    if (a[i] = item) then

        set loc = i

        Exit
    endif
end for
set loc = -1
End
```

Time Complexity Analysis

- **Best case**
- In the best possible case,
- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes O(1) operations.

Worst Case

- In the worst possible case,
- The element being searched may be present at the last position or not present in the array a all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus, in worst case, linear search algorithm takes O(n) operations.
- **Time Complexity of Linear Search Algorithm is O(n).**
- Here, n is the number of elements in the linear array.

Space Complexity Analysis

- The space complexity of the linear search is O(1), as we don't need any auxiliary space for the algorithm.

Example

- Consider-
- We are given the following linear array.
- Element 15 has to be searched in it using Linear Search Algorithm.

- Linear Search algorithm compares element 15 with all the elements of the array one by one.
- It continues searching until either the element 15 is found or all the elements are searched.

- Linear Search Algorithm works in the following steps-

- **Step-01:**
- It compares element 15 with the 1st element 92.
- Since 15 ≠ 92, so required element is not found.
- So, it moves to the next element.

- **Step-02:**
- It compares element 15 with the 2nd element 87.
- Since 15 ≠ 87, so required element is not found.
- So, it moves to the next element.

- **Step-03:**
- It compares element 15 with the 3rd element 53.
- Since 15 ≠ 53, so required element is not found.
- So, it moves to the next element.

- **Step-04:**
- It compares element 15 with the 4th element 10.
- Since 15 ≠ 10, so required element is not found.
- So, it moves to the next element.

- **Step-05:**
- It compares element 15 with the 5^{th} element 15.
- Since 15 = 15, so required element is found.
- Now, it stops the comparison and returns index 4 at which element 15 is present.

Advantages

- Easy to understand
- No special data structure required
- Can be used on unsorted data
- No additional memory required
- Not affected by data size

Disadvantages

- Time-consuming
- Not suitable for large data sets
- Not suitable for ordered data
- No additional memory required
- Not suitable for repetitive tasks
- Not suitable for real-time applications

SORTING ALGORITHMS

SORTING

- Sorting is the process of arranging the elements of an array so that they can be placed either in ascending or descending order.
- Sorting algorithm depends on
- the size of the data set,
- the time and space complexity of the algorithm,
- the specific requirements of the application.

TYPES OF SORTING

1.SELECTION SORT

2.INSERTION SORT

3.BUBBLE SORT

4.MERGE SORT

5.QUICK SORT

1.SELECTION SORT

SELECTION SORT

Selection sort is a simple sorting algorithm that sorts an array by repeatedly finding the minimum element from the unsorted portion of the array and placing it at the beginning of the unsorted portion.

- Time complexity = $O(N^2)$

- Space Complexity=O(1)

AN OVERVIEW

- The algorithm maintains two subarrays in each array.
- The subarray which already sorted.
- The remaining subarray was unsorted

- In every iteration, the minimum element from the unsorted subarray is selected and moved to the beginning of sorted subarray.

- After every iteration sorted subarray size increase by one and unsorted subarray size decrease by one.

- After N iterations, we will get sorted array.

Algorithm

- Initialize minimum value(**min**) to location 0.
- Traverse the array to find the minimum element in the array.
- If any element smaller than **min** is found then swap both values.
- Increment **min** to point to the next element.
- Repeat until the array is sorted.

Example

- Consider the following array as an example:

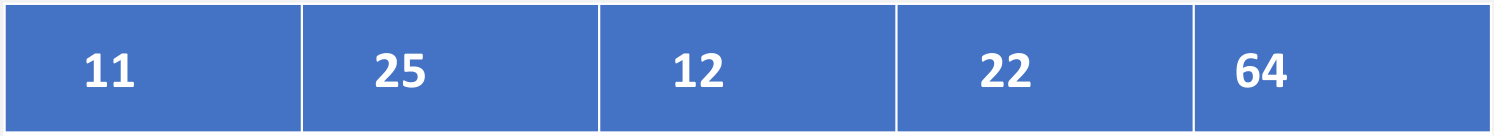
- ***First pass:***
- The whole array is traversed from index 0 to 4 sequentially. The first position where **64** is stored presently, after traversing it is clear that **11** is the lowest value.

11	25	12	22	64

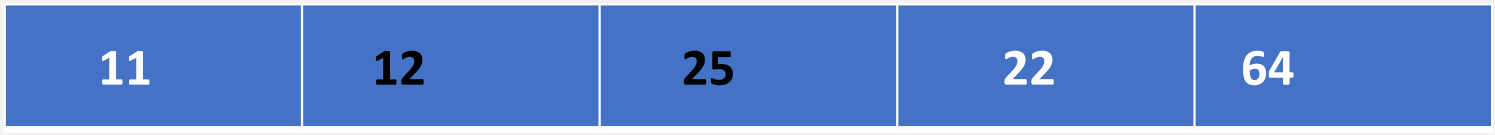
•Thus, replace 64 with 11. After one iteration **11**, which happens to be the least value in the array, tends to appear in the first position of the sorted list.

Second Pass:

For the second position, where 25 is present, again traverse the rest of the array in a sequential manner.



After traversing,**12** is the second lowest value in the array and it should appear at the second place in the array, thus swap these values



- **Third Pass:**

- Now, for third place, where **25** is present again traverse the rest of the array and find the third least value present in the array.

11	12	25	22	64

- While traversing, **22** came out to be the third least value and it should appear at the third place in the array, thus swap **22** with element present at third position.

11	12	22	25	64

- **Fourth pass:**

- Similarly, for fourth position traverse the rest of the array and find the fourth least element in the array
- A[illegible]on.

11	12	22	25	64

- Hence the resulted array is the sorted array,

11	12	22	25	64

ADVANTAGES

- Works well with small datasets.
- It can be used in limited memory environments as it requires minimum extra memory.
- Preserves the relative order of items with equal keys which means it is stable.
- Relatively efficient and easy to implement

DISADVANTAGES

- Does not work well on large datasets.
- It has poor cache performance
- It does not handle data with many duplicates well, as it makes many unnecessary swaps.
- The selection sort algorithm needs to iterate over the list multiple times, thus it can lead to an unbalanced branch.

2. INSERTION SORT

INSERTION SORT

- Insertion sort algorithm is a basic sorting algorithm that sequentially sorts each item in the final sorted array or list.

- Time Complexity :$O(n^2)$
- Space Complexity :O(1)

CHARACTERISTICS OF INSERTION SORT

- This algorithm is one of the simplest algorithm with simple implementation.
- Insertion sort is efficient for small data values.
- Insertion sort is adaptive in nature.
- That is, it is appropriate for data sets which are already partially sorted.

ALGORITHM

- **Step 1**: If the element is the first element, assume that it is already sorted. Return 1.
- **Step 2:** Pick the next element and store it separately in a **key.**
- **Step 3:** Now, compare the **key** with all elements in the sorted array.
- **Step 4**: If the element in the sorted array is smaller than the
- current element, then move to the next element.
- Else, shift greater elements in the array towards the right.
- **Step 5**: Insert the value.
- **Step 6**: Repeat until the array is sorted.

EXAMPLE

- Consider an array of elements: {12, 11, 13, 5, 6}

- **First Pass:**
- Initially, the first two elements of the array are compared in insertion sort.

12	11	13	5	6

- Here, 12 is greater than 11 hence they are not in the ascending order and 12 is not at its correct position. Thus, swap 11 and 12.
- So, for now 11 is stored in a sorted sub-array.

11	12	13	5	6

- **Second Pass:**
- Now, move to the next two elements and compare them

11	12	13	5	6

- Here, 13 is greater than 12, thus both elements seems to be in ascending order, hence, no swapping will occur. 12 also stored in a sorted sub-array along with 11

11	12	13	5	6

- **Third Pass:**
- Now, two elements are present in the sorted sub-array which are 11 and 12
- Moving forward to the next two elements which are 13 and 5

11	12	13	5	6

- Both 5 and 13 are not present at their correct place so swap them

11	12	5	13	6

- After swapping, elements 12 and 5 are not sorted, thus swap again

11	5	12	13	6

- Here, again 11 and 5 are not sorted, hence swap again

5	11	12	13	6

- Here, 5 is at its correct position

- **Fourth Pass:**
- Now, the elements which are present in the sorted sub-array are 5, 11 and 12
- Moving to the next two elements 13 and 6

5	11	12	13	6

- Clearly, they are not sorted, thus perform swap between both

5	11	12	6	13

- Now, 6 is smaller than 12, hence, swap again

- Here, also swapping makes 11 and 6 unsorted hence, swap again

- Finally, the array is completely sorted.

DISADVANTAGES

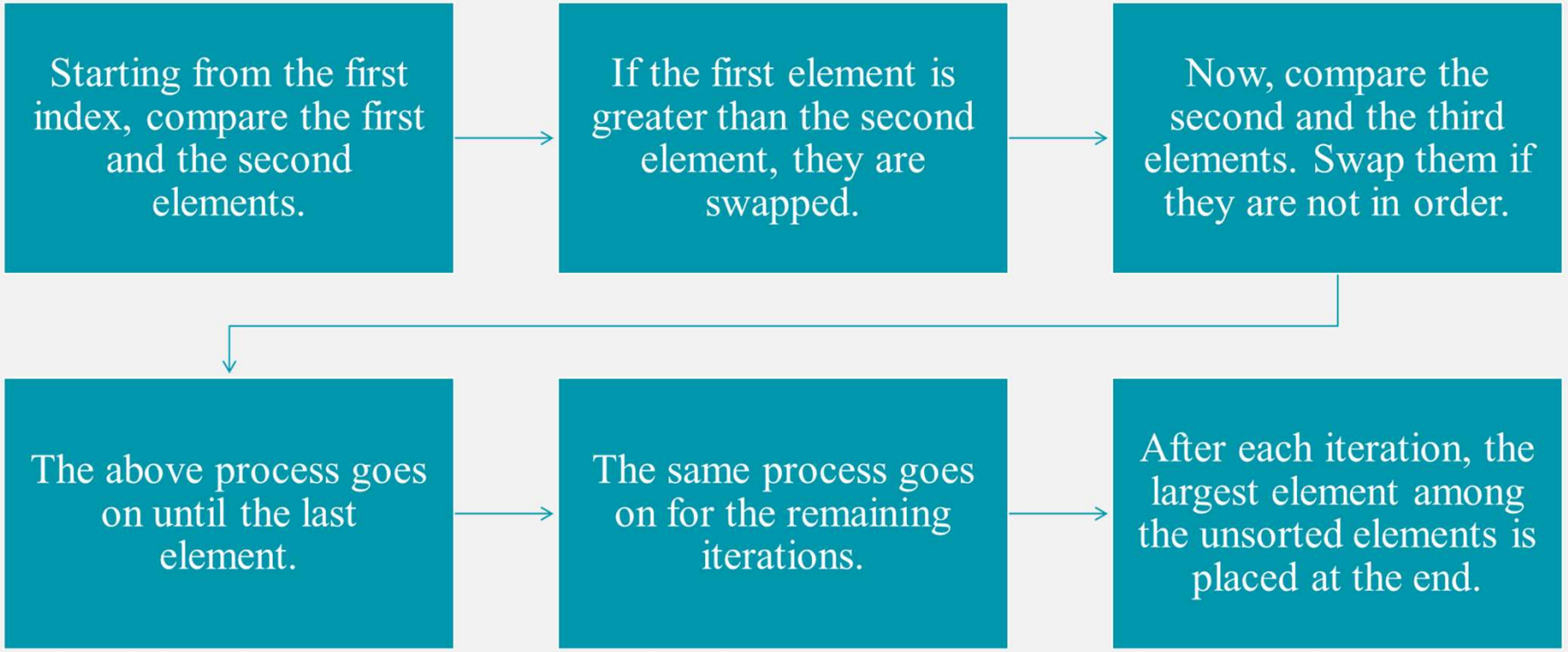
- Inefficient for large lists (time complexity increases exponentially with the size of the list).
- Not suitable for lists with many duplicate elements.
- Not adaptable to different data structures.

3. BUBBLE SORT

BUBBLE SORT

- Simplest sorting algorithm that works by repeatedly swapping the adjacent elements.
- This algorithm is not suitable for large data sets.
- Average and worst-case time complexity is quite high.
- The average and worst-case complexity of Bubble sort is **O(n2),** where **n** is the number of items.
- Time complexity of best case is O(n)

WORKING PROCESS



ALGORITHM

```
Begin BubbleSort( list)
  For all elements of list
   if list[i]>list[i+1]
    swap(list[i], list[i+1])
      end if
     end for
     return list
     end BubbleSort
```

ADVANTAGES

- Bubble sort is easy to understand and implement.
- It does not require any additional memory space.
- It’s adaptability to different types of data.

DISADVANTAGES

- Time complexity of O(n^2) which makes it very slow for large data sets.
- It is not a stable sorting algorithm, elements with the same key value may not maintain their relative order.

EXAMPLES

- **Input:** arr[] = {6, 3, 0, 5}
- **First Pass:**
- Bubble sort starts with very first two elements, comparing them to check which one is greater.
  - (6 3 0 5) –> (**3 6** 0 5), Here, algorithm compares the first two elements, and swaps since 6 > 3.
  - (3 **6 0** 5) –> (3 **0 6** 5), Swap since 6 > 0
  - (3 0 **6 5**) –> (3 0 **5 6**), Swap since 6 > 5

- **Second Pass:**
- Now, during second iteration it should look like this:
    - (**3 0** 5 6) –> (**0 3** 5 6), Swap since 3 > 0
    - (0 **3 5** 6) –> (0 **3 5** 6), No change as 5 > 3
- **Third Pass:**
- Now, the array is already sorted, but our algorithm does not know if it is completed.
- The algorithm needs one **whole** pass without **any** swap to know it is sorted.
    - (**0 3** 5 6) –> (**0 3** 5 6), No change as 3 > 0

    Array is now sorted and no more pass will happen.

4. MERGE SORT

MERGE SORT

- **Merge sort** is a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.
- Time Complexity: O(N log N)
- Space Complexity: O(N)

WORKING PROCESS

Continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop.

If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves.

Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

ALGORITHM

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
   return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

ADVANTAGES

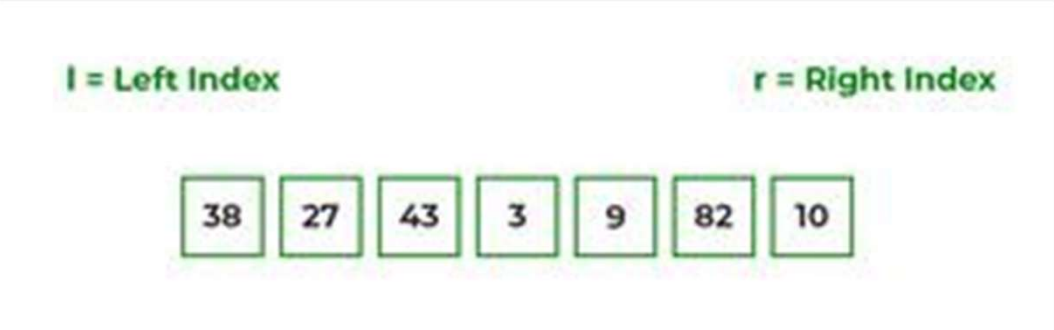
- Sort large arrays relatively quickly.
- Order of elements with equal values is preserved during the sort.
- Relatively efficient and easy to implement.

DISADVANTAGES

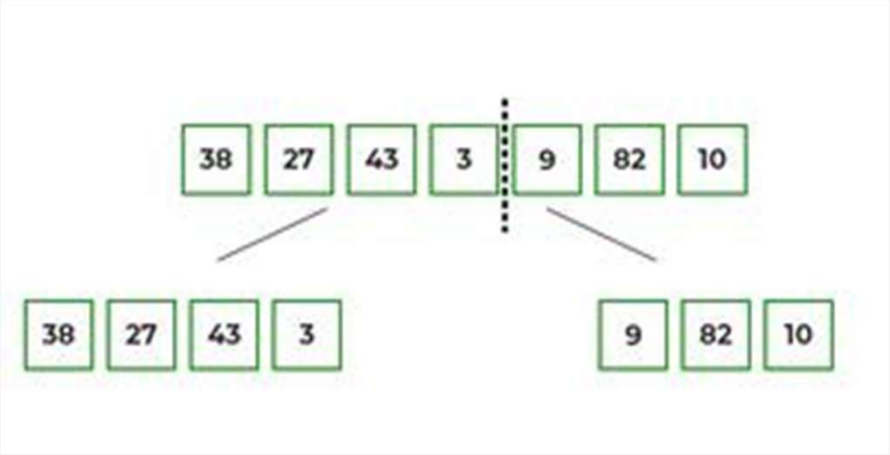
- Requires an additional memory space to store the subarrays that are used
- Requires more code to implement
- Slower compared to the other sort algorithms for smaller tasks.

EXAMPLES

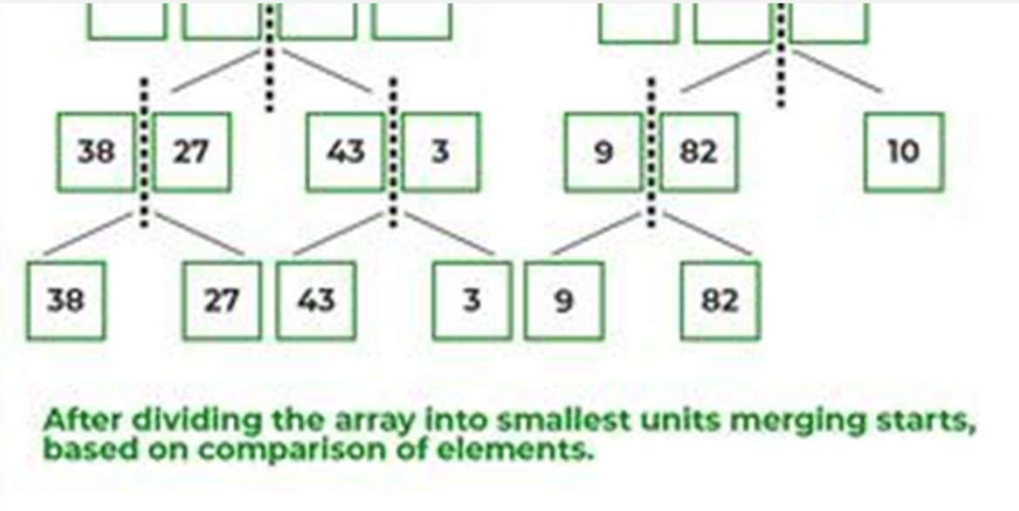
- consider an array
- arr[] = {38, 27, 43, 3, 9, 82, 10}
- check if the left index of array is less than the right index, if yes then calculate its midpoint.



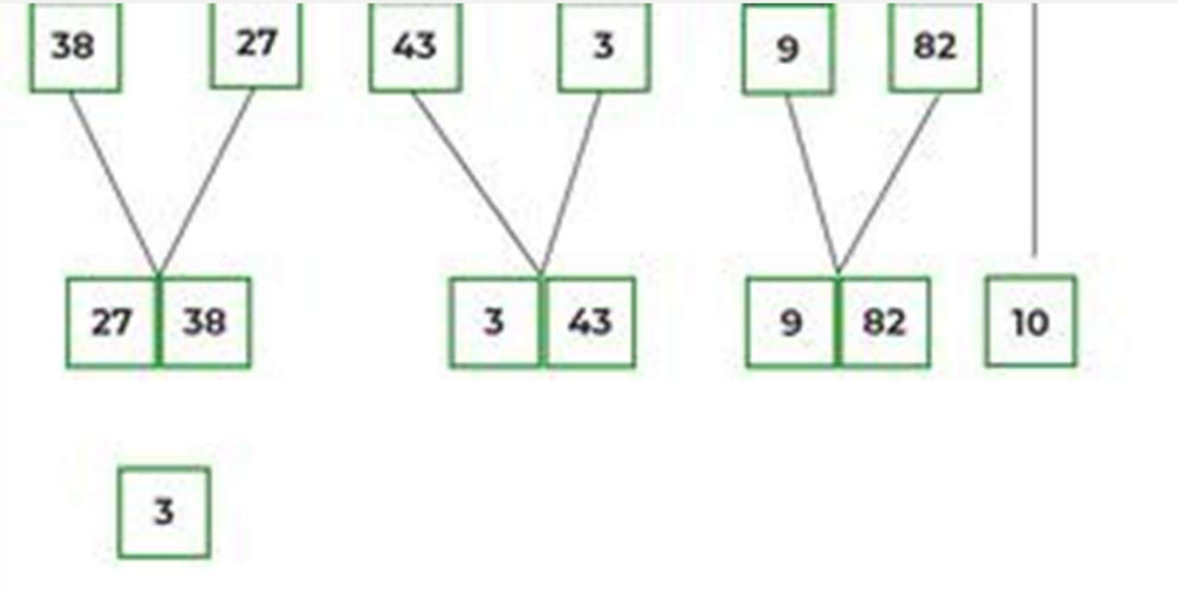
- Array of 7 items is divided into two arrays of size 4 and 3 respectively.
- If left index is less than the right index for both arrays, then again calculate mid points for both the arrays.



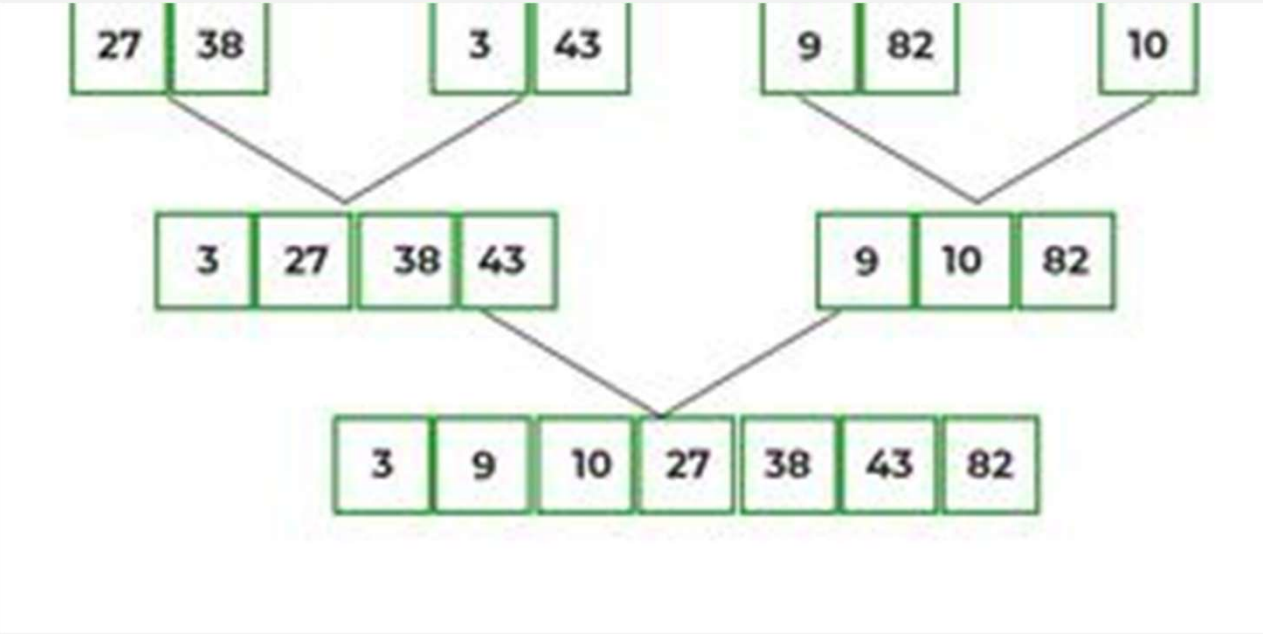
- Further divide these two arrays into further halves, until further division is not possible.

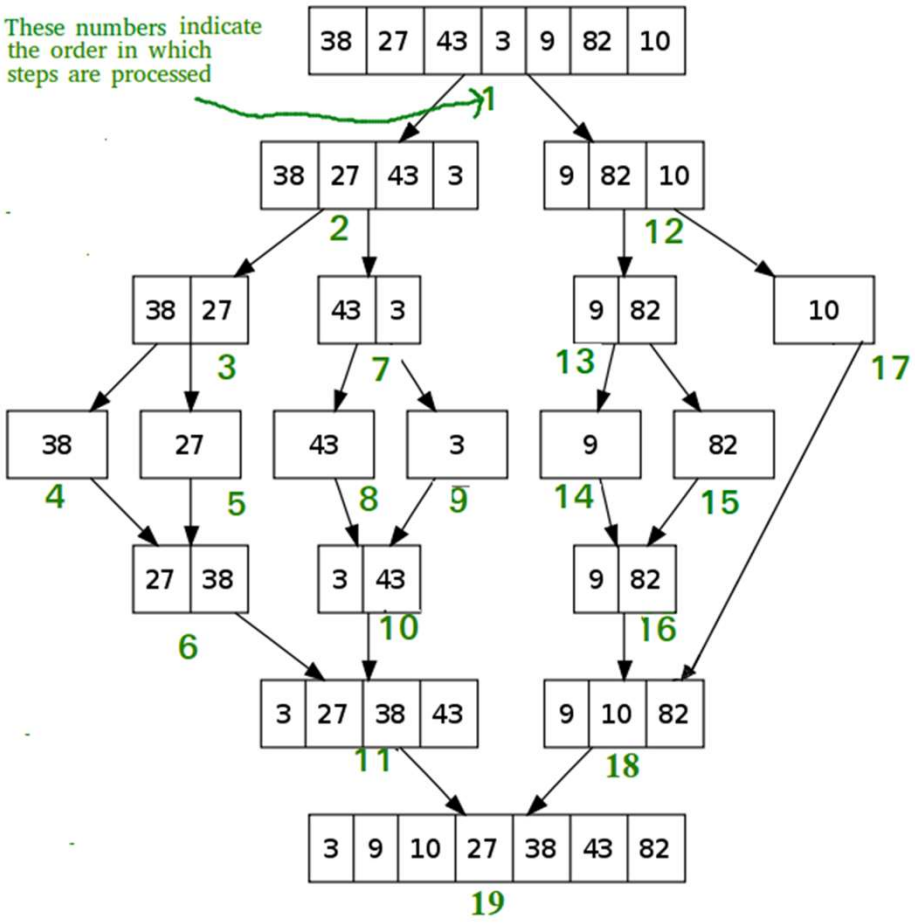


•After dividing the array into smallest units, start merging the elements again based on comparison of size of elements



- After the final merging




These numbers indicate the order in which steps are processed
38 27 43 3 9 82 10
1
38 27 43 3
9 82 10
2
12
38 27
43 3
9 82
10
3
7
13
17
38
27
43
3
9
82
4
5
8
9
14
15
27 38
3 43
9 82
6
10
16
3 27 38 43
9 10 82
11
18
3 9 10 27 38 43 82
19

5.QUICK SORT

QUICK SORT

QuickSort is a sorting algorithm based on the Divide And Conquer that picks an element as a pivot and partitions the given array around the picked pivot by placing the pivot in its correct position in the sorted array.

- Time Complexity: O(N logN)
- Worst Case Complexity:$O(n^2)$
- Space complexity of quicksort is O(n*logn)

Working

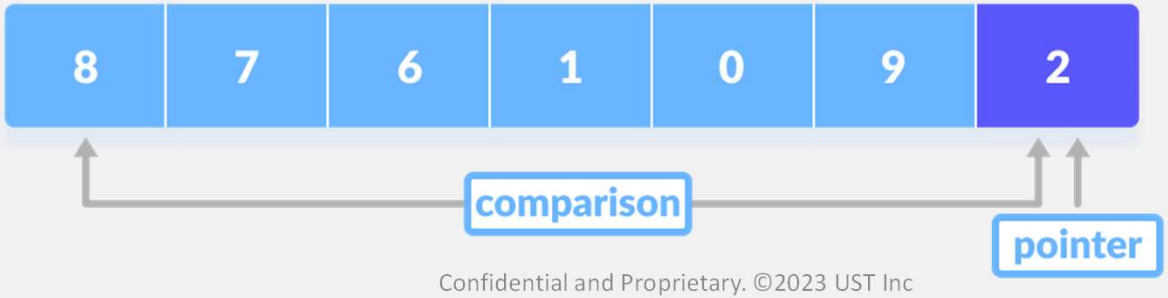
1.Select the Pivot Element

A pivot element is selected from different positions. Here, we will be selecting the rightmost element of the array as the pivot element.

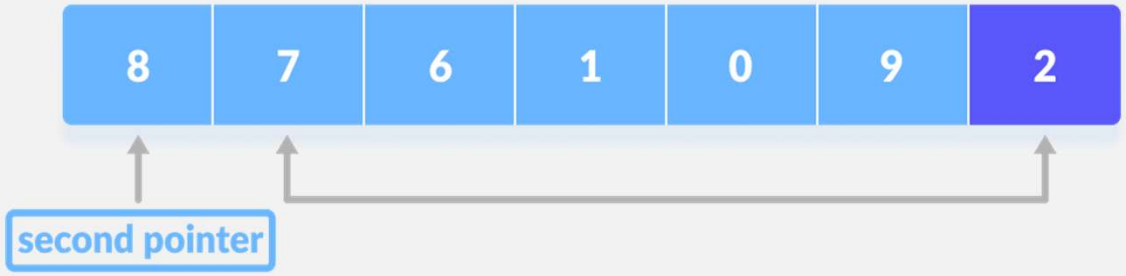
2. Rearrange the Array

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

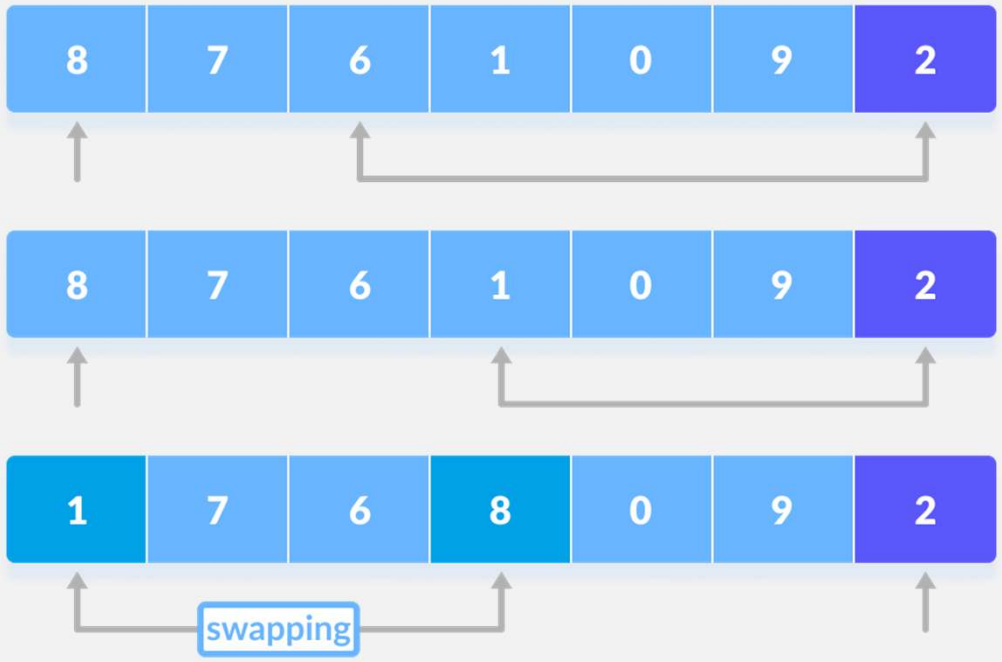
A pointer is fixed at the pivot element. The pivot element is compared with the elements beginning from the first index.



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

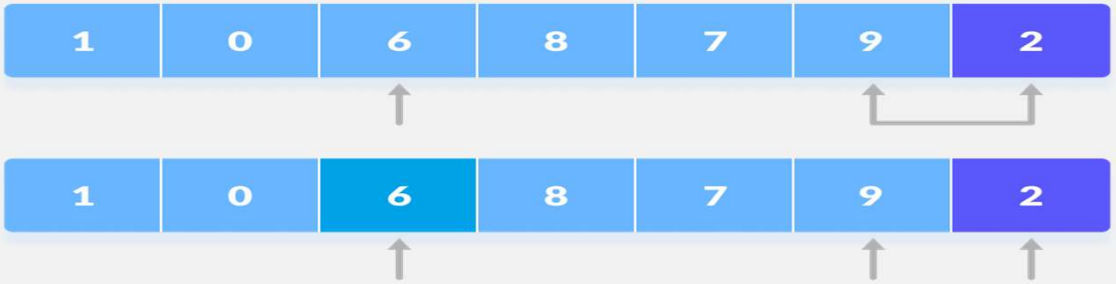


Now, pivot is compared with other elements. If an element smaller than the pivot element is reached, the smaller element is swapped with the greater element found earlier.



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

The process goes on until the second last element is reached



Finally, the pivot element is swapped with the second pointer

3.Divide Subarrays

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

Advantages

- It is a divide-and-conquer algorithm that makes it easier to solve problems.
- It is efficient on large data sets.
- It has a low overhead, as it only requires a small amount of memory to function.

U ▪
S T

THANK YOU

BATCH 3