

MCA 202A- Data Structures

UNIT I: Linear Data Structures: Abstract Data Types - Asymptotic Notations: Big-Oh, Omega and Theta – Best, Worst and Average case Analysis: Definition and an example – Arrays and its representations – Stacks and Queues – Linked lists – Linked list-based implementation of Stacks and Queues – Evaluation of Expressions – Linked list based polynomial addition.

UNIT II: Non-Linear Data Structures; Trees – Binary Trees – Binary tree representation and traversals– Threaded binary trees – Binary Search Tree-Insertion, Deletion, Searching, AVL Trees Insertion, Deletion, Searching, Red-Black-Trees- Insertion, Deletion and Splay Trees.

UNIT III: Graph and its representations – Graph Traversals DFS and BFS – Connected components, Applications of Graphs-Minimum cost spanning tree using Kruskal's algorithm, Dijkstra's algorithm for Single Source Shortest Path Problem.

UNIT IV: Sorting: Insertion sort – Merge sort – Quick sort – Heap sort – Radix Sort- Comparison of sorting algorithms in terms of Complexity - Sorting with disks – k-way merging – Sorting with tapes – Polyphase merge.

UNIT V: Searching And Indexing: Linear Search – Binary Search - Hash tables – Overflow handling Cylinder Surface Indexing – Hash Index – B-Tree Indexing, B+ Trees

UNIT I

Linear Data Structures:

Linear Data Structures are a type of data structure in computer science where data elements are arranged sequentially or linearly. Each element has a previous and next adjacent, except for the first and last elements.

Characteristics of Linear Data Structure:

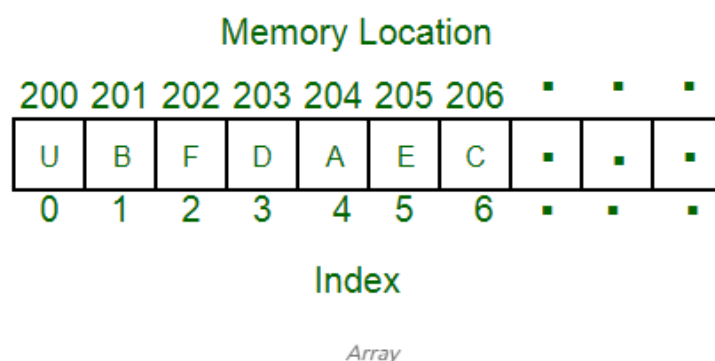
- **Sequential Organization:** In linear data structures, data elements are arranged sequentially, one after the other. Each element has a unique predecessor (except for the first element) and a unique successor (except for the last element)
- **Order Preservation:** The order in which elements are added to the data structure is preserved. This means that the first element added will be the first one to be accessed or removed, and the last element added will be the last one to be accessed or removed.
- **Fixed or Dynamic Size:** Linear data structures can have either fixed or dynamic sizes. Arrays typically have a fixed size when they are created, while other structures like linked lists, stacks, and queues can dynamically grow or shrink as elements are added or removed.
- **Efficient Access:** Accessing elements within a linear data structure is typically efficient. For example, arrays offer constant-time access to elements using their index.

Linear data structures are commonly used for organising and manipulating data in a sequential fashion. Some of the most common linear data structures include:

- **Arrays:** A collection of elements stored in contiguous memory locations.
- **Linked Lists:** A collection of nodes, each containing an element and a reference to the next node.
- **Stacks:** A collection of elements with Last-In-First-Out (LIFO) order.
- **Queues:** A collection of elements with First-In-First-Out (FIFO) order.

1. Array

An array is a collection of items of same data type stored at contiguous memory locations.



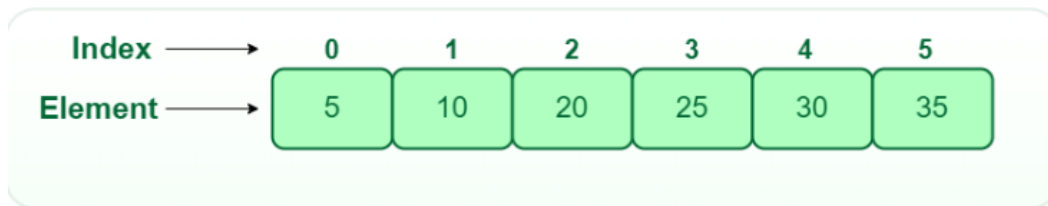
Characteristics of Array Data Structure:

- **Homogeneous Elements:** All elements within an array must be of the same data type.
- **Contiguous Memory Allocation:** In most programming languages, elements in an array are stored in contiguous (adjacent) memory locations.

- **Zero-Based Indexing:** In many programming languages, arrays use zero-based indexing, which means that the first element is accessed with an index of 0, the second with an index of 1, and so on.
- **Random Access:** Arrays provide constant-time ($O(1)$) access to elements. This means that regardless of the size of the array, it takes the same amount of time to access any element based on its index.

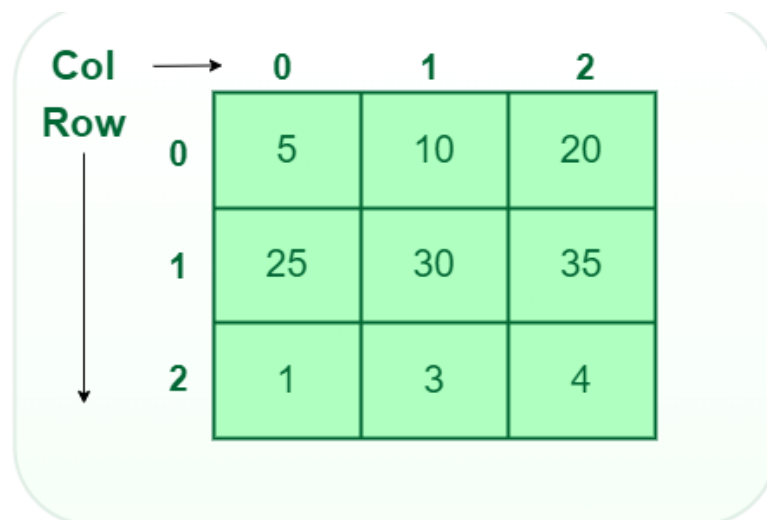
Types of arrays:

- **One-Dimensional Array:** This is the simplest form of an array, which consists of a single row of elements, all of the same data type. Elements in a 1D array are accessed using a single index.



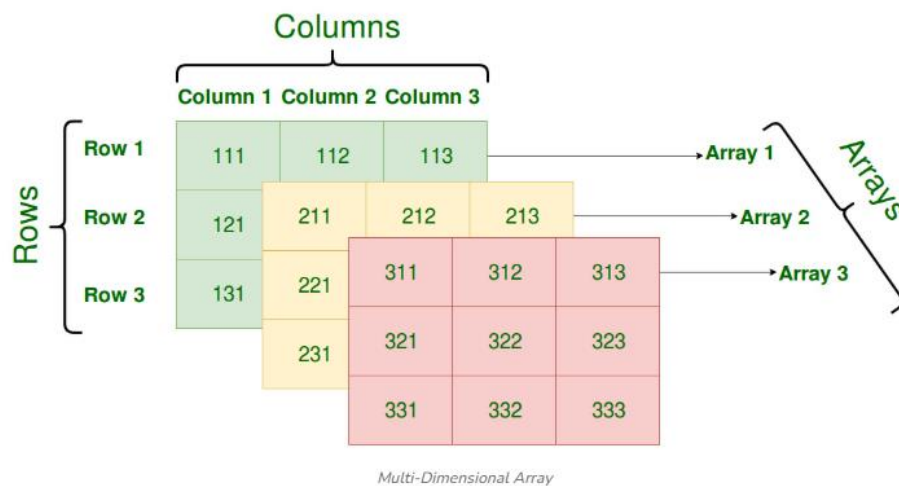
One-Dimensional Array

- **Two-Dimensional Array:** A two-dimensional array, often referred to as a matrix or 2D array, is an array of arrays. It consists of rows and columns, forming a grid-like structure. Elements in a 2D array are accessed using two indices, one for the row and one for the column.



Two-Dimensional Array:

- **Multi-Dimensional Array:** Arrays can have more than two dimensions, leading to multi-dimensional arrays. These are used when data needs to be organized in a multi-dimensional grid.



Types of Array operations:

- **Accessing Elements:** Accessing a specific element in an array by its index is a constant-time operation. It has a time complexity of $O(1)$.
- **Insertion:** Appending an element to the end of an array is usually a constant-time operation, $O(1)$ but insertion at the beginning or any specific index takes $O(n)$ time because it requires shifting all of the elements.
- **Deletion:** Same as insertion, deleting the last element is a constant-time operation, $O(1)$ but deletion of element at the beginning or any specific index takes $O(n)$ time because it requires shifting all of the elements.
- **Searching:** Linear Search takes $O(n)$ time which is useful for unsorted data and Binary Search takes $O(\log n)$ time which is useful for sorted data.

2. Linked List

A **Linked List** is a linear data structure which looks like a chain of nodes, where each node contains a **data** field and a **reference(link)** to the next node in the list. Unlike Arrays, Linked List elements are not stored at a contiguous location.

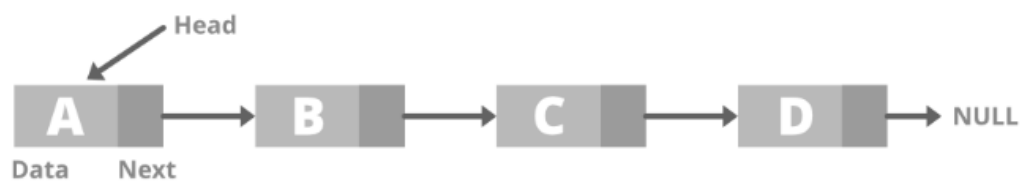
Common Features of Linked List:

- **Node:** Each element in a linked list is represented by a node, which contains two components:
 - **Data:** The actual data or value associated with the element.
 - **Next Pointer(or Link):** A reference or pointer to the next node in the linked list.
- **Head:** The first node in a linked list is called the "head." It serves as the starting point for traversing the list.
- **Tail:** The last node in a linked list is called the "tail."

Types of Linked Lists:

- **Singly Linked List:** In this type of linked list, every node stores the address or reference of the next node in the list and the last node has the next address or reference as NULL. For example: 1->2->3->4->NULL

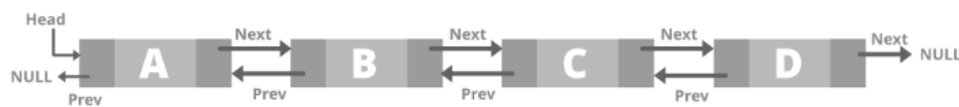
Singly Linked List



Singly Linked List

- **Doubly Linked Lists:** In a doubly linked list, each node has two pointers: one pointing to the next node and one pointing to the previous node. This bidirectional structure allows for efficient traversal in both directions.

Doubly Linked List



Doubly Linked Lists

- **Circular Linked Lists:** A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle, there is no NULL at the end.

Circular Linked List



Circular Linked Lists

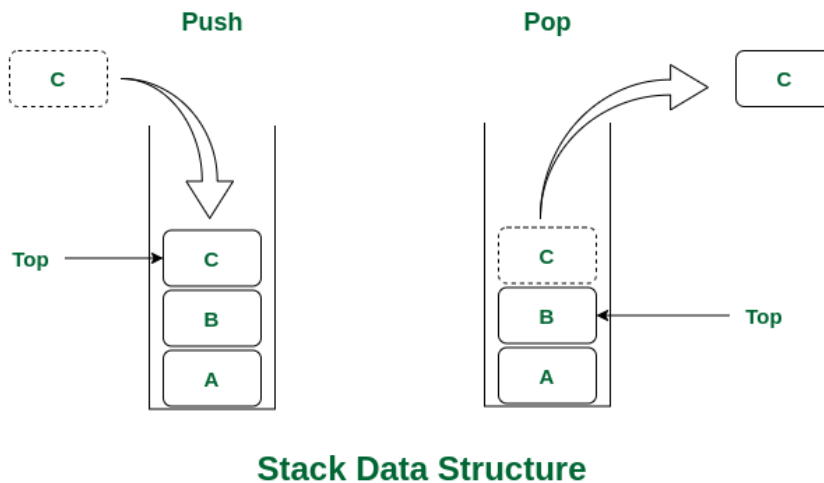
Types of Linked List operations:

- **Accessing Elements:** Accessing a specific element in a linked list takes $O(n)$ time since nodes are stored in non contiguous locations so random access is not possible.
- **Searching:** Searching of a node in linked list takes $O(n)$ time as whole list needs to be traversed in worst case.

- **Insertion:** Insertion takes $O(1)$ time if we are at the position where we have to insert an element.
- **Deletion:** Deletion takes $O(1)$ time if we know the position of the element to be deleted.

3. Stack Data Structure

A **stack** is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle, meaning that the last element added to the stack is the first one to be removed.



Types of Stacks:

- **Fixed Size Stack:** As the name suggests, a fixed size stack has a fixed size and cannot grow or shrink dynamically. If the stack is full and an attempt is made to add an element to it, an overflow error occurs. If the stack is empty and an attempt is made to remove an element from it, an underflow error occurs.
- **Dynamic Size Stack:** A dynamic size stack can grow or shrink dynamically. When the stack is full, it automatically increases its size to accommodate the new element, and when the stack is empty, it decreases its size. This type of stack is implemented using a **linked list**, as it allows for easy resizing of the stack.

Stack Operations:

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

4. Queue Data Structure

A **queue** is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. In a queue, the first element added is the first one to be removed.



Queue Data Structure

Types of Queue:

- **Input Restricted Queue:** This is a simple queue. In this type of queue, the input can be taken from only one end but deletion can be done from any of the ends.
- **Output Restricted Queue:** This is also a simple queue. In this type of queue, the input can be taken from both ends but deletion can be done from only one end.
- **Circular Queue:** This is a special type of queue where the last position is connected back to the first position. Here also the operations are performed in FIFO order. To know more refer this.
- **Double-Ended Queue (Deque):** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. To know more refer this.
- **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. To know more refer this.

Queue Operations:

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

Advantages of Linear Data Structures

- **Efficient data access:** Elements can be easily accessed by their position in the sequence.
- **Dynamic sizing:** Linear data structures can dynamically adjust their size as elements are added or removed.

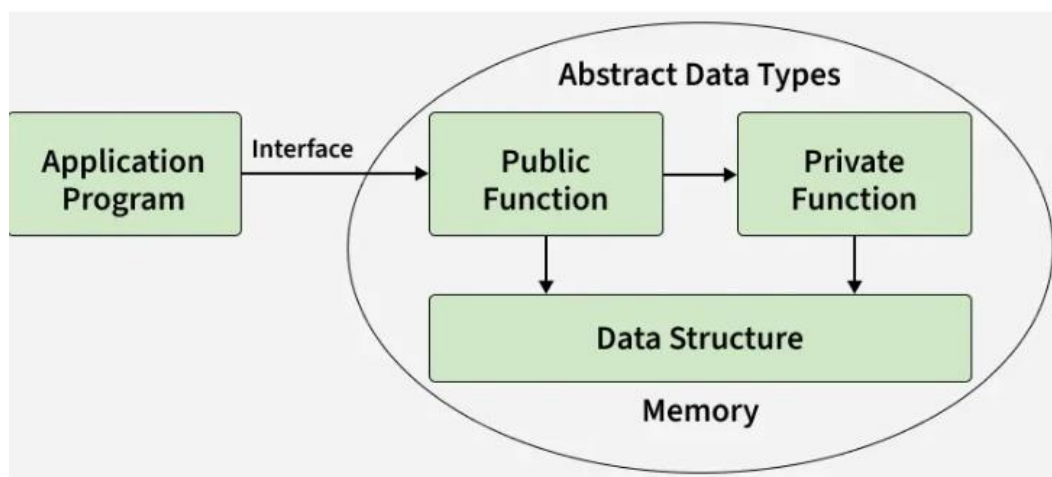
- **Ease of implementation:** Linear data structures can be easily implemented using arrays or linked lists.
- **Versatility:** Linear data structures can be used in various applications, such as searching, sorting, and manipulation of data.
- **Simple algorithms:** Many algorithms used in linear data structures are simple and straightforward.

Disadvantages of Linear Data Structures

- **Limited data access:** Accessing elements not stored at the end or the beginning of the sequence can be time-consuming.
- **Memory overhead:** Maintaining the links between elements in linked lists and pointers in stacks and queues can consume additional memory.
- **Complex algorithms:** Some algorithms used in linear data structures, such as searching and sorting, can be complex and time-consuming.
- **Inefficient use of memory:** Linear data structures can result in inefficient use of memory if there are gaps in the memory allocation.
- **Unsuitable for certain operations:** Linear data structures may not be suitable for operations that require constant random access to elements, such as searching for an element in a large dataset.

Abstract Data Type (ADT):

Abstract Data Type (ADT) is a conceptual model. They define what a data structure does without dictating how it does it. By separating the logical properties from the physical implementation, ADTs allow for cleaner, more maintainable code.



As depicted in the diagram, the following features of ADTs are highlighted:

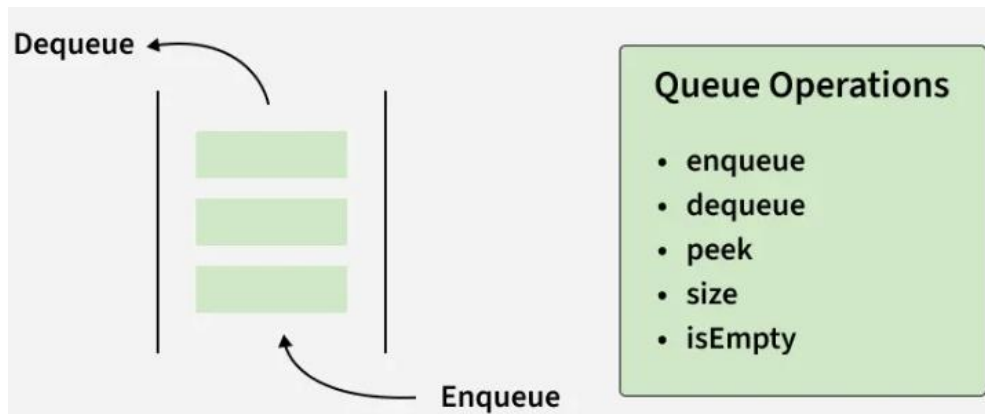
- Only the essentials are exposed, omitting implementation details (Abstraction).
- Data and its related operations are bundled into a single unit (Encapsulation).
- You can change the underlying implementation (e.g., from an Array to a Linked List) without affecting how the rest of the program uses the ADT (Data Independence).

Common Examples of ADTs:

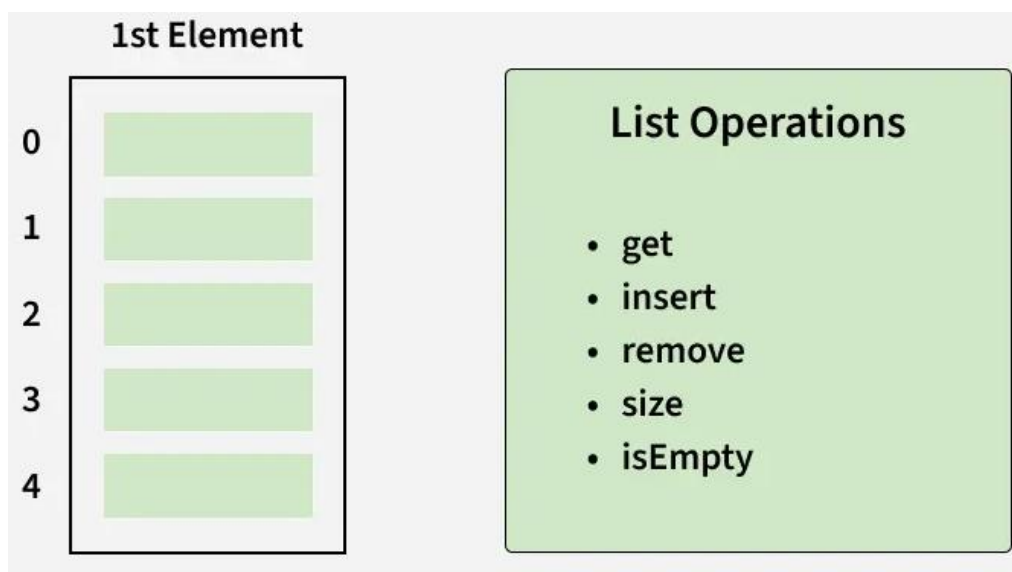
While ADTs define the logical model for data handling, several standard structures exemplify this concept. The following examples adhere to the ADT principle by providing specific operations while concealing the underlying storage logic.

Queue: A linear structure following the FIFO (First In, First Out) principle.

- enqueue(value): Adds element to the bottom.
- dequeue(): Removes top element
- peek(): Returns top element.

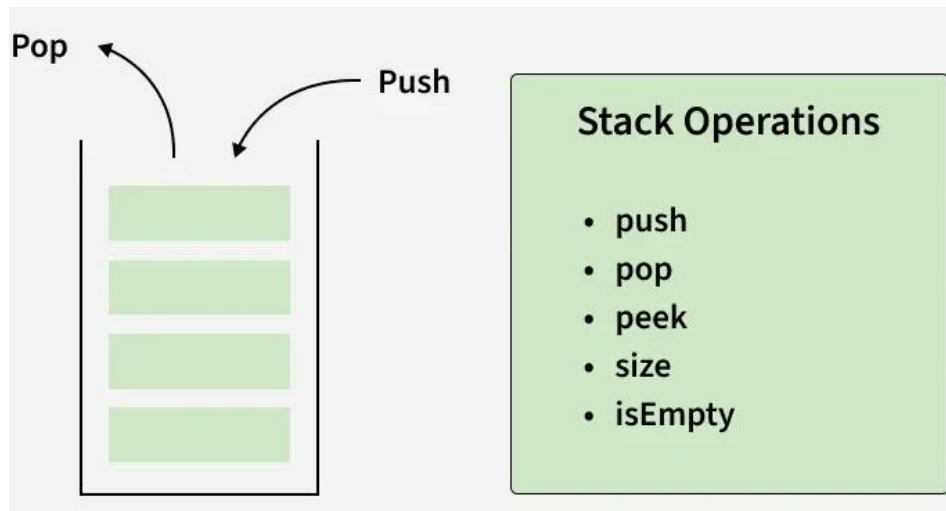


List: A sequential collection of elements.



- insert(value,index): Adds element to the given position.
- remove(value,index): Removes element from given position.
- get(index): Returns value at given position.

Stack: A linear structure following the LIFO (Last In, First Out) principle.



- `push(value)` : Adds element to the top.
- `pop()`: Removes top element.
- `peek()`: Returns top element.

Differences Between ADTs and UDTs

Abstract Data Types (ADTs) define what operations are allowed, while User Defined Types (UDTs) define how data is stored and implemented.

Abstract Data Types (ADTs)	User-Defined Data Types (UDTs)
Focuses on allowed operations and their behaviour, without implementation details.	Focuses on how data is organized in memory and how operations are executed.
Provides an abstract model for defining data structures conceptually.	Allows programmers to create concrete implementations of data structures using primitive types.
Does not define the implementation of operations or data structure.	Specifies how to create and organize data types for implementing the structure.
List, Stack, Queue	Classes, Structs, Enumerations

Advantages:

- **Modularity:** Simplifies complex systems by breaking them into smaller, manageable pieces.
- **Security & Integrity:** Information hiding prevents unauthorized or accidental modification of internal data.

- **Reusability:** Once defined, the interface can be reused across different projects regardless of the backend logic.

Disadvantages:

- **Performance Overhead:** The layers of abstraction can sometimes lead to slightly higher memory or CPU usage.
- **Initial Complexity:** Designing a robust ADT requires more upfront planning and a steeper learning curve for beginners.

Asymptotic Notations:

- Asymptotic Notations are mathematical tools used to analyze the performance of algorithms by understanding how their efficiency changes as the input size grows.
- These notations provide a concise way to express the behaviour of an algorithm's time or space complexity as the input size approaches infinity.
- Rather than comparing algorithms directly, asymptotic analysis focuses on understanding the relative growth rates of algorithms' complexities.
- It enables comparisons of algorithms' efficiency by abstracting away machine-specific constants and implementation details, focusing instead on fundamental trends.
- Asymptotic analysis allows for the comparison of algorithms' space and time complexities by examining their performance characteristics as the input size varies.
- By using asymptotic notations, such as Big O, Big Omega, and Big Theta, we can categorize algorithms based on their worst-case, best-case, or average-case time or space complexities, providing valuable insights into their efficiency.

There are mainly three asymptotic notations:

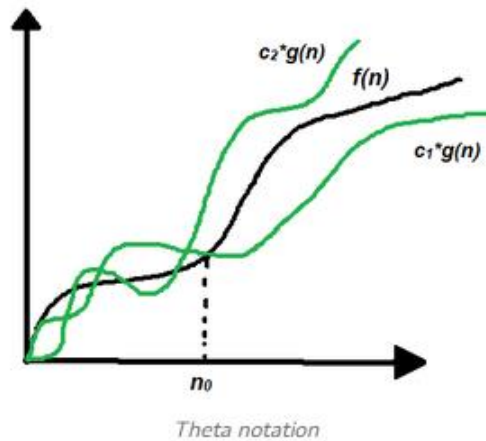
1. **Big-O Notation (O-notation)**
2. **Omega Notation (Ω -notation)**
3. **Theta Notation (Θ -notation)**

1. Theta Notation (Θ -Notation):

Theta notation encloses the function from above and below. Since it represents the upper and the lower bound of the running time of an algorithm, it is used for analyzing the **average-case** complexity of an algorithm.

.Theta (Average Case) You add the running times for each possible input combination and take the average in the average case.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Theta(g)$, if there are constants $c_1, c_2 > 0$ and a natural number n_0 such that $c_1 * g(n) \leq f(n) \leq c_2 * g(n)$ for all $n \geq n_0$



Mathematical Representation of Theta notation:

$\Theta(g(n)) = \{f(n) : \text{there exist positive constants } c_1, c_2 \text{ and } n_0 \text{ such that } 0 \leq c_1 * g(n) \leq f(n) \leq c_2 * g(n) \text{ for all } n \geq n_0\}$

Note: $\Theta(g)$ is a set

The above expression can be described as if $f(n)$ is theta of $g(n)$, then the value $f(n)$ is always between $c_1 * g(n)$ and $c_2 * g(n)$ for large values of n ($n \geq n_0$). The definition of theta also requires that $f(n)$ must be non-negative for values of n greater than n_0 .

The execution time serves as both a lower and upper bound on the algorithm's time complexity.

It exist as both, most, and least boundaries for a given input value.

A simple way to get the Theta notation of an expression is to drop low-order terms and ignore leading constants. For example, Consider the expression $3n^3 + 6n^2 + 6000 = \Theta(n^3)$, the dropping lower order terms is always fine because there will always be a number(n) after which $\Theta(n^3)$ has higher values than $\Theta(n^2)$ irrespective of the constants involved. For a given function $g(n)$, we denote $\Theta(g(n))$ is following set of functions.

Examples :

$\{ 100, \log(2000), 10^4 \}$ belongs to $\Theta(1)$

$\{ (n/4), (2n+3), (n/100 + \log(n)) \}$ belongs to $\Theta(n)$

$\{ (n^2+n), (2n^2), (n^2+\log(n)) \}$ belongs to $\Theta(n^2)$

Note: Θ provides exact bounds.

2. Big-O Notation (O-notation):

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

.It is the most widely used notation for Asymptotic analysis.

.It specifies the upper bound of a function.

.The maximum time required by an algorithm or the worst-case time complexity.

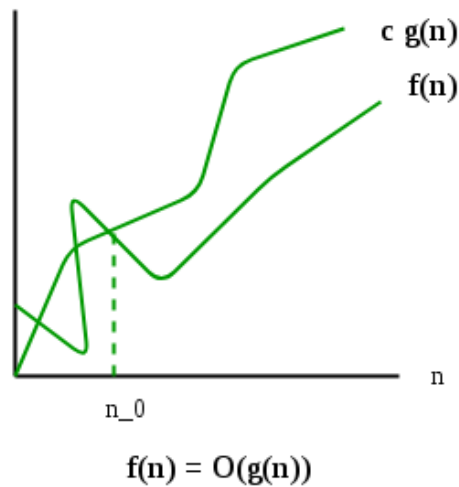
.It returns the highest possible output value(big-O) for a given input.

.Big-O(Worst Case) It is defined as the condition that allows an algorithm to complete statement execution in the longest amount of time possible.

If $f(n)$ describes the running time of an algorithm, $f(n)$ is $O(g(n))$ if there exist a positive constant C and n_0 such that, $0 \leq f(n) \leq cg(n)$ for all $n \geq n_0$

It returns the highest possible output value (big-O) for a given input.

The execution time serves as an upper bound on the algorithm's time complexity.



Mathematical Representation of Big-O Notation:

$O(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq f(n) \leq cg(n) \text{ for all } n \geq n_0 \}$

For example, Consider the case of Insertion Sort. It takes linear time in the best case and quadratic time in the worst case. We can safely say that the time complexity of the Insertion sort is $O(n^2)$.

Note: $O(n^2)$ also covers linear time.

If we use Θ notation to represent the time complexity of Insertion sort, we have to use two statements for best and worst cases:

- The worst-case time complexity of Insertion Sort is $\Theta(n^2)$.
- The best case time complexity of Insertion Sort is $\Theta(n)$.

The Big-O notation is useful when we only have an upper bound on the time complexity of an algorithm. Many times we easily find an upper bound by simply looking at the algorithm.

Examples :

$\{ 100, \log(2000), 10^4 \}$ belongs to **$O(1)$**

$\cup$ $\{ (n/4), (2n+3), (n/100 + \log(n)) \}$ belongs to **$O(n)$**

$\cup$ $\{ (n^2+n), (2n^2), (n^2+\log(n)) \}$ belongs to **$O(n^2)$**

Note: Here, **$\cup$** represents union, we can write it in these manner because **O** provides exact or upper bounds .

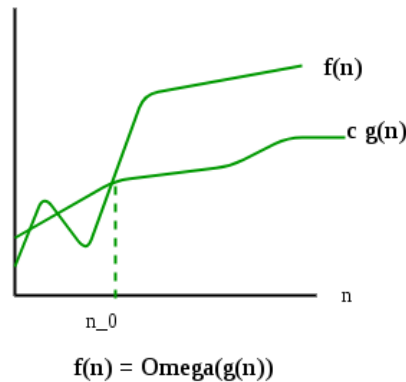
3. Omega Notation (Ω -Notation):

Omega notation represents the lower bound of the running time of an algorithm. Thus, it provides the best case complexity of an algorithm.

The execution time serves as a lower bound on the algorithm's time complexity.

It is defined as the condition that allows an algorithm to complete statement execution in the shortest amount of time.

Let g and f be the function from the set of natural numbers to itself. The function f is said to be $\Omega(g)$, if there is a constant $c > 0$ and a natural number n_0 such that $c \cdot g(n) \leq f(n)$ for all $n \geq n_0$



Mathematical Representation of Omega notation :

$\Omega(g(n)) = \{ f(n) : \text{there exist positive constants } c \text{ and } n_0 \text{ such that } 0 \leq cg(n) \leq f(n) \text{ for all } n \geq n_0 \}$

Let us consider the same Insertion sort example here. The time complexity of Insertion Sort can be written as $\Omega(n)$, but it is not very useful information about insertion sort, as we are generally interested in worst-case and sometimes in the average case.

Examples :

$\{ (n^2+n), (2n^2), (n^2+\log(n)) \}$ belongs to $\Omega(n^2)$

$\cup \{ (n/4), (2n+3), (n/100 + \log(n)) \}$ belongs to $\Omega(n)$

$\cup \{ 100, \log(2000), 10^4 \}$ belongs to $\Omega(1)$

Note: Here, $\cup$ represents union, we can write it in these manner because Ω provides exact or lower bounds.

Properties of Asymptotic Notations:

1. General Properties:

If $f(n)$ is $O(g(n))$ then $a \cdot f(n)$ is also $O(g(n))$, where a is a constant.

Example:

$f(n) = 2n^2+5$ is $O(n^2)$

then, $7 \cdot f(n) = 7(2n^2+5) = 14n^2+35$ is also $O(n^2)$.

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If $f(n)$ is $\Theta(g(n))$ then $a \cdot f(n)$ is also $\Theta(g(n))$, where a is a constant.

If $f(n)$ is $\Omega(g(n))$ then $a \cdot f(n)$ is also $\Omega(g(n))$, where a is a constant.

2. Transitive Properties:

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

Example:

If $f(n) = n$, $g(n) = n^2$ and $h(n) = n^3$

n is $O(n^2)$ and n^2 is $O(n^3)$ then, n is $O(n^3)$

Similarly, this property satisfies both Θ and Ω notation.

We can say,

If $f(n)$ is $\Theta(g(n))$ and $g(n)$ is $\Theta(h(n))$ then $f(n) = \Theta(h(n))$.

If $f(n)$ is $\Omega(g(n))$ and $g(n)$ is $\Omega(h(n))$ then $f(n) = \Omega(h(n))$.

3. Reflexive Properties:

Reflexive properties are always easy to understand after transitive.

If $f(n)$ is given then $f(n)$ is $O(f(n))$. Since MAXIMUM VALUE OF $f(n)$ will be $f(n)$ ITSELF!

Hence $x = f(n)$ and $y = O(f(n))$ tie themselves in reflexive relation always.

Example:

$$f(n) = n^2 ; O(n^2) \text{ i.e } O(f(n))$$

Similarly, this property satisfies both Θ and Ω notation.

We can say that,

If $f(n)$ is given then $f(n)$ is $\Theta(f(n))$.

If $f(n)$ is given then $f(n)$ is $\Omega(f(n))$.

4. Symmetric Properties:

If $f(n)$ is $\Theta(g(n))$ then $g(n)$ is $\Theta(f(n))$.

Example:

$$\text{If } f(n) = n^2 \text{ and } g(n) = n^2$$

$$\text{then, } f(n) = \Theta(n^2) \text{ and } g(n) = \Theta(n^2)$$

This property only satisfies for Θ notation.

5. Transpose Symmetric Properties:

If $f(n)$ is $O(g(n))$ then $g(n)$ is $\Omega(f(n))$.

Example:

$$\text{If } f(n) = n, g(n) = n^2$$

$$\text{then } n \text{ is } O(n^2) \text{ and } n^2 \text{ is } \Omega(n)$$

This property only satisfies O and Ω notations.

6. Some More Properties:

1. If $f(n) = O(g(n))$ and $f(n) = \Omega(g(n))$ then $f(n) = \Theta(g(n))$

2. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then $f(n) + d(n) = O(\max(g(n), e(n)))$

Example:

$$f(n) = n \text{ i.e } O(n)$$

$$d(n) = n^2 \text{ i.e } O(n^2)$$

$$\text{then } f(n) + d(n) = n + n^2 \text{ i.e } O(n^2)$$

3. If $f(n) = O(g(n))$ and $d(n) = O(e(n))$ then $f(n) * d(n) = O(g(n) * e(n))$

Example:

$$f(n) = n \text{ i.e } O(n)$$

$$d(n) = n^2 \text{ i.e } O(n^2)$$

$$\text{then } f(n) * d(n) = n * n^2 = n^3 \text{ i.e } O(n^3)$$

1. Worst Case Analysis (Mostly used)

- In the worst-case analysis, we calculate the upper bound on the running time of an algorithm. We must know the case that causes a maximum number of operations to be executed.
- For [Linear Search](#), the worst case happens when the element to be searched (x) is not present in the array. When x is not present, the `search()` function compares it with all the elements of `arr[]` one by one.
- This is the most commonly used analysis of algorithms (We will be discussing below why). Most of the time we consider the case that causes maximum operations.

2. Best Case Analysis (Very Rarely used)

- In the best-case analysis, we calculate the lower bound on the running time of an algorithm. We must know the case that causes a minimum number of operations to be executed.
- For [linear search](#), the best case occurs when x is present at the first location. The number of operations in the best case is constant (not dependent on n). So the order of growth of time taken in terms of input size is constant.

3. Average Case Analysis (Rarely used)

- In average case analysis, we take all possible inputs and calculate the computing time for all of the inputs. Sum all the calculated values and divide the sum by the total number of inputs.
- Assuming all cases (including x not being in the array) are equally likely in linear search, the average case is computed by summing all cases and dividing by $n+1$, to account for the possibility that the element is absent.

Why is Worst Case Analysis Mostly Used?

Average Case: The average case analysis is not easy to do in most practical cases and it is rarely done. In the average case analysis, we need to consider every input, its frequency and time taken by it which may not be possible in many scenarios

Best Case: Best-case analysis is of limited value, as knowing a lower bound gives little insight when the algorithm's worst-case running time can be extremely large.

Worst Case: This is easier than average case and gives an upper bound which is useful information to analyze software products.

Interesting information about asymptotic notations:

A) For some algorithms, all the cases (worst, best, average) are asymptotically the same. i.e., there are no worst and best cases.

- **Example:** [Merge Sort](#) does order of $n \log(n)$ operations in all cases.

B) Where as most of the other sorting algorithms have worst and best cases.

- **Example 1:** In the typical implementation of [Quick Sort](#) (where pivot is chosen as a corner element), the worst occurs when the input array is already sorted and the best occurs when the pivot elements always divide the array into two halves.
- **Example 2:** For insertion sort, the worst case occurs when the array is reverse sorted and the best case occurs when the array is sorted in the same order as output.

Examples with their complexity analysis:

1. Linear search algorithm:

```
import java.util.Arrays;
```

```
// Linearly search target in arr.
```

```
// If target is present, return the index;
```

```
// otherwise, return -1
```



```

public class GfG {

    public static int search(int[] arr, int x) {

        for (int i = 0; i < arr.length; i++) {

            if (arr[i] == x)

                return i;

        }

        return -1;

    }

    public static void main(String[] args) {

        int[] arr = {1, 10, 30, 15};

        int x = 30;

        System.out.println(search(arr, x));

    }

}

```

Output

```
2
```

Best Case: Constant Time irrespective of input size. This will take place if the element to be searched is on the first index of the given list. So, the number of comparisons, in this case, is 1.

Average Case: Linear Time, This will take place if the element to be searched is at the middle index (in an average search) of the given list.

Worst Case: The element to be searched is not present in the list

2. Special Array Sum:

In this example, we will take an array of length (n) and deals with the following cases :

- If (n) is even then our output will be 0
- If (n) is odd then our output will be the sum of the elements of the array.

```
import java.util.Arrays;
```

```

public class Main {

    public static int getSum(int[] arr1) {

```

```

    int n = arr1.length;

    if (n % 2 == 0) // (n) is even

        return 0;

    int sum = 0;

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

        sum += arr1[i];

    }

    return sum; // (n) is odd

}

public static void main(String[] args) {

    // Declaring two arrays, one with an odd length

    // and the other with an even length

    int[] arr1 = {1, 2, 3, 4};

    int[] arr2 = {1, 2, 3, 4, 5};

    System.out.println(getSum(arr1));

    System.out.println(getSum(arr2));

}

}

```

Output

```

0
15

```

Time Complexity Analysis:

- **Best Case:** The order of growth will be **constant** because in the best case we are assuming that (n) is even.
- **Average Case:** In this case, we will assume that even and odd are equally likely, therefore Order of growth will be **linear**.
- **Worst Case:** The order of growth will be **linear** because in this case, we are assuming that (n) is always odd.

Array:

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

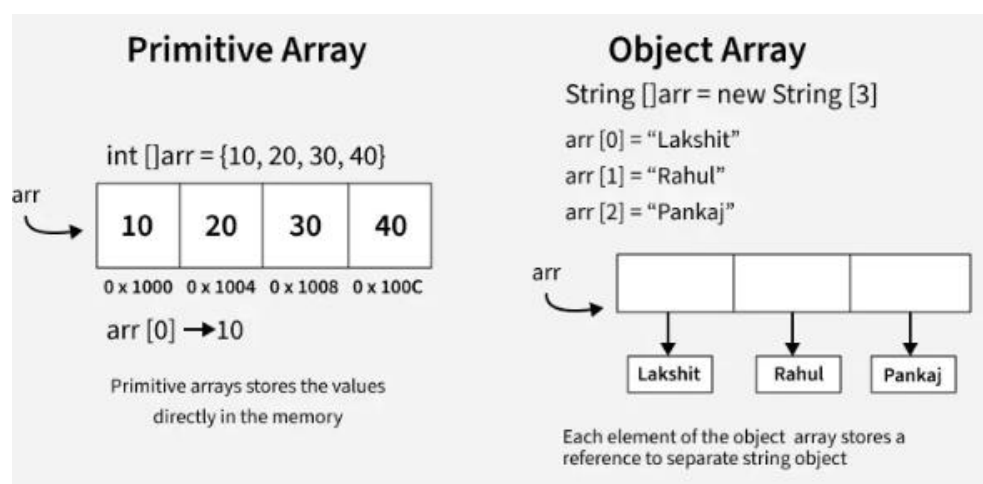
Basic terminologies of Array

- **Array Element:** Elements are items stored in an array.
- **Array Index:** Elements are accessed by their indexes. Indexes in most of the programming languages start from 0.

Memory representation of Array

In an array, all the elements or their references are stored in contiguous memory locations.

This allows for efficient access and manipulation of elements.



Declaration of Array

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

// This array will store integer type element

```
int arr[];
```

// This array will store char type element

```
char arr[];
```

// This array will store float type element

```
float arr[];
```

Initialization of Array

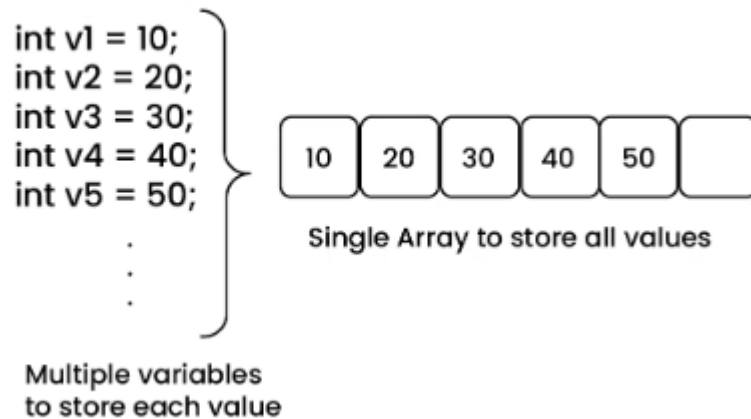
Arrays can be initialized in different ways in different languages. Below are some language-specific array initialization:

```
int arr[] = { 1, 2, 3, 4, 5 };
```

```
char arr[] = { 'a', 'b', 'c', 'd', 'e' };
```

```
float arr[] = { 1.4f, 2.0f, 24f, 5.0f, 0.0f };
```

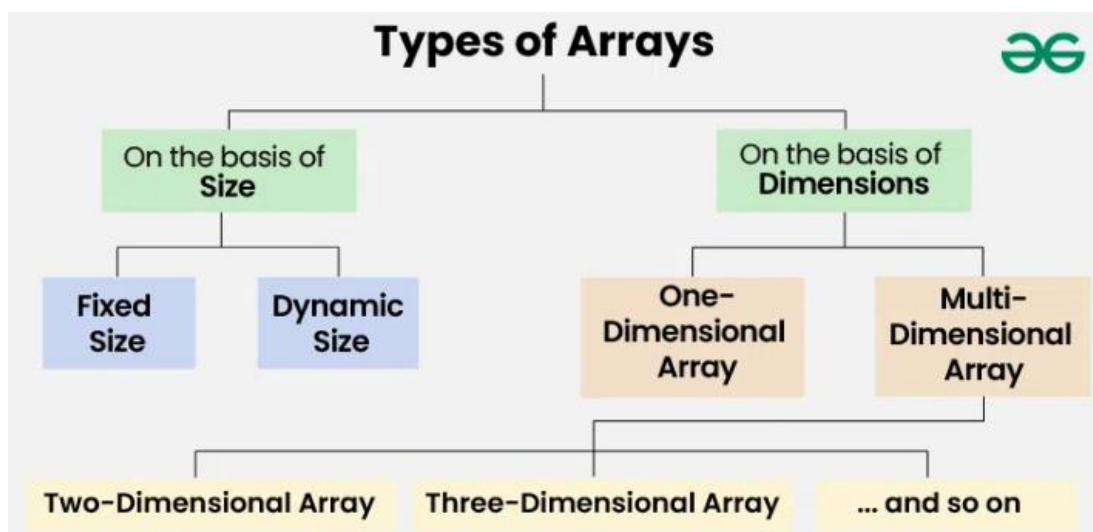
Importance of Array



Types of Arrays

Arrays can be classified in two ways:

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



Types of Arrays on the basis of Size

1. Fixed Sized Arrays

- We cannot alter or update the size of this array. Here only a fixed size (i.e. the size that is mentioned in square brackets []) of memory will be allocated for storage.
- In case, we don't know the size of the array then if we declare a larger size and store a lesser number of elements, it will result in a wastage of memory. And if we declare a lesser size than the number of elements then we won't get enough memory to store all the elements.

```
# Create a fixed-size list of length 5,
```

```
# initialized with zeros
```

```
arr = [0] * 5
```

```
# Output the fixed-size list
```

```
print(arr)
```

2. Dynamic Sized Arrays

The size of the array changes as per user requirements during execution of code so the coders do not have to worry about sizes. They can add and removed the elements as per the need. The memory is mostly dynamically allocated and de-allocated in these arrays.

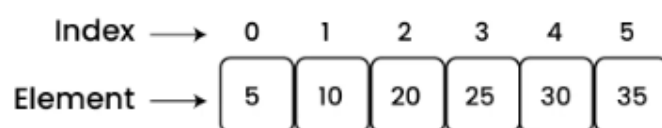
```
// Dynamic Integer Array
```

```
ArrayList<Integer> arr = new ArrayList<>();
```

Types of Arrays on the basis of Dimensions

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

One-Dimensional Array (1-D Array)



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

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

Two-Dimensional Array (2-D Array or Matrix)

		Columns →		
		0	1	2
Rows ↓	0	a_{00}	a_{01}	a_{02}
	1	a_{10}	a_{11}	a_{12}
	2	a_{20}	a_{21}	a_{22}

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

Three-Dimensional Array (3-D Array)

		Columns			
		Column 1	Column 2	Column 3	
Rows	Row 1	a_{000}	a_{001}	a_{002}	→ Matrix 1
	Row 2	a_{010}	a_{100}	a_{101}	→ Matrix 2
	Row 3	a_{020}	a_{110}	a_{200}	→ Matrix 3
			a_{120}	a_{201}	
			a_{210}	a_{202}	
				a_{211}	
				a_{212}	
				a_{220}	
				a_{221}	
				a_{222}	

Matrices

Operations on Array

- Traversal in Array
- **Insertions** : At Beginning, At given position and At the end.
- **Deletion** : From Beginning, Given Position, First Occurrence, All occurrences and From End
- **Searching** : Linear Search and Binary Search

Traversal in Array:

Traversal in an array refers to the process of accessing each element in the array sequentially, typically to perform a specific operation, such as searching, sorting, or modifying the elements.

Traversing an array is essential for a wide range of tasks in programming, and the most common methods of traversal include iterating through the array using loops like for, while, or foreach. Efficient traversal plays a critical role in algorithms that require scanning or manipulating data.

Examples:

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

Output: "10 20 30 40 50 "

Explanation: Just traverse and print the numbers.

Input: arr[] = [7, 8, 9, 1, 2]

Output: "7 8 9 1 2 "

Explanation: Just traverse and print the numbers.

Input: arr[] = [100, 200, 300, 400, 500]

Output: "100 200 300 400 500 "

Explanation: Just traverse and print the numbers.

Types of Array Traversal:

There are mainly two types of array traversal:

1. Linear Traversal

Linear traversal is the process of visiting each element of an array sequentially, starting from the first element and moving to the last element. During this traversal, each element is processed (printed, modified, or checked) one after the other, in the order they are stored in the array. This is the most common and straightforward way of accessing the elements of an array.

```
public class Main {  
    public static void main(String[] args) {  
        int[] arr = {1, 2, 3, 4, 5};  
        int n = arr.length;  
  
        System.out.print("Linear Traversal: ");  
        for(int i = 0; i < n; i++) {  
            System.out.print(arr[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

```
}
```

2. Reverse Traversal:

Reverse traversal is the process of visiting each element of an array starting from the last element and moving towards the first element. This method is useful when you need to process the elements of an array in reverse order. In this type of traversal, you begin from the last index (the rightmost element) and work your way to the first index (the leftmost element).

```
class Main {  
    public static void main(String[] args) {  
        int[] arr = {1, 2, 3, 4, 5};  
        int n = arr.length;  
  
        System.out.print("Reverse Traversal: ");  
        for (int i = n - 1; i >= 0; i--) {  
            System.out.print(arr[i] + " ");  
        }  
        System.out.println();  
    }  
}
```

Insertions:

1. Insert Element at the Beginning of an Array
2. Insert Element at a given position in an Array
3. Insert Element at the End of an Array

Insert Element at the Beginning of an Array:

Inserting an element at the beginning of an array involves shifting all existing elements one position to the right to create an empty space for the new element at index 0.

Examples:

Input: arr[] = [10, 20, 30, 40], ele = 50

Output: [50, 10, 20, 30, 40]

Input: arr[] = [], ele = 20

Output: [20]

Insert Element at a given position in an Array

Inserting an element at a given position in an array involves shifting the elements from the specified position onward one index to the right to make an empty space for the new element. This operation ensures that the order of the existing elements is preserved while placing the new element at the desired index. After shifting the elements, the new element is inserted at the target position.

Examples:

Input: arr[] = [10, 20, 30, 40], pos = 2, ele = 50

Output: [10, 50, 20, 30, 40]

Input: arr[] = [], pos = 1, ele = 20

Output: [20]

Input: arr[] = [10, 20, 30, 40], pos = 5, ele = 50

Output: [10, 20, 30, 40, 50]

Insert Element at the End of an Array:

Inserting an element at the end of an array involves adding the new element to the last available index of the array. Inserting an element at the end of the array takes constant time provided there is enough space in the array to add the new element.

Examples:

Input: arr[] = [10, 20, 30, 40], ele = 50

Output: [10, 20, 30, 40, 50]

Input: arr[] = [], ele = 20

Output: [20]

Searching:

1. Searching in an Unsorted Array using Linear Search
2. Searching in a Sorted Array using Linear Search

Searching operations in an Unsorted Array using Linear Search:

In an unsorted array, the search operation can be performed by linear traversal from the first element to the last element, i.e. Linear Search



```
class Main {
```

```

// Function to implement

// search operation

static int findElement(int arr[], int n, int key)

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

        if (arr[i] == key)

            return i;

    // If the key is not found

    return -1;
}

// Driver's Code

public static void main(String args[])

{

    int arr[] = { 12, 34, 10, 6, 40 };

    int n = arr.length;

    // Using a last element as search element

    int key = 40;

    // Function call

    int position = findElement(arr, n, key);

    if (position == -1)

        System.out.println("Element not found");

    else

        System.out.println("Element Found at Position: "

            + (position + 1));

}

```

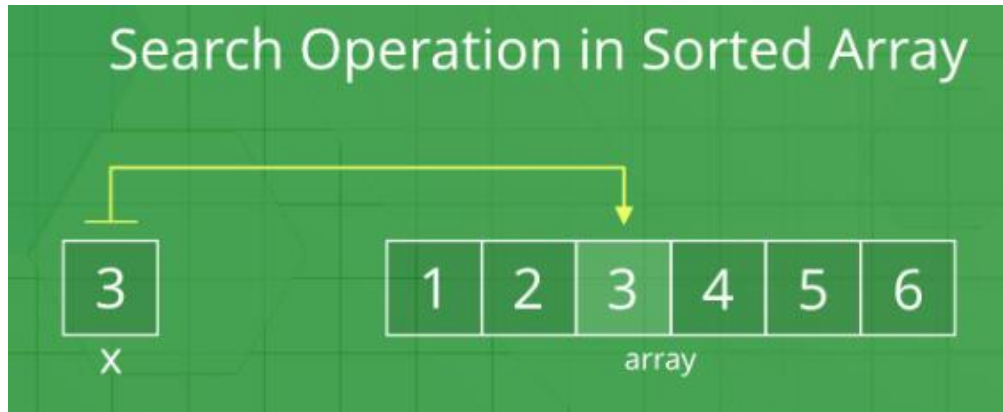
```
}
```

Output

Element Found at Position: 5

Searching in a Sorted Array using Linear Search:

In a sorted array, the most trivial method for search operation is by using Linear Search.



```
// Java program to implement linear  
// search in sorted arrays
```

```
class Main {  
  
    // Function to implement  
    // search operation  
    static int findElement(int arr[], int n, int key)  
    {  
        for (int i = 0; i < n; i++)  
            if (arr[i] == key)  
                return i;  
  
        // If the key is not found  
        return -1;  
    }  
  
    // Driver's Code  
    public static void main(String args[])  
    {  
        int arr[] = { 5, 6, 7, 8, 9, 10 };  
        int n = arr.length;  
  
        // Using a last element as search element
```

```

int key = 10;

// Function call
int position = findElement(arr, n, key);

if (position == -1)
    System.out.println("Element not found");
else
    System.out.println("Element Found at Position: "
        + (position + 1));
}
}

```

Output

```
Element Found at Position: 6
```

Deletion:

- Delete an Element from the **Beginning** of an Array
- Delete an Element from a **Given Position** in an Array
- Delete **First Occurrence** of Given Element from an Array
- Remove **All Occurrences** of an Element in an Array
- Delete an Element from the **End** of an array

Delete an Element from the Beginning of an Array:

Given an array of integers, the task is to delete an element from the **beginning** of the array.

Examples:

Input: arr[] = [10, 20, 30, 40]

Output: [20, 30, 40]

Input: arr[] = [20]

Output: []

To delete an element from the beginning of an array, we need to shift all subsequent elements one position to the left to fill the gap created by the removal. After shifting all the elements, **reduce** the array size by **1** to remove the extra element at the end. **To know more about the implementation, please refer Delete an Element from the Beginning of an Array.**

Delete an Element from a Given Position in an Array:

Given an array of integers, the task is to delete an element from a **given position** in the array.

Examples:

Input: arr[] = [10, 20, 30, 40], pos = 1

Output: [20, 30, 40]

Input: arr[] = [10, 20, 30, 40], pos = 2

Output: [10, 30, 40]

Input: arr[] = [10, 20, 30, 40], pos = 4

Output: [10, 20, 30]

To delete an element from a given position in an array, all elements occurring after the given position need to be shifted one position to the left. After shifting all the elements, **reduce** the array size by **1** to remove the extra element at the end. To know more about the implementation, please refer [Delete an Element from a Given Position in an Array](#).

Delete First Occurrence of Given Element from an Array

Given an array of integers, the task is to delete a given element from the array. If there are multiple occurrences of the element, we need to remove only its first occurrence.

Examples:

Input: arr[] = [10, 20, 30, 40], ele = 20

Output: [10, 30, 40]

Input: arr[] = [10, 20, 30, 40], ele = 25

Output: [10, 20, 30, 40]

Input: arr[] = [10, 20, 20, 20, 30], ele = 20

Output: [10, 20, 20, 30]

To delete the first occurrence of a given element from an array, first iterate through the array to find the first occurrence of the element and as soon as we find it, we need to shift all subsequent elements one position to the left. After shifting the elements, the reduce the array size by one to remove the extra element at the end. To know more about the implementation, please refer [Delete First Occurrence of Given Element from an Array](#).

Remove All Occurrences of an Element in an Array

Given an array of integers and an integer ele, the task is to remove all occurrences of ele in-place and return the count of elements not equal to ele. The modified array should first have those elements which are not equal to ele, and the remaining elements can be ignored.

Examples:

Input: arr[] = [3, 2, 2, 3], ele = 3

Output: 2

Explanation: The answer is 2 because there are 2 elements which are not equal to 3 and modified arr[] = [2, 2, _, _]

Input: arr[] = [0, 1, 3, 0, 2, 2, 4, 2], ele = 2

Output: 5

Explanation: The answer is 5 because there are 5 elements which are not equal to 2 and modified arr[] = [0, 1, 3, 0, 4, _, _, _].

The approach is to iterate through the array while maintaining a subarray at the beginning containing only elements that are not equal to ele. We can use a counter, k, to track the end of this subarray. Each time we encounter an element not equal to ele, we place it at the k-th index and then increment k. To know more about the implementation, please refer [Remove All Occurrences of an Element in an Array](#).

Delete an Element from the End of an Array

Given an array of integers, the task is to delete an element from the **end** of the array.

Examples:

Input: arr[] = [10, 20, 30, 40]

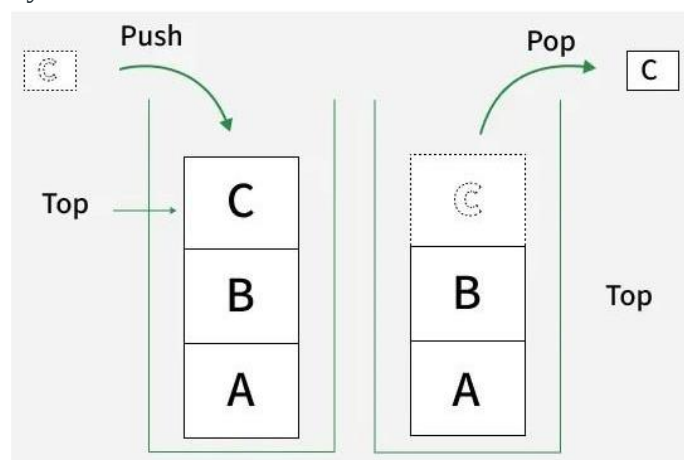
Output: [10, 20, 30]

Input: arr[] = [20]

Output: []

Stack:

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



LIFO (Last In First Out) Principle

The LIFO principle means that the last element added to a stack is the first one to be removed.

- New elements are always pushed on top.
- Removal (pop) also happens only from the top.
- This ensures a strict order: last in → first out.

Real-world examples of LIFO:

- **Stack of plates** – The last plate placed on top is the first one you pick up.
- **Shuttlecock box** – The last shuttlecock inserted is the first one taken out, since both operations happen from the same end.

Basic Terminologies of Stack

- **Top:** The position of the most recently inserted element. Insertions (push) and deletions (pop) are always performed at the top.
- **Size:** Refers to the current number of elements present in the stack.

Types of Stack:

Fixed Size Stack

- A fixed size stack has a predefined capacity.

- Once it becomes full, no more elements can be added (this causes **overflow**).
- If the stack is empty and we try to remove an element, it causes **underflow**.
- Typically implemented using a static array.

Example: Declaring a stack of size 10 using an array.

Dynamic Size Stack

- A dynamic size stack can grow and shrink automatically as needed.
- If the stack is full, its capacity expands to allow more elements.
- As elements are removed, memory usage can shrink as well.
- Can be implemented using:
 - > **Linked List** → grows/shrinks naturally.
 - > **Dynamic Array** (like vector in C++ or ArrayList in Java) → resizes automatically.

Example: Stack implementation using linked list or resizable array.

Common Operations on Stack:

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

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

Stack using Array:

A stack is a linear data structure that follows the Last-In-First-Out (LIFO) principle. It can be implemented using an array by treating the end of the array as the top of the stack.

A stack can be implemented using an array where we maintain:

- An integer array to store elements.
- A variable capacity to represent the maximum size of the stack.
- A variable top to track the index of the top element. Initially, top = -1 to indicate an empty stack.

```
class myStack {

    // array to store elements
    private int[] arr;

    // maximum size of stack
    private int capacity;

    // index of top element
    private int top;

    // constructor
    public myStack(int cap) {
        capacity = cap;
        arr = new int[capacity];
        top = -1;
    }
}
```

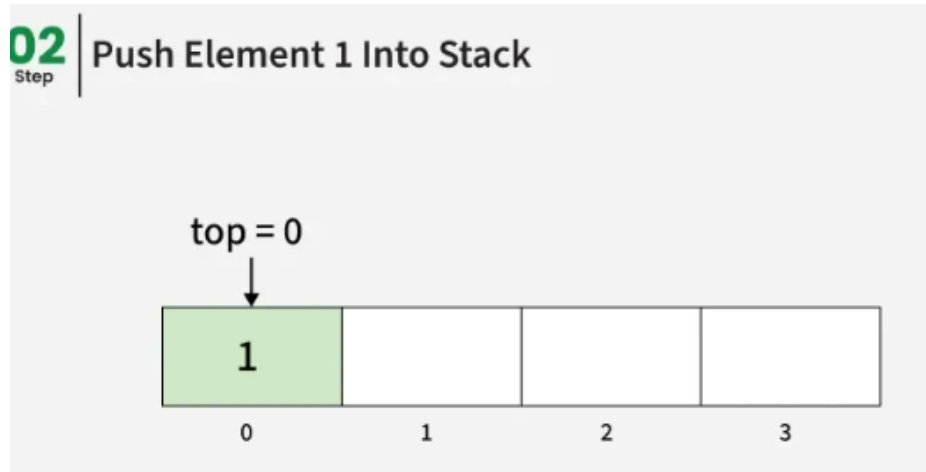
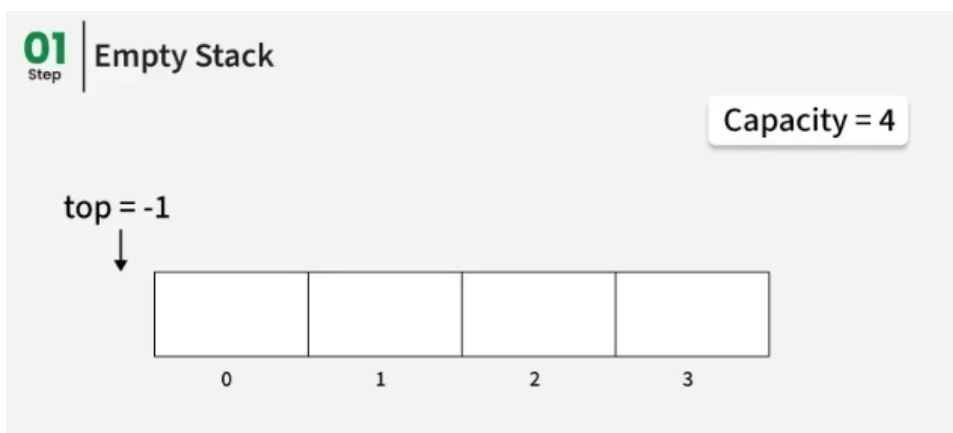
}

Operations On Stack

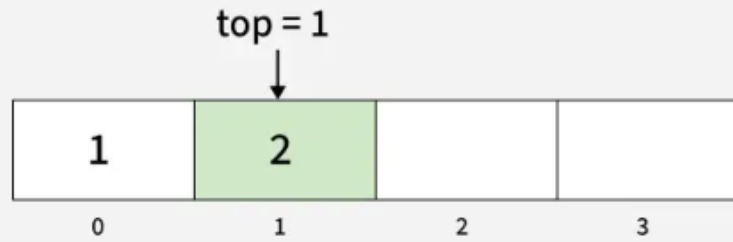
Push Operation:

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

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

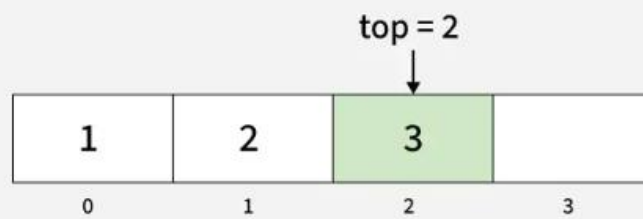


03 Step | Push Element 2 Into Stack

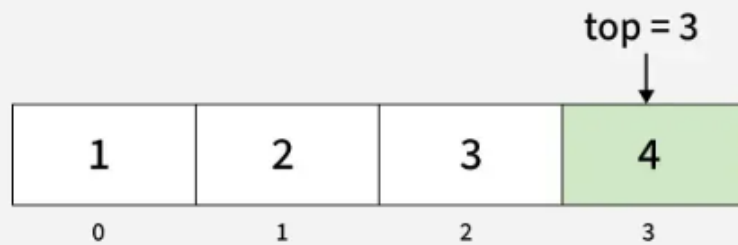


04 Step | Push Element 3 Into Stack

Capacity = 4

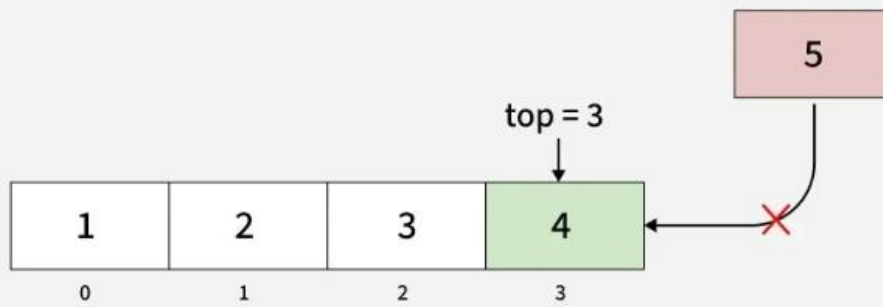


05 Step | Push Element 4 Into Stack



06
Step

Push Element 5 Into Stack (Stack Overflow)



```
void push(int x) {  
    if (top == capacity - 1) {  
        System.out.println("Stack Overflow");  
        return;  
    }  
    arr[++top] = x;  
}
```

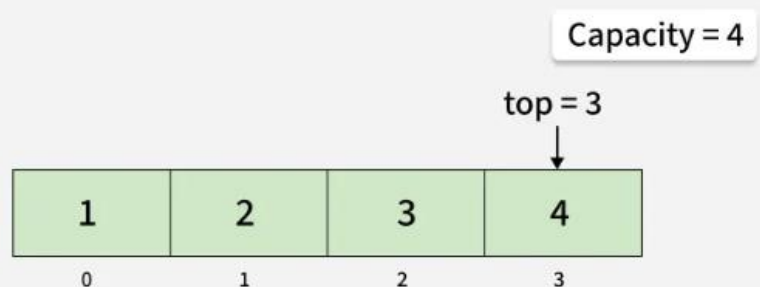
Pop Operation:

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

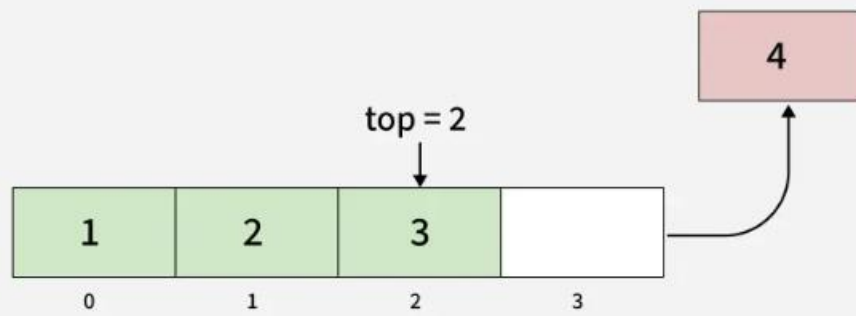
- Before popping the element from the stack, we check if the stack is empty .
- If the stack is empty ($\text{top} == -1$), then Stack Underflows and we cannot remove any element from the stack.
- Otherwise, we store the value at top, decrement the value of top by 1 ($\text{top} = \text{top} - 1$) and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

01
Step

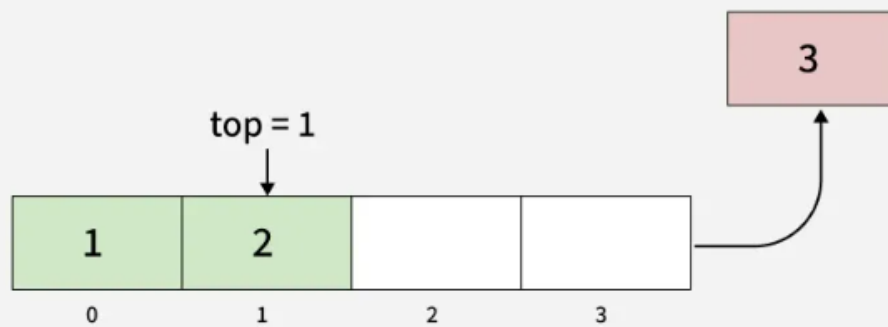
Initial Stack



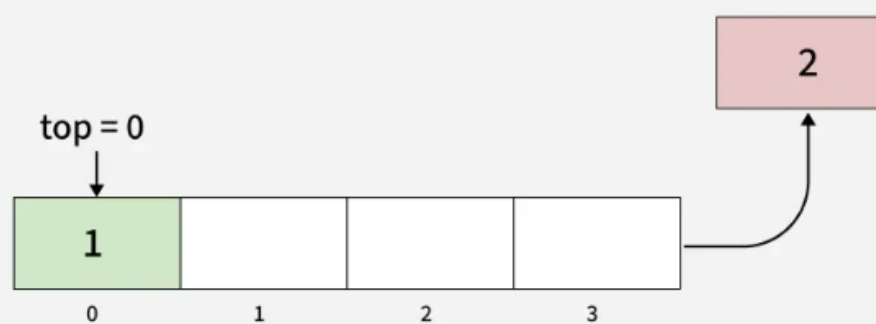
02 Step Pop Element 4 From Stack



03 Step Pop Element 3 From Stack

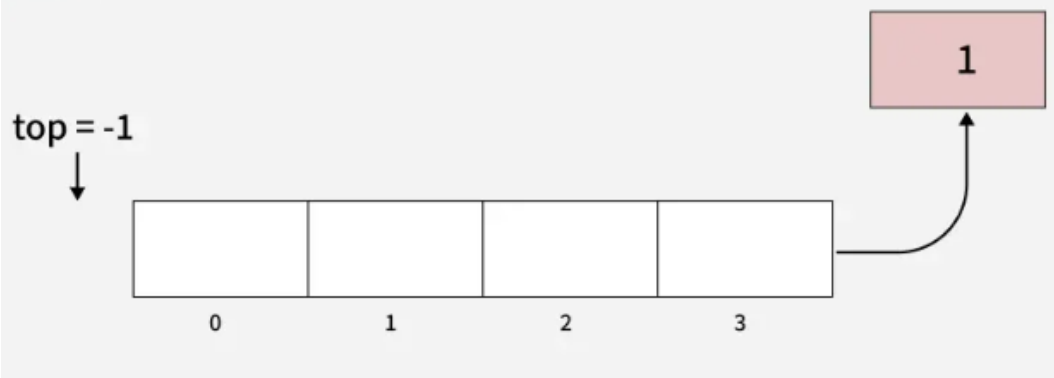


04 Step Pop Element 2 From Stack



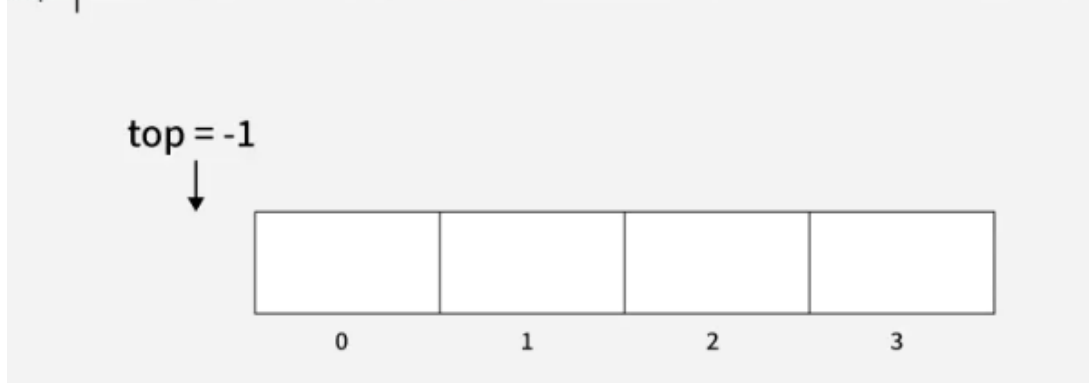
05
Step

Pop Element 1 From Stack



06
Step

Empty Stack (Popping an Element will cause underflow)

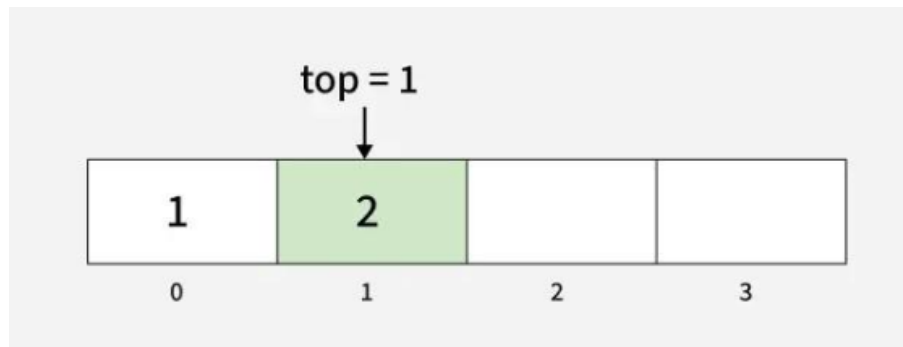


```
int pop() {  
    if (top == -1) {  
        System.out.println("Stack Underflow");  
        return -1;  
    }  
    return arr[top--];  
}
```

Top or Peek Operation in Stack:

Returns the top element of the stack.

- Before returning the top element from the stack, we check if the stack is empty.
- If the stack is empty ($\text{top} == -1$), we simply print “Stack is empty”.
- Otherwise, we return the element stored at **index = top**.



```
int peek() {
    if (top == -1) {
        System.out.println("Stack is Empty");
        return -1;
    }

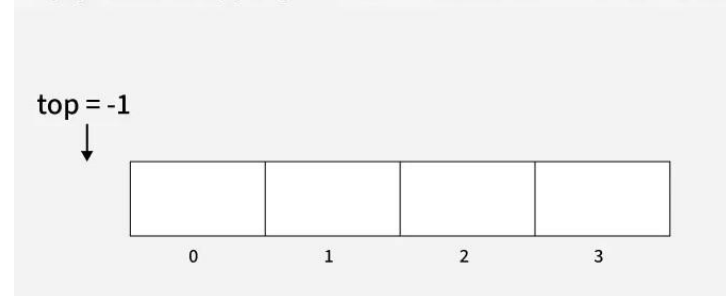
    return arr[top];
}
```

isEmpty Operation in Stack:

Returns true if the stack is empty, else false.

- Check for the value of top in stack.
- If (top == -1) , then the stack is empty so return true.
- Otherwise, the stack is not empty so return false.

Empty Stack (Popping an Element will cause underflow)



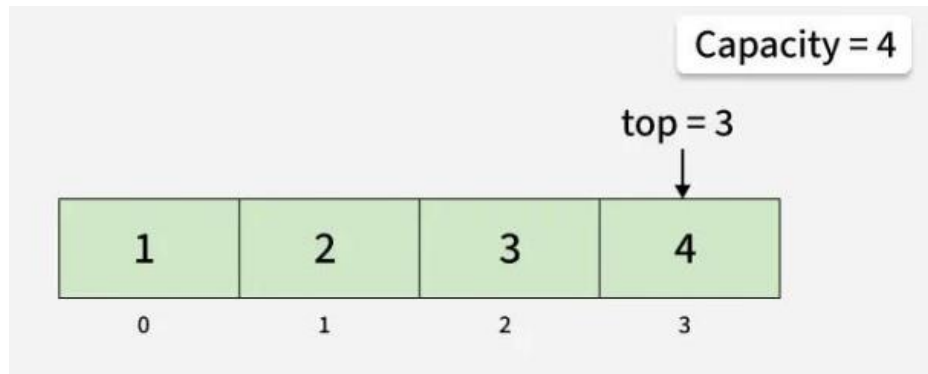
```
boolean isEmpty() {
    return top == -1;
}
```

isFull Operation in Stack :

Returns true if the stack is full, else false.

- Check for the value of top in stack.
- If (top == capacity-1), then the stack is full so return true.

- Otherwise, the stack is not full so return false.



```
import java.util.Arrays;
class myStack {
    // array to store elements
    private int[] arr;
    // maximum size of stack
    private int capacity;
    // index of top element
    private int top;
    // constructor
    public myStack(int cap) {
        capacity = cap;
        arr = new int[capacity];
        top = -1;
    }
    // push operation
    public void push(int x) {
        if (top == capacity - 1) {
            System.out.println("Stack Overflow");
            return;
        }
        arr[++top] = x;
    }
    // pop operation
    public int pop() {
        if (top == -1) {
            System.out.println("Stack Underflow");
            return -1;
        }
        return arr[top--];
    }
    // peek (or top) operation
    public int peek() {
```

```

        if (top == -1) {
            System.out.println("Stack is Empty");
            return -1;
        }
        return arr[top];
    }
    // check if stack is empty
    public boolean isEmpty() {
        return top == -1;
    }
    // check if stack is full
    public boolean isFull() {
        return top == capacity - 1;
    }
}
public class Main {
    public static void main(String[] args) {
        myStack st = new myStack(4);
        // pushing elements
        st.push(1);
        st.push(2);
        st.push(3);
        st.push(4);
        // popping one element
        System.out.println("Popped: " + st.pop());
        // checking top element
        System.out.println("Top element: " + st.peek());
        // checking if stack is empty
        System.out.println("Is stack empty: " +
            (st.isEmpty() ? "Yes" : "No"));
        // checking if stack is full
        System.out.println("Is stack full: " +
            (st.isFull() ? "Yes" : "No"));
    }
}

```

Output

```

Popped: 4
Top element: 3
Is stack empty: No
Is stack full: No

```

Stack Implementation using Dynamic Array

When using a fixed-size array, the stack has a maximum capacity that cannot grow beyond its initial size. To overcome this limitation, we can use **dynamic arrays**. Dynamic arrays automatically resize themselves as elements are added or removed, which makes the stack more flexible.

- In C++, we can use vector.
- In Java, we can use Array List.
- In Python, we can use list.
- In C#, we can use List.
- In JavaScript, we can use Array.

```
import java.util.ArrayList;
class myStack {
    ArrayList<Integer> arr = new ArrayList<>();
    // push operation
    void push(int x) {
        arr.add(x);
    }
    // pop operation
    int pop() {
        if (arr.isEmpty()) {
            System.out.println("Stack Underflow");
            return -1;
        }
        int val = arr.get(arr.size() - 1);
        arr.remove(arr.size() - 1);
        return val;
    }

    // peek operation
    int peek() {
        if (arr.isEmpty()) {
            System.out.println("Stack is Empty");
            return -1;
        }
        return arr.get(arr.size() - 1);
    }

    // check if stack is empty
    boolean isEmpty() {
        return arr.isEmpty();
    }

    // current size
    int size() {
        return arr.size();
    }

    public static void main(String[] args) {
```



```

        myStack st = new myStack();
        // pushing elements
        st.push(1);
        st.push(2);
        st.push(3);
        st.push(4);
        // popping one element
        System.out.println("Popped: " + st.pop());
        // checking top element
        System.out.println("Top element: " + st.peek());
        // checking if stack is empty
        System.out.println("Is stack empty: " +
            (st.isEmpty() ? "Yes" : "No"));
        // checking current size
        System.out.println("Current size: " + st.size());
    }
}

```

Output

```

Popped: 4
Top element: 3
Is stack empty: No
Current size: 3

```

Stack - Linked List Implementation:

A stack is a linear data structure that follows the **Last-In-First-Out (LIFO)** principle. It can be implemented using a [linked list](#), where each element of the stack is represented as a node. The head of the linked list acts as the top of the stack.

Declaration of Stack using Linked List

A stack can be implemented using a **linked list** where we maintain:

- A Node structure/class that contains:
 - data → to store the element.
 - next → pointer/reference to the next node in the stack.
- A pointer/reference top that always points to the current **top node** of the stack. Initially, top = null to represent an empty stack.

```

/* Node structure */
class Node {
    public int data;
    public Node next;

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

/* Stack class */

```

```

class MyStack {

    // pointer to top node
    private Node top;

    public MyStack() {
        // initially stack is empty
        top = null;
    }
}

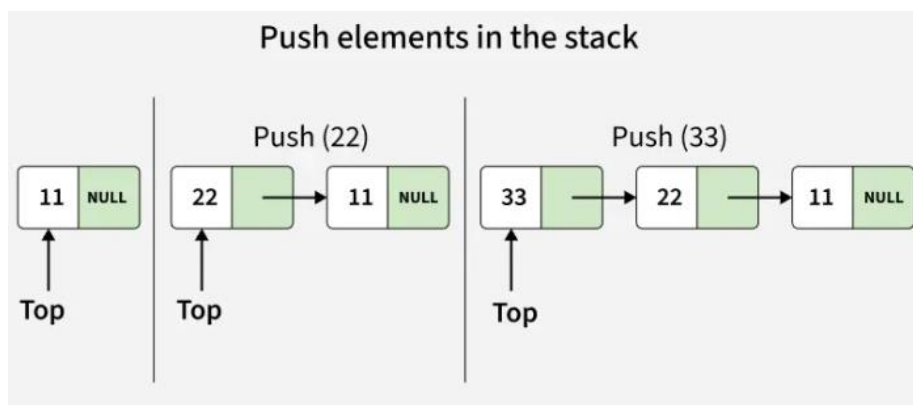
```

Operations on Stack using Linked List

Push Operation

Adds an item to the stack. Unlike array implementation, there is no fixed capacity in linked list. Overflow occurs only when memory is exhausted.

- A new node is created with the given value.
- The new node's next pointer is set to the current top.
- The top pointer is updated to point to this new node.



```

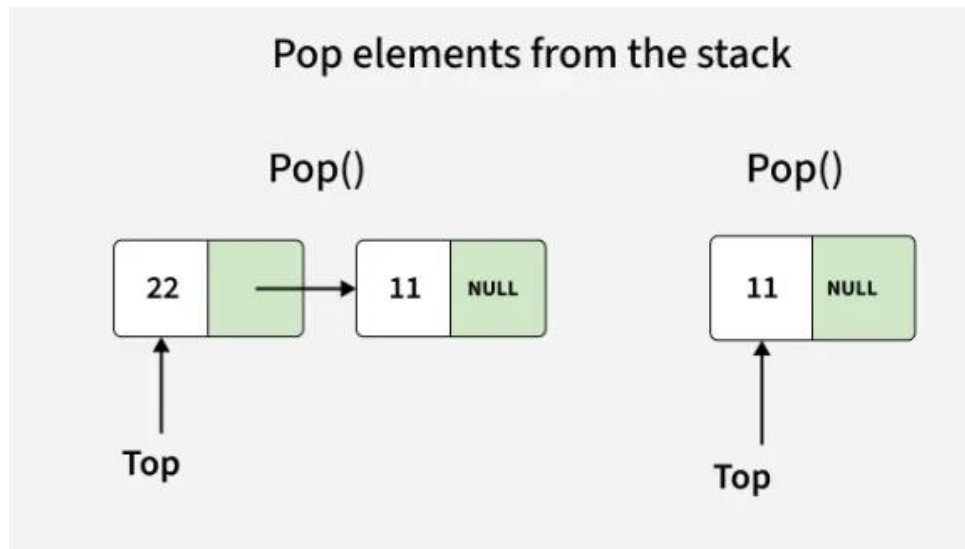
void push(int x) {
    Node temp = new Node(x);
    temp.next = top;
    top = temp;
}

```

Pop Operation

Removes the top item from the stack. If the stack is empty, it is said to be an **Underflow condition**.

- Before deleting, we check if the stack is empty (`top == NULL`).
- If the stack is empty, underflow occurs and deletion is not possible.
- Otherwise, we store the current top node in a temporary pointer.
- Move the top pointer to the next node.
- Delete the temporary node to free memory.



```
public int pop() {  
    if (top == null) {  
        System.out.println("Stack Underflow");  
        return -1;  
    }  
  
    Node temp = top;  
    top = top.next;  
    int val = temp.data;  
  
    temp = null;  
    return val;  
}
```

Peek (or Top) Operation:

Returns the value of the top item without removing it from the stack.

- If the stack is empty (top == NULL), then no element exists.
- Otherwise, simply return the data of the node pointed by top.

```
int peek() {  
    if (top == null) {  
        System.out.println("Stack is Empty");  
        return -1;  
    }  
  
    return top.data;  
}
```

isEmpty Operation

Checks whether the stack has no elements.

- If the top pointer is NULL, it means the stack is empty and the function returns true.
- Otherwise, it returns false.

```
boolean isEmpty() {  
    return top == null;  
}
```

Stack Implementation using Linked List:

```
class Node {  
    int data;  
    Node next;  
    Node(int x) {  
        data = x;  
        next = null;  
    }  
}  
  
// Stack implementation using linked list  
class myStack {  
    Node top;  
    // To Store current size of stack  
    int count;  
    myStack() {  
        // initially stack is empty  
        top = null;  
        count = 0;  
    }  
    // push operation  
    void push(int x) {  
        Node temp = new Node(x);  
        temp.next = top;
```

```

        top = temp;
        count++;
    }
    // pop operation
    int pop() {
        if (top == null) {
            System.out.println("Stack Underflow");
            return -1;
        }
        Node temp = top;
        top = top.next;
        int val = temp.data;

        count--;
        return val;
    }
    // peek operation
    int peek() {
        if (top == null) {
            System.out.println("Stack is Empty");
            return -1;
        }
        return top.data;
    }
    // check if stack is empty
    boolean isEmpty() {
        return top == null;
    }
    // size of stack
    int size() {
        return count;
    }

```

```

}

public static void main(String[] args) {
    myStack st = new myStack();

    // pushing elements
    st.push(1);
    st.push(2);
    st.push(3);
    st.push(4);

    // popping one element
    System.out.println("Popped: " + st.pop());

    // checking top element
    System.out.println("Top element: " + st.peek());

    // checking if stack is empty
    System.out.println("Is stack empty: " + (st.isEmpty() ? "Yes" : "No"));

    // checking current size
    System.out.println("Current size: " + st.size());
}
}

```

Output

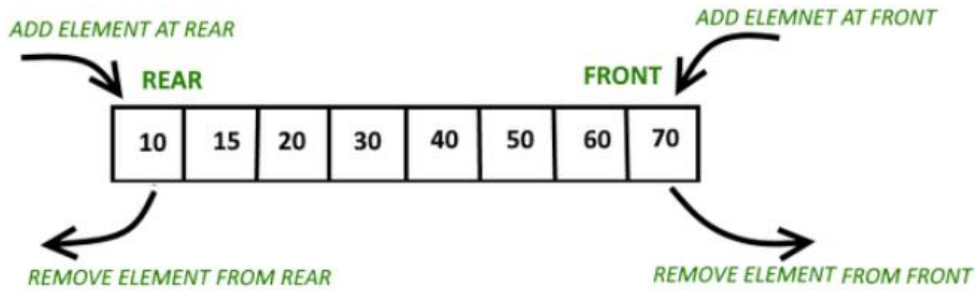
```

Popped: 4
Top element: 3
Is stack empty: No
Current size: 3

```

Stack Implementation using Deque:

A doubly ended queue or deque allows insertion and deletion at both ends. In a stack, we need to do insertions and deletions at one end only. We can use either end of deque (front or back) to implement a stack,



In the below implementation, we use back (or rear) of stack to do both insertions and deletions. Please note that the time complexities of all the operations (insertions and deletions at both ends) in a deque is $O(1)$. So the time complexity of the below implementation is $O(1)$ only.

- In Java, the deque implementation of stack is preferred over standard Stack class, because the standard Stack class is legacy class and mainly designed for multi threaded environment.
- In Python, there is no standard stack class, so we can either use list or deque. The deque implementation is preferred because it is optimized for insertion and deletion at ends.

```
import java.util.*;
class GfG {
    public static void main(String[] args) {
        Deque<Integer> stack = new ArrayDeque<>();

        stack.push(10);
        stack.push(20);
        stack.push(30);

        System.out.println(stack.pop() + " popped from stack");
        System.out.println("Top element is: " + stack.peek());
    }
}
```

Output

```
30 popped from deque
Top element is: 20
```

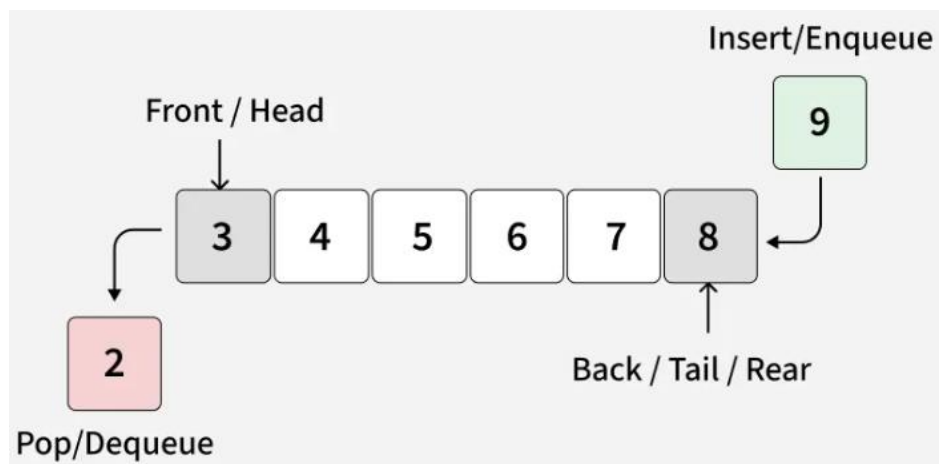
Queue :

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

- It is an ordered list in which insertions are done at one end which is known as the rear and deletions are done from the other end known as the front.
- A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first.
- The difference between stack and queue is in removing an element. In a stack we remove the item that is most recently added while in a queue, we remove the item that is least recently added.

FIFO Principle in Queue:

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

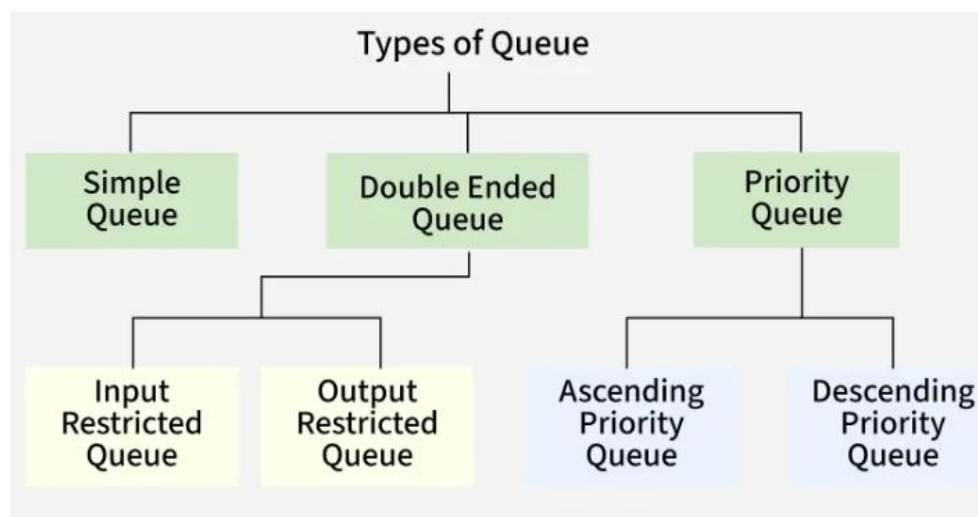


Basic Terminologies of Queue

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

Types of Queues

Queue data structure can be classified into 3 types:



1. Simple Queue:

A simple queue follows the FIFO (First In, First Out) principle.

- Insertion is allowed only at the rear (back).

- Deletion is allowed only from the front.
- Can be implemented using a linked list or a circular array.

When an array is used, we often prefer a **circular queue**, which is mainly an efficient array implementation of a simple queue. It efficiently utilizes memory by reusing the empty spaces left after deletion, avoiding wastage that occurs in a normal linear array implementation..

2. Double-Ended Queue (Deque)

In a deque, insertion and deletion can be performed from both ends.

3. Priority Queue

A queue where each element is assigned a **priority**, and deletion always happens based on priority (not just position).

Queue Operations

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

Implementation of Queue

Queue can be implemented using following data structures:

- Simple Array implementation of Queue
- Implementation of Queue using Linked List.

Queue using Array - Simple Implementation:

A queue is a linear data structure that follows the **FIFO (First In, First Out)** principle. The first element inserted is the first one to be removed.

Declaration of Queue Using Array:

A queue can be implemented using an array, and there are two main ways:

1. Infinite (or Dynamically Growable) Array Queue
2. Fixed-Size Array Queue

Infinite (or Dynamically Growable) Array Queue:

We can implement a queue using a conceptually infinite array by maintaining only a **front pointer**. The front pointer tracks the first valid element.

- **Enqueue:** Insert at the next available position at the end. No rear pointer is required.
- **Dequeue:** Remove the element at front and increment the front pointer.

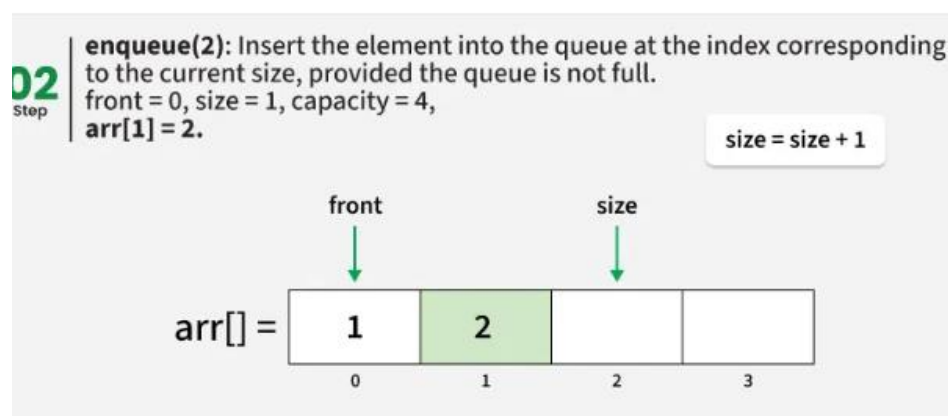
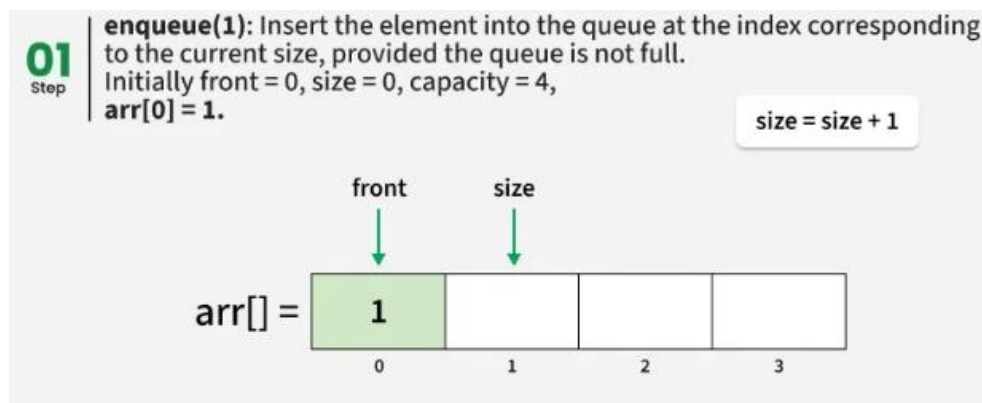
The space before front is never reused, and unlike basic array implementations, we do not shift elements after each dequeue. This ensures both enqueue and dequeue operations run in $O(1)$ time with a simple design.

Limitations:

- **Wasted space:** The elements before the front pointer are never reused, so memory can be wasted if many elements are dequeued.
- **Infinite array assumption:** We assume the array is conceptually infinite. In practice, memory is finite, so very large queues can cause memory issues.

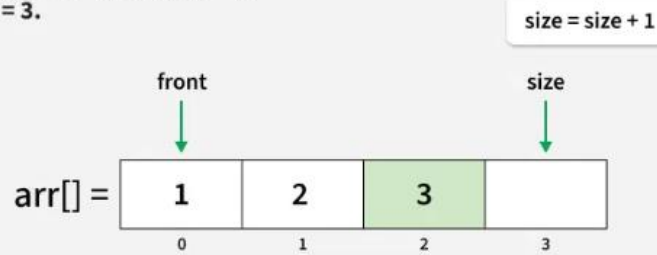
Fixed-Size Array Queue:

- A fixed-size array `arr[]` to store the elements.
- A variable `size` to track the current number of elements in the queue.
- A variable `capacity` to represent the maximum number of elements the queue can hold.



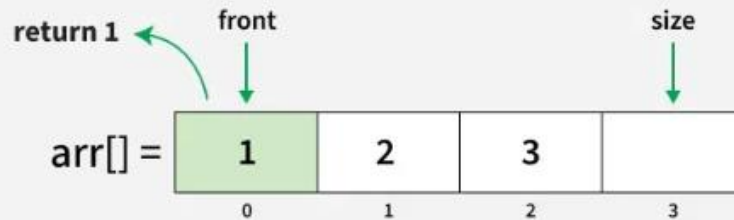
03
Step

enqueue(3): Insert the element into the queue at the index corresponding to the current size, provided the queue is not full.
front = 0, size = 2, capacity = 4,
arr[2] = 3.



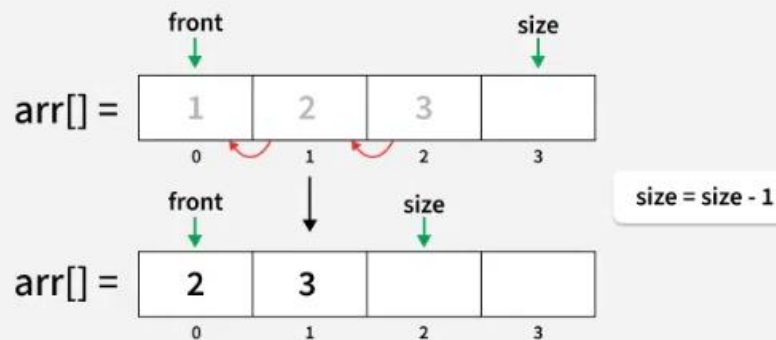
04
Step

getFront(): Returns the front element of the queue if the size is not 0; otherwise, returns -1



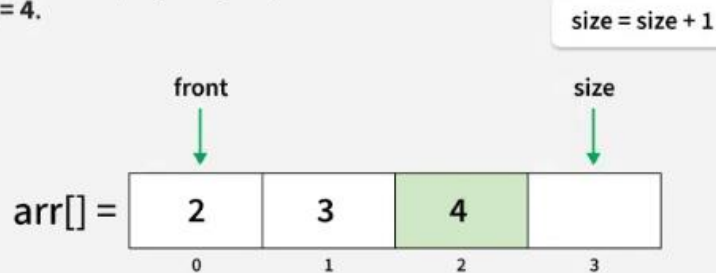
05
Step

dequeue(): Removes the front element of the queue by shifting all remaining elements one position to the left and then decrements the size.



06
Step

enqueue(4): Insert the element into the queue at the index corresponding to the current size, provided the queue is not full.
front = 0, size = 2, capacity = 4,
arr[2] = 4.



```

class myQueue {
    private int[] arr;
    private int capacity;
    private int size;

    // Constructor
    public myQueue(int capacity) {

        //Maximum number of elements the queue can hold.
        this.capacity = capacity;

        // Array to store queue elements.
        arr = new int[capacity];

        // Current number of elements in the queue.
        size = 0;
    }
}

```

Operations on Queue:

Enqueue (Insert):

- Add element at the end of the queue if space is available; otherwise, it results in an Overflow condition.
- Time: $O(1)$, Space: $O(1)$

```

void enqueue(int x) {
    if (size == capacity) {
        System.out.println("Queue Overflow");
        return;
    }
    arr[size++] = x;
}

```

Dequeue:

- Remove element from the front of the queue; if the queue is empty, it results in an Underflow condition.
- Time: $O(n)$ (because of shifting) , Space : $O(1)$

getFront (Peek):

- Return first element if not empty, else -1.
- Time: $O(1)$, Space: $O(1)$

```

int getFront() {
    if (size == 0) {
        System.out.println("Queue is empty");
        return -1;
    }

    return arr[0];
}

```

```
}
```

getRear():

- Return last element if not empty, else -1.
- Time: $O(1)$, Space: $O(1)$

```
public int getRear() {  
    if (isEmpty()) {  
        System.out.println("Queue is empty!");  
        return -1;  
    }  
    return arr[size - 1];  
}
```

isEmpty():

- Checks whether the queue has any elements or not.
- Returns true if the queue is empty, otherwise false.
- Time: $O(1)$, Space: $O(1)$

```
public boolean isEmpty() {  
    return size == 0;  
}
```

isFull():

- Checks whether the queue has reached its maximum capacity.
- Returns true if the queue is full, otherwise false.
- Time: $O(1)$, Space: $O(1)$

```
public boolean isFull() {  
    return size == capacity;  
}
```

Full Implementations of Queue using Array:

```
class myQueue {  
    private int[] arr;  
    private int capacity;  
    private int size;  
    // Constructor  
    public myQueue(int capacity) {  
        //Maximum number of elements the queue can hold.  
        this.capacity = capacity;  
        // Array to store queue elements.  
        arr = new int[capacity];  
        // Current number of elements in the queue.  
        size = 0;  
    }  
    // Check if queue is empty  
    public boolean isEmpty() {  
        return size == 0;  
    }  
}
```

```

    }
    // Check if queue is full
    public boolean isFull() {
        return size == capacity;
    }
    // Enqueue
    public void enqueue(int x) {
        if (isFull()) {
            System.out.println("Queue is full!");
            return;
        }
        arr[size] = x;
        size++;
    }
    // Dequeue
    public void dequeue() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return;
        }
        for (int i = 1; i < size; i++) {
            arr[i - 1] = arr[i];
        }
        size--;
    }

    // Get front element
    public int getFront() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1;
        }
        return arr[0];
    }
    //Get last element
    public int getRear() {
        if (isEmpty()) {
            System.out.println("Queue is empty!");
            return -1;
        }
        return arr[size - 1];
    }
}

public class Main {
    public static void main(String[] args) {
        myQueue q = new myQueue(3);
        q.enqueue(10);
        q.enqueue(20);
        q.enqueue(30);
        System.out.println("Front: " + q.getFront());
    }
}

```

```

        q.dequeue();
        System.out.println("Front: " + q.getFront());
        System.out.println("Rear: " + q.getRear());
        q.enqueue(40);
    }
}

```

Output

```

Front: 10
Front: 20
Rear: 30

```

Queue - Linked List Implementation:

A Queue is a linear data structure that follows the **First-In-First-Out (FIFO)** principle. The element inserted first is the first one to be removed.



Declaration of Queue using Linked List

To implement a queue with a linked list, we maintain:

A Node structure/class that contains:

- data → to store the element.
- next → pointer/reference to the next node in the queue.

Two pointers/references:

- front → points to the first node (head of the queue).
- rear → points to the last node (tail of the queue).

```

class Node {
    int data;
    Node next;
    Node(int new_data) {
        data = new_data;
    }
}

```

```

        next = null;
    }
}

// Queue class definition
class myQueue {
    private Node front;
    private Node rear;

    public myQueue() {
        front = rear = null;
    }
};

```

Operations on Queue using Linked List:

Enqueue Operation

The enqueue operation adds an element to the rear of the queue. Unlike array implementation, there is no fixed capacity in linked list. Overflow occurs only when memory is exhausted.

- A new node is created with the given value.
- If the queue is empty (`front == null` and `rear == null`), both front and rear are set to this new node.
- Otherwise, the current rear's next pointer is set to the new node.
- The rear pointer is updated to point to the new node.

```

public void enqueue(int new_data) {
    Node new_node = new Node(new_data);
    if (isEmpty()) {
        front = rear = new_node;
    } else {
        rear.next = new_node;
        rear = new_node;
    }
}

```

Dequeue Operation

The dequeue operation removes an element from the front of the queue.

- If the queue is empty (`front == null`), return underflow (queue is empty).
- Otherwise, store the current front node in a temporary pointer.
- Move the front pointer to the next node (`front = front.next`).
- If the front becomes null, also set `rear = null` (queue becomes empty).

```

public void dequeue() {
    if (isEmpty()) {
        System.out.println("Queue Underflow");
    }
}

```

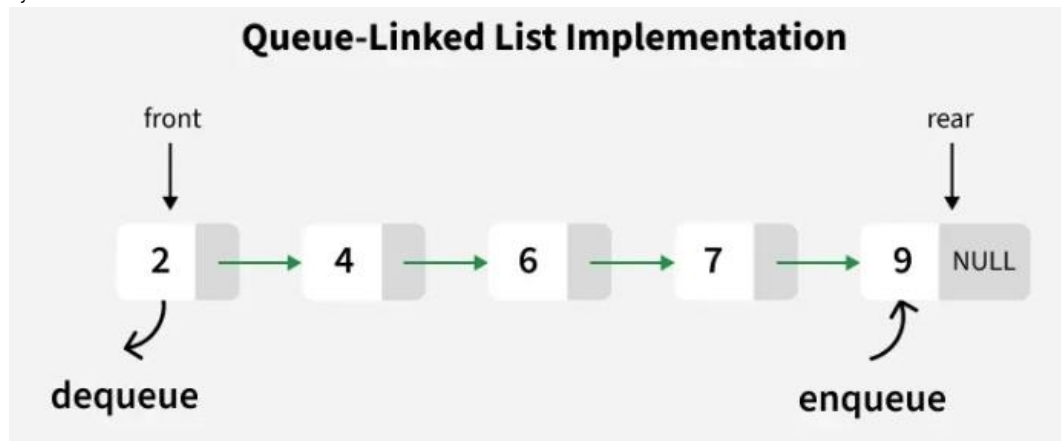


```

        return;
    }

    Node temp = front;
    front = front.next;
    if (front == null) rear = null;
    temp = null;
}

```



isEmpty Operation

The isEmpty operation checks whether the queue has no elements.

- If the front pointer is NULL, it means the queue is empty → return true.
- Otherwise, the queue has elements → return false.

```

// Check if the queue is empty
public boolean isEmpty() {
    return front == null;
}

```

Front Operation

The front() function returns the element at the front of the queue without removing it.

- If the queue is empty (front == NULL), print a message and return -1.
- Otherwise, return front->data (the value at the front).

```

// Return front element
public int getfront() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }
    return front.data;
}

```

Full Implementations of Queue Using Linked List

```
// Node class definition
class Node {
    int data;
    Node next;
    Node(int new_data) {
        data = new_data;
        next = null;
    }
}

// Queue class definition
class myQueue {
    private Node front;
    private Node rear;
    private int currSize;
    public myQueue() {
        currSize = 0;
        front = rear = null;
    }

    // Check if the queue is empty
    public boolean isEmpty() {
        return front == null;
    }

    // Enqueue operation
    public void enqueue(int new_data) {
        Node new_node = new Node(new_data);
        if (isEmpty()) {
            front = rear = new_node;
        } else {
            rear.next = new_node;
            rear = new_node;
        }
    }
}
```

```

    }

    currSize++;
}

// Dequeue operation (returns removed element)
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue Underflow");
        return -1;
    }

    Node temp = front;
    int removedData = temp.data;
    front = front.next;
    if (front == null) rear = null;
    currSize--;
    return removedData;
}

// Return front element
public int getfront() {
    if (isEmpty()) {
        System.out.println("Queue is empty");
        return -1;
    }

    return front.data;
}

// Return size of the queue
public int size() {
    return currSize;
}
}

class GFG {

```

```

public static void main(String[] args) {

    myQueue q = new myQueue();

    q.enqueue(10);

    q.enqueue(20);

    System.out.println("Dequeue: " + q.dequeue());

    q.enqueue(30);

    System.out.println("Front: " + q.getfront());

    System.out.println("Size: " + q.size());

}
}

```

Output

```

Dequeue: 10
Front: 20
Size: 2

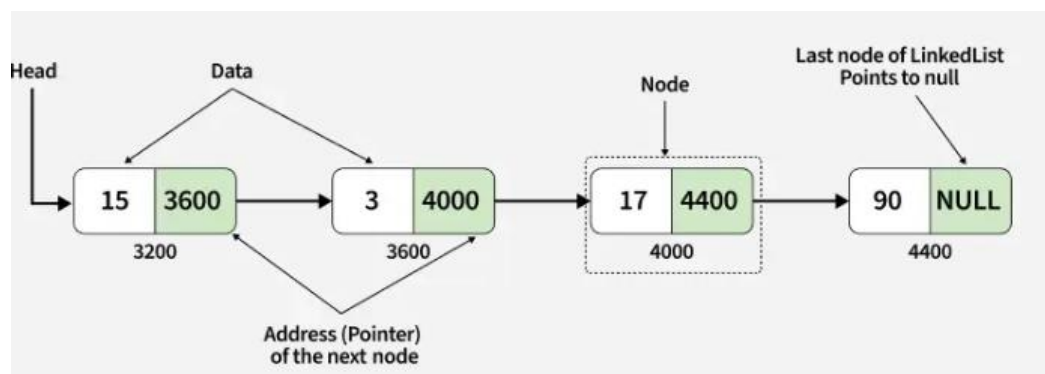
```

Linked list:

A linked list is a fundamental data structure in computer science. It mainly allows efficient insertion and deletion operations compared to arrays. Like arrays, it is also used to implement other data structures like stack, queue and deque.

A linked list is a type of linear data structure individual items are not necessarily at contiguous locations. The individual items are called nodes and connected with each other using links.

- A node contains two things first is data and second is a link that connects it with another node.
- The first node is called the head node and we can traverse the whole list using this head and next links.



Here's the comparison of Linked List and Arrays

Linked List:

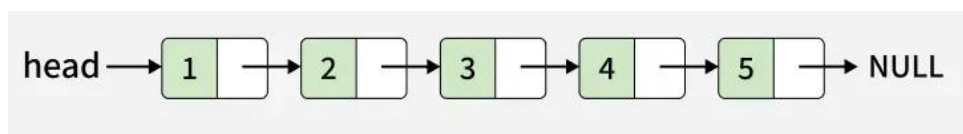
- **Data Structure:** Non-contiguous
- **Memory Allocation:** Typically allocated one by one to individual elements
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

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

singly linked list :

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



Understanding Node Structure:

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

```
// Definition of a Node in a singly linked list
public class Node {

    // Data part of the node
    int data;

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

    // Constructor to initialize the node with data
    public Node(int data){
        this.data = data;
        this.next = null;
    }
}
```

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

Creating an Example Linked List of Size 3 to Understand Working

Create the first node

- Allocate memory for the first node and Store data in it.
- Mark this node as head.

Create the second node

- Allocate memory for the second node and Store data in it.
- Link the first node's next to this new node.

Create the third node

- Allocate memory for the third node and Store data in it.
- Link the second node's next to this node.
- Set its next to NULL to ensure that the next of the last is NULL.

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

    // constructor to initialize a new node with data
    Node(int new_data) {
        this.data = new_data;
        this.next = null;
    }
}
```

```
public class Main {
    public static void main(String[] args) {

        // Create the first node (head of the list)
        Node head = new Node(10);

        // Link the second node
        head.next = new Node(20);

        // Link the third node
        head.next.next = new Node(30);

        // Link the fourth node
        head.next.next.next = new Node(40);

        // printing linked list
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
    }
}
```

Output

```
10 20 30 40
```

Common Operation in Linked List

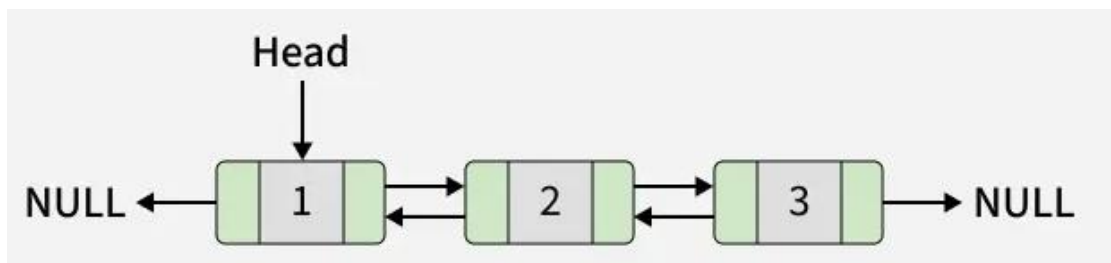
A linked list supports several operations. Here are the most common ones:

- **Traversal** : Traversing singly linked list

- **Insertion** : At the beginning, At the end and At a specific position
- **Deletion** : From beginning, From end and From a specific position
- **Searching** : Find whether a given key exists in the list
- **Updating (Modification)** : Modify contents of linked list.
- **Reversal** : Reverse the linked list and make the last node as new head.

Doubly linked list:

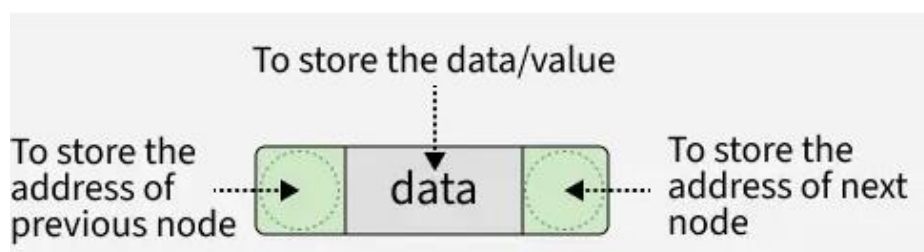
A doubly linked list is a more complex data structure than a singly linked list, but it offers several advantages. The main advantage of a doubly linked list is that it allows for efficient traversal of the list in both directions. This is because each node in the list contains a pointer to the previous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



Representation of Doubly Linked List in Data Structure:

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

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



Node Definition

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

```

class Node {
    // To store the Value or data.
    int data;

    // Reference to the Previous Node
    Node prev;

    // Reference to the next Node
    Node next;

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

```

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

Creating a Doubly Linked List with 4 Nodes

Create the head node.

- Allocate a node and set head to it. Its prev and next should be null/None.

Create the next node and link it to head.

- head.next = new Node(value2)
- head.next.prev = head

Create further nodes the same way.

- For the third node:
=> head.next.next = new Node(value3)
=> head.next.next.prev = head.next
- Repeat until you have the required nodes.

Ensure the tail's next is null.

The last node you created must have next == null

Set / keep track of head (and optionally tail).

Use head to access the list from the front. Keeping a tail pointer simplifies appends.

```
class Node {  
    int data;  
    Node prev;  
    Node next;  
    Node(int value) {  
        data = value;  
        prev = null;  
        next = null;  
    }  
}  
  
class GfG {  
    public static void main(String[] args) {  
        // Create the first node (head of the list)  
        Node head = new Node(10);  
        // Create and link the second node  
        head.next = new Node(20);  
        head.next.prev = head;  
        // Create and link the third node  
        head.next.next = new Node(30);  
        head.next.next.prev = head.next;  
        // Create and link the fourth node  
        head.next.next.next = new Node(40);  
        head.next.next.next.prev = head.next.next;  
        // Traverse the list forward and print elements  
        Node temp = head;  
        while (temp != null) {
```

```

        System.out.print(temp.data);

        if (temp.next != null) {

            System.out.print(" <-> ");

        }

        temp = temp.next;

    }

}

```

Output

```
10 <-> 20 <-> 30 <-> 40
```

Common Operation in Doubly Linked List

- **Traversal** : Display Linked List Elements
- **Insertion** : At the Beginning, At the End and At the specific position
- **Deletion** : From the Beginning, From End and From a Specific Position

circular linked list :

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

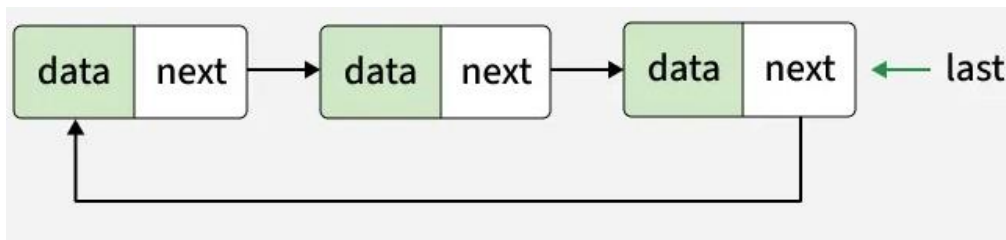
- **Structure:** All nodes are connected in a circle, enabling continuous traversal without encountering NULL.
- **Difference from Regular Linked List:** In a regular linked list, the last node points to NULL, whereas in a circular linked list, it points to the first node.
- **Uses:** Ideal for tasks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

Types of Circular Linked Lists

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

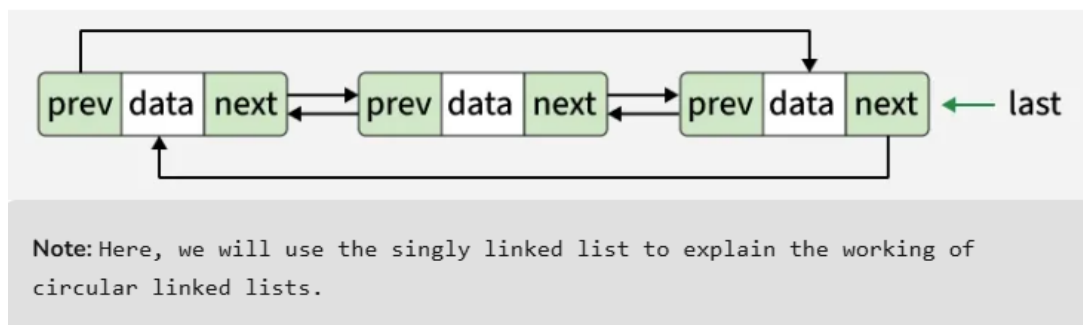
1. Circular Singly Linked List

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



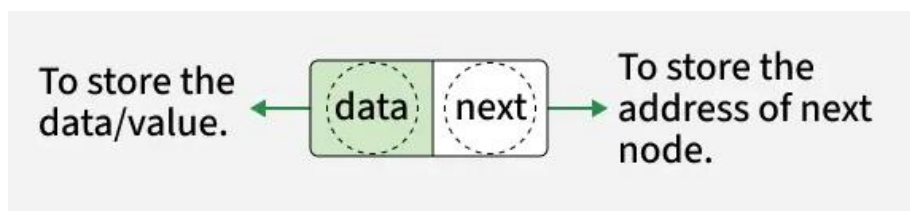
2. Circular Doubly Linked List:

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



Representation of a Circular Singly Linked List:

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



Create/Declare a Node of Circular Linked List:

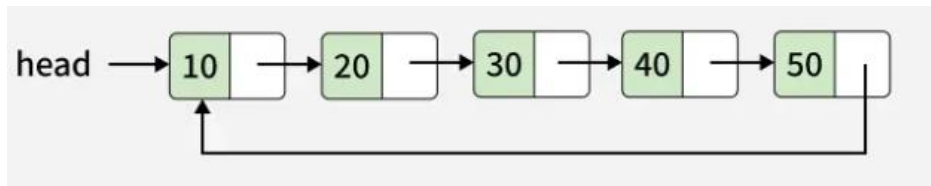
```
class Node {
    int data;
    Node next;

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

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

Example of Creating a Circular Linked List

Here's an example of creating a circular linked list with three nodes (10, 20, 30, 40, 50):



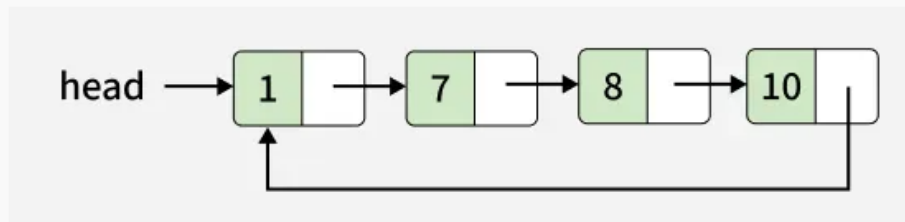
Operations on the Circular Linked List

- **Insertion** : At the Beginning, At the End and At a Specific Position
- **Deletion** : Removal from different positions

Insertion at the beginning in circular linked list:

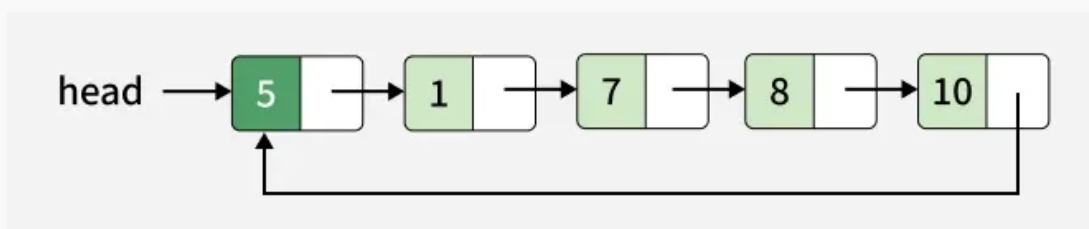
Given a Circular Linked List and an integer key representing a node. Insert the given key node at the beginning of the circular linked list.

Input: key = 5

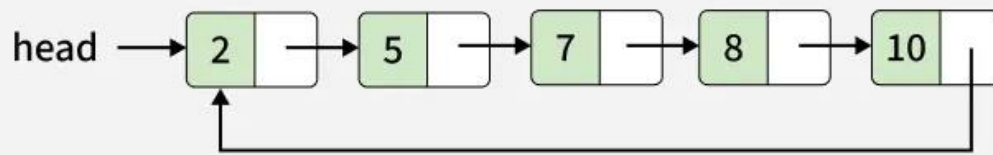


Output: 5 -> 1 -> 7 -> 8 -> 10

Explanation: After inserting 5 at the beginning of the given circular linked list, it has elements as 5, 1, 7, 8, 10.

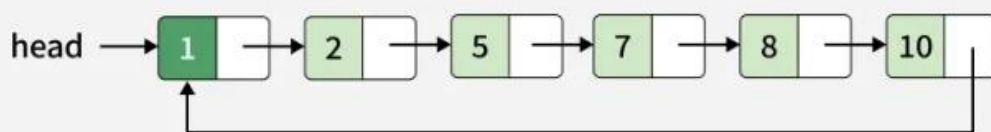


Input: key = 1



Output: 1 -> 2 -> 5 -> 7 -> 8 -> 10

Explanation: After inserting 1 at the beginning of the given circular linked list, it has elements as 1, 2, 5, 7, 8, 10.



Step-by-step approach:

- Create a new node with the given value.
- Check Empty List (**last == nullptr**):
 - Make **newNode->next** point to itself.
- Insert at Beginning:
 - Set **newNode->next** to **last->next**.
 - Update **last->next** to **newNode**.

Below is the implementation of the above approach:

```
class Node {  
  
    int data;  
  
    Node next;  
  
    Node(int x) {  
  
        data = x;  
  
        next = null;  
  
    }  
  
}
```

```
class GfG {
```

```
static Node insertAtBeginning(Node last, int key) {
```

```
    Node newNode = new Node(key);
```

```
    if (last == null) {
```

```
        newNode.next = newNode;
```

```
        return newNode;
```

```
    }
```

```
    newNode.next = last.next;
```

```
    last.next = newNode;
```

```
    return last;
```

```
}
```

```
static void printList(Node last) {
```

```
    if (last == null) return;
```

```
    Node head = last.next;
```

```
    Node temp = head;
```

```
    do {
```

```
        System.out.print(temp.data);
```

```
        temp = temp.next;
```

```
        if (temp != head) System.out.print(" -> ");
```

```
    } while (temp != head);
```

```
    System.out.println();
```

```
}
```

```
public static void main(String[] args) {
```

```
    // Create circular linked list: 2 -> 3 -> 4
```

```
    Node first = new Node(2);
```

```
    first.next = new Node(3);
```

```

first.next.next = new Node(4);

Node last = first.next.next;

last.next = first;

// Insert 5 at the beginning

last = insertAtBeginning(last, 5);

printList(last);

}

}

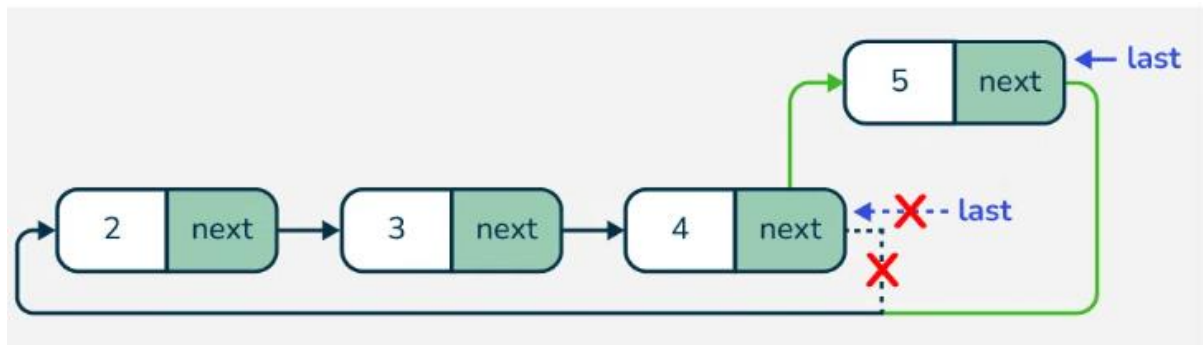
```

Output

```
5 -> 2 -> 3 -> 4
```

Insertion at the end in circular linked list:

A circular linked list is a data structure where each node points to the next, and the last node connects back to the first, creating a loop. **Insertion at the end in circular linked list** is an important operation. Understanding how to perform this insertion is essential for effectively manage and use circular linked lists in various applications.



Step-by-step approach:

- Create a new node with the given value.
- Check Empty List, If **last** is **nullptr** then initialize the list with the new node and make it point to itself.
- Otherwise, Set the **new node's** next to the **head** node.
- Update the current **last's** next to point to the **new node**.
- Update **last** to the **new node**.

Below is the implementation of the above approach:

```

class Node {
    int data;
    Node next;
}

```

```

Node(int value){
    data = value;
    next = null;
}
}

public class GFG {

    // Function to insert a node at the end of a circular
    // linked list
    static Node insertEnd(Node tail, int value){
        Node newNode = new Node(value);
        if (tail == null) {
            // If the list is empty, initialize it with the
            // new node
            tail = newNode;
            newNode.next = newNode;
        }
        else {
            // Insert new node after the current tail and
            // update the tail pointer
            newNode.next = tail.next;
            tail.next = newNode;
            tail = newNode;
        }
        return tail;
    }

    // Function to print the circular linked list
    static void printList(Node last){
        if (last == null) return;

        Node head = last.next;
        while (true) {

```



```

        System.out.print(head.data + " ");
        head = head.next;
        if (head == last.next) break;
    }
    System.out.println();
}

public static void main(String[] args){
    // Create circular linked list: 2, 3, 4
    Node first = new Node(2);
    first.next = new Node(3);
    first.next.next = new Node(4);
    Node last = first.next.next;
    last.next = first;
    System.out.print("Original list: ");
    printList(last);
    // Insert elements at the end of the circular linked
    // list
    last = insertEnd(last, 5);
    last = insertEnd(last, 6);
    System.out.print("List after inserting 5 and 6: ");
    printList(last);
}
}

```

Output

Original list: 2 3 4

List after inserting 5 and 6: 2 3 4 5 6

Insertion at specific position in circular linked list:

Inserting an element at a specific position in a circular linked list is a common operation that involves adjusting pointers in a circular structure. Unlike a regular linked list, where the last node points to NULL, a circular linked list's last node points back to the head, forming a loop. This property makes insertion at various positions, including the beginning, middle, and end, slightly different compared to a linear linked list.

Examples:

Input: data = 5, pos = 2

Output: Original list: 2 3 4

List after insertions: 2 5 3 4

Explanation: The new node with data 5 is inserted at position 2. The list becomes 2 -> 5 -> 3 -> 4.

Input: data = 1, pos = 1

Output: Original list: 2 3 4

List after insertions: 1 2 3 4

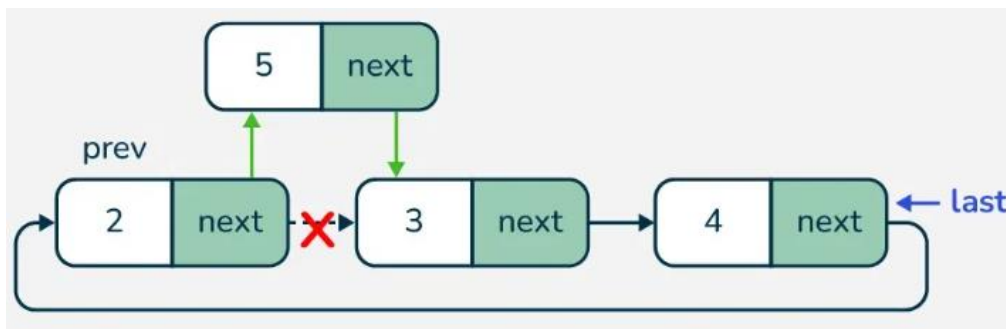
Explanation: The new node with data 1 is inserted at the beginning (position 1). The list becomes 1 -> 2 -> 3 -> 4.

Input: data = 6, pos = 5

Output: Original list: 2 3 4

List after insertions: Invalid position!

Explanation: The position 5 is invalid because the list has only 3 elements. Therefore, no insertion is made, and the list remains unchanged.



Steps for Insertion at a Specific Position in a Circular Linked List

- **Check if the list is empty**
 - If the position is not 1, print an error message (since the position doesn't exist in an empty list).
 - If the position is 1, create the new node and make its next pointer point to itself (since it is the only node).
- **If the list is not empty:** Create the new node with the given data.
- **For insertion at position 1 (insert at the beginning):**
 - Traverse the list to find the last node (the node whose next pointer points to the head).
 - Set the new node's next pointer to the current head.
 - Set the next pointer of the last node to the new node.
 - Update the head to point to the new node.
- **For insertion at positions other than 1**
 - Traverse the list to reach the node just before the desired position (position - 1).
 - Once you reach that node, set the new node's next pointer to point to the node that was previously at the desired position.
 - Set the (position-1)-th node's next pointer to the new node.
- **If the new node is inserted at the end**
 - Update the last node's next pointer to point to the new node, maintaining the circular nature of the list.
 - Ensure the new node's next pointer points to the head, completing the circle.
 - This process ensures that the circular structure of the linked list is preserved after every insertion.

```

class Node {
    int data;
    Node next;

    Node(int value) {
        data = value;
        next = null;
    }
}

public class CircularLinkedList {

    // Function to insert a node at a specific position in a circular linked list
    public static Node insertAtPosition(Node last, int data, int pos) {
        if (last == null) {
            // If the list is empty
            if (pos != 1) {
                System.out.println("Invalid position!");
                return last;
            }
            // Create a new node and make it point to itself
            Node newNode = new Node(data);
            last = newNode;
            last.next = last;
            return last;
        }

        // Create a new node with the given data
        Node newNode = new Node(data);

        // curr will point to head initially
        Node curr = last.next;

        if (pos == 1) {
            // Insert at the beginning
            newNode.next = curr;
            last.next = newNode;
            return last;
        }

        // Traverse the list to find the insertion point
        for (int i = 1; i < pos - 1; ++i) {
            curr = curr.next;
            // If position is out of bounds (beyond the circular linked list)
            if (curr == last) {
                System.out.println("Invalid position!");
                return last;
            }
        }
        // Insert the new node at the desired position
        newNode.next = curr.next;

```

```

curr.next = newNode;

// Update last if the new node is inserted at the end
if (curr == last) {
    last = newNode;
}

return last;
}

// Function to print the circular linked list
public static void printList(Node last) {
    if (last == null) return;
    Node head = last.next;
    while (true) {
        System.out.print(head.data + " ");
        head = head.next;
        if (head == last.next) break;
    }
    System.out.println();
}

public static void main(String[] args) {
    // Create circular linked list: 2 -> 3 -> 4 -> (back to 2)
    Node first = new Node(2);
    first.next = new Node(3);
    first.next.next = new Node(4);
    first.next.next.next = first;

    Node last = first.next.next;
    System.out.print("Original list: ");
    printList(last);

    // Insert element at position 2
    int data = 5, pos = 2;
    last = insertAtPosition(last, data, pos);
    System.out.print("List after insertion: ");
    printList(last);
}
}

```

Output

Original list: 2 3 4

List after insertions: 2 5 3 4

There are three main ways to delete a node from circular linked list:

- Deletion at the beginning
- Deletion at specific position
- Deletion at the end

Now, let's look at the methods and steps for these three deletion operations.

Deletion from a Circular Linked List

Deletion involves removing a node from the linked list. The main difference is that we need to ensure the list remains circular after the deletion. We can delete a node in a circular linked list in three ways:

1. Deletion from the beginning of the circular linked list - O(1) Time and O(1) Space

To delete the first node of a circular linked list, check if the list is empty and return NULL if so. If it has only one node, delete it and set last = NULL. Otherwise, update last->next to skip the head, delete the head node, and return the updated last pointer.

Step-by-step approach:

- Check if List is Empty: If last is nullptr, print "List is empty" and return nullptr.
- Get Head Node: Set head to last->next.
- Check for Single Node: If head equals last, delete head and set last to nullptr.
- Handle Multiple Nodes:
 - => Update last->next to point to head->next.
 - => Delete head.
- Return Updated last

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

public class GFG {
    public static Node deleteFirstNode(Node last) {
        if (last == null) {
            // If the list is empty
            System.out.println("List is empty");
            return null;
        }
        Node head = last.next;
```

```

if (head == last) {

    // If there is only one node in the list
    last = null;
} else {
    // More than one node in the list
    last.next = head.next;
}
return last;
}

public static void printList(Node last) {
    if (last == null) return;
    Node head = last.next;
    while (true) {
        System.out.print(head.data);
        head = head.next;
        if (head == last.next) break;
        System.out.print(" <-> ");
    }
    System.out.println();
}

public static void main(String[] args) {
    Node first = new Node(2);
    first.next = new Node(3);
    first.next.next = new Node(4);
    Node last = first.next.next;
    last.next = first;
    last = deleteFirstNode(last);
    printList(last);
}
}

```

Output

```
3 <-> 4
```

Evaluation of Expressions:

The way to write arithmetic expression is known as a **notation**. An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are –

- Infix Notation
- Prefix (Polish) Notation
- Postfix (Reverse-Polish) Notation

These notations are named as how they use operator in expression. We shall learn the same here in this chapter.

Infix Notation:

We write expression in **infix** notation, e.g. $a - b + c$, where operators are used **in**-between operands. It is easy for us humans to read, write, and speak in infix notation but the same does not go well with computing devices. An algorithm to process infix notation could be difficult and costly in terms of time and space consumption.

Prefix Notation

In this notation, operator is **prefixed** to operands, i.e. operator is written ahead of operands. For example, **+ab**. This is equivalent to its infix notation **a + b**. Prefix notation is also known as **Polish Notation**.

Postfix Notation :

This notation style is known as **Reversed Polish Notation**. In this notation style, the operator is **postfixed** to the operands i.e., the operator is written after the operands. For example, **ab+**. This is equivalent to its infix notation **a + b**.

The following table briefly tries to show the difference in all three notations –

Sr.No.	Infix Notation	Prefix Notation	Postfix Notation
1	$a + b$	$+ a b$	$a b +$
2	$(a + b) * c$	$* + a b c$	$a b + c *$
3	$a * (b + c)$	$* a + b c$	$a b c + *$
4	$a / b + c / d$	$+ / a b / c d$	$a b / c d / +$
5	$(a + b) * (c + d)$	$* + a b + c d$	$a b + c d + *$
6	$((a + b) * c) - d$	$- * + a b c d$	$a b + c * d -$

Parsing Expressions :

As we have discussed, it is not a very efficient way to design an algorithm or program to parse infix notations. Instead, these infix notations are first converted into either postfix or prefix notations and then computed.

To parse any arithmetic expression, we need to take care of operator precedence and associativity also.

Precedence :

When an operand is in between two different operators, which operator will take the operand first, is decided by the precedence of an operator over others. For example – As multiplication operation has precedence over addition, $b * c$ will be evaluated first. A table of operator precedence is provided later.

Associativity

Associativity describes the rule where operators with the same precedence appear in an example,

expression. For in expression $a + b - c$, both $+$ and $-$ have the same precedence, then which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both $+$ and $-$ are left associative, so the expression will be evaluated as $(a + b) - c$.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Sr.No.	Operator	Precedence
1	Exponentiation $^$	Highest
2	Multiplication $(*)$ & Division $(/)$	Second Highest
3	Addition $(+)$ & Subtraction $(-)$	Lowest

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example – In $a + b * c$, the expression part $b * c$ will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for $a + b$ to be evaluated first, like $(a + b) * c$.

Postfix Evaluation Algorithm

We shall now look at the algorithm on how to evaluate postfix notation – Step 1 – scan the expression from left to right

Step 2 – if it is an operand push it to stack

Step 3 – if it is an operator pull operand from stack and perform operation

Step 4 – store the output of step 3, back to stack

Step 5 – scan the expression until all operands are consumed

Step 6 – pop the stack and perform operation

Adding two polynomials using Linked List:

Given **two polynomial** numbers represented by a linked list. The task is to **add** these lists meaning the coefficients with the same variable powers will be added.

Note: Given polynomials are sorted in **decreasing** order of power.

Example:

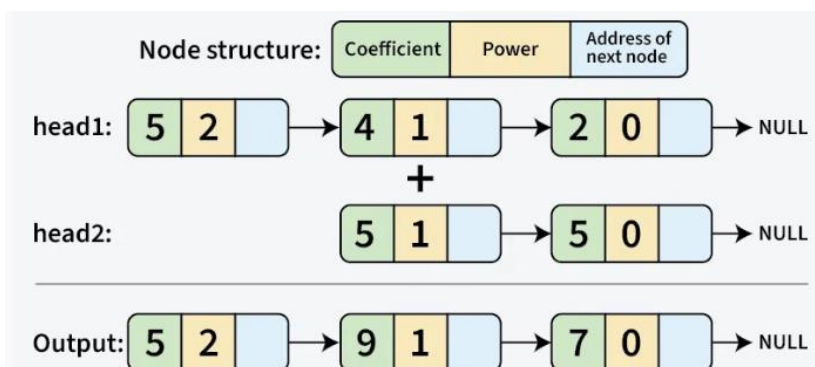
Input:

head1: $[[5, 2], [4, 1], [2, 0]]$

head2: $[[5, 1], [5, 0]]$

Output: $[[5, 2], [9, 1], [7, 0]]$

Explanation:



Input:

head1: [[5, 3], [4, 2], [2, 0]]

head2: [[5, 1], [-5, 0]]

Output: [[5, 3], [4, 2], [5, 1], [-3, 0]]

Explanation: head1 can be represented as $5x^3 + 4x^2 + 2$, head2 can be represented as $5x - 5$, add both the polynomials to get $5x^3 + 4x^2 + 5x - 3$

[Expected Approach - 1] Using Recursion - $O(m+n)$ Time and $O(\max(m,n))$ Space:

The idea is to **recursively** check the **heads** of both lists. If one of the heads is **NULL**, then return the other head. Otherwise, compare the **power** of both nodes. If the power of one list is **greater** than the other, then recursively find the **next** node of the **greater** power list. Otherwise, store the **sum** of coefficients in **one** list, and return its **head**.

Below is the implementation of the above approach:

/ Java program to add two polynomials

```
class Node {  
    int coeff;  
  
    int pow;  
  
    Node next;  
  
    Node(int c, int p) {  
        coeff = c;  
  
        pow = p;  
  
        next = null;  
    }  
}  
  
class Main {  
  
    static Node addPolynomial(Node head1, Node head2) {  
  
        // if any list is empty, then return  
  
        // the other list.
```

```

if (head1 == null) return head2;

if (head2 == null) return head1;

// If head1.pow is greater, then recursively find
// its next node, and return head1.

if (head1.pow > head2.pow) {
    Node nextPtr = addPolynomial(head1.next, head2);

    head1.next = nextPtr;

    return head1;
}

// If head2.pow is greater, then recursively find its
// next node, and return head2.

else if (head1.pow < head2.pow) {
    Node nextPtr = addPolynomial(head1, head2.next);

    head2.next = nextPtr;

    return head2;
}

// else store the sum of head1.coeff and head2.coeff in
// head1.coeff, then find its next node and return head1.

Node nextPtr = addPolynomial(head1.next, head2.next);

head1.coeff += head2.coeff;

head1.next = nextPtr;

return head1;
}

```

```

static void printList(Node head) {
    Node curr = head;
    while (curr != null) {
        System.out.print(curr.coeff + "," + curr.pow + " ");
        curr = curr.next;
    }
    System.out.println();
}

public static void main(String[] args) {
    // 1st polynomial: 5x^2+4x^1+2x^0
    Node head1 = new Node(5, 2);
    head1.next = new Node(4, 1);
    head1.next.next = new Node(2, 0);
    // 2nd polynomial: -5x^1-5x^0
    Node head2 = new Node(-5, 1);
    head2.next = new Node(-5, 0);
    Node head = addPolynomial(head1, head2);
    printList(head);
}
}

```

[Expected Approach] Using Iterative Method - O(m+n) Time and O(1) Space:

The idea is to create a **dummy** node which will act as the **head** of resultant list. Start traversing both the lists, if **list1->pow** if **not equal to list2->pow**, then Link the node with **greater power** to the **resultant** list. Otherwise, add the sum of **list1->coeff + list2->coeff** to the **resultant** list.

Below is the implementation of the above approach:

/ Java program to add two polynomials

```
class Node {

    int coeff;

    int pow;

    Node next;

    Node(int c, int p) {

        coeff = c;

        pow = p;

        next = null;

    }

}

class GfG {

    static Node addPolynomial(Node head1, Node head2) {

        Node dummy = new Node(0, 0);

        // Node to append other nodes to the end

        // of list

        Node prev = dummy;

        Node curr1 = head1, curr2 = head2;

        while (curr1 != null && curr2 != null) {

            // if curr2.pow > curr1.pow, then

            // append curr2 to list

            if (curr1.pow < curr2.pow) {

                prev.next = curr2;
```

```

    prev = curr2;

    curr2 = curr2.next;

}

// if curr1.pow > curr2.pow, then

// append curr2 to list

else if (curr1.pow > curr2.pow) {

    prev.next = curr1;

    prev = curr1;

    curr1 = curr1.next;

}

// else, add the sum of curr1.coeff and

// curr2.coeff to curr1.coeff, and append

// curr1 to the list

else {

    curr1.coeff = curr1.coeff + curr2.coeff;

    prev.next = curr1;

    prev = curr1;

    curr1 = curr1.next;

    curr2 = curr2.next;

}

}

// if curr1 is not null, then append the rest

// to the list

if (curr1 != null) {

```

```

        prev.next = curr1;

    }

    // if curr2 is not null, then append the rest
    // to the list

    if (curr2 != null) {

        prev.next = curr2;

    }

    return dummy.next;
}

static void printList(Node head) {

    Node curr = head;

    while (curr != null) {

        System.out.print(curr.coeff + "," + curr.pow + " ");

        curr = curr.next;

    }

    System.out.println();

}

public static void main(String[] args) {

    // 1st polynomial:  $5x^2 + 4x^1 + 2x^0$ 

    Node head1 = new Node(5, 2);

    head1.next = new Node(4, 1);

    head1.next.next = new Node(2, 0);

    // 2nd polynomial:  $-5x^1 - 5x^0$ 

    Node head2 = new Node(-5, 1);

```

```
head2.next = new Node(-5, 0);

Node head = addPolynomial(head1, head2);

printList(head);

}

}
```

UNIT II

Non-Linear Data Structures:

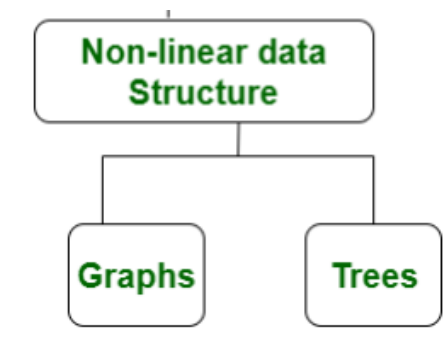
Data structures where data elements are not arranged sequentially or linearly are called **non-linear data structures**. In a non-linear data structure, single level is not involved. Therefore, we can't traverse all the elements in single run only. Non-linear data structures are not easy to implement in comparison to linear data structure. It utilizes computer memory efficiently in comparison to a linear data structure. Its examples are [trees](#) and [graphs](#).

1. Trees

A tree data structure consists of various nodes linked together. The structure of a tree is hierarchical that forms a relationship like that of the parent and a child. The structure of the tree is formed in a way that there is one connection for every parent-child node relationship. Only one path should exist between the root to a node in the tree. Various types of trees are present based on their structures like AVL tree, binary tree, binary search tree, etc.

2. Graph

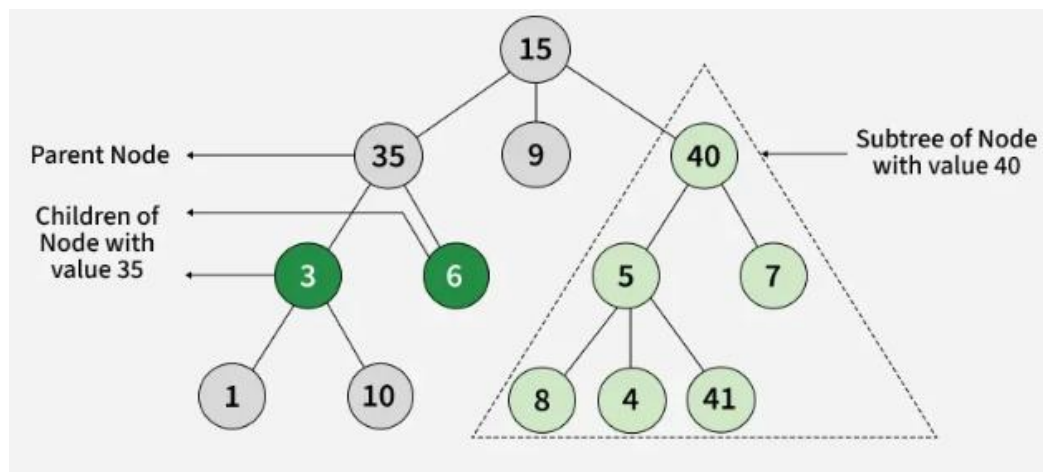
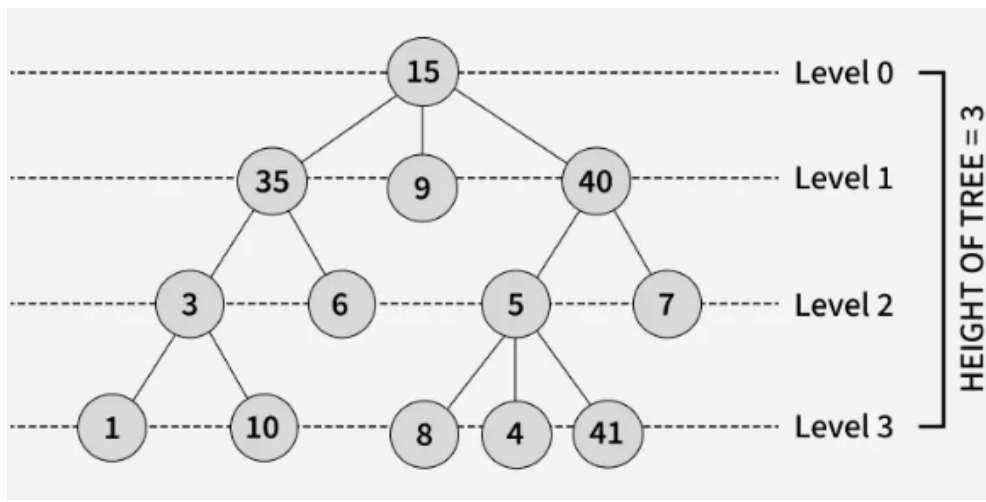
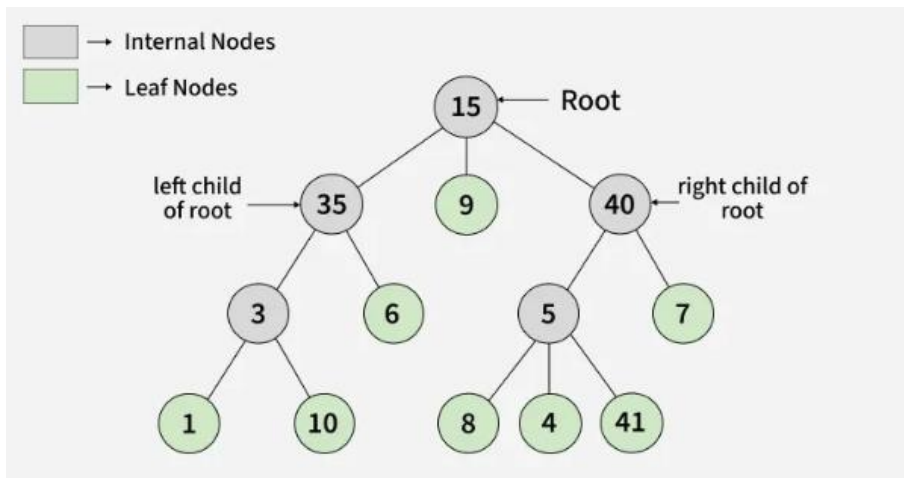
Graphs are those types of non-linear data structures which consist of a definite quantity of vertices and edges. The vertices or the nodes are involved in storing data and the edges show the vertices relationship. The difference between a graph to a tree is that in a graph there are no specific rules for the connection of nodes. Real-life problems like social networks, telephone networks, etc. can be represented through the graphs.

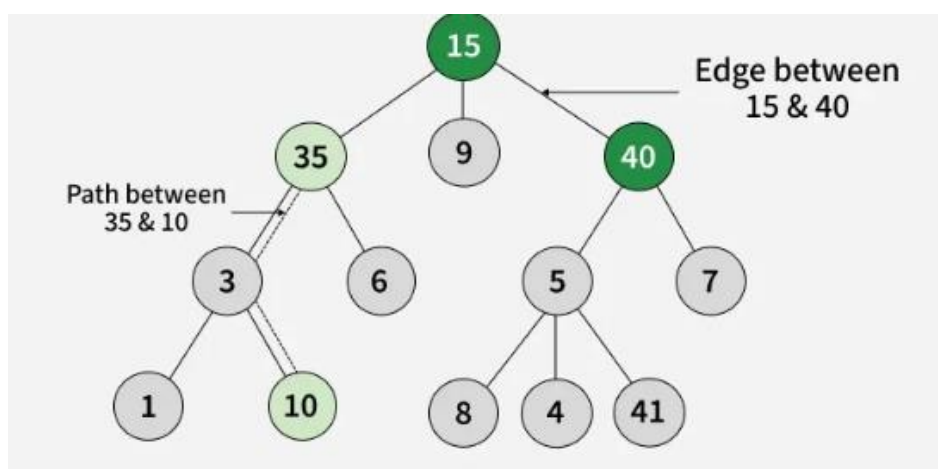
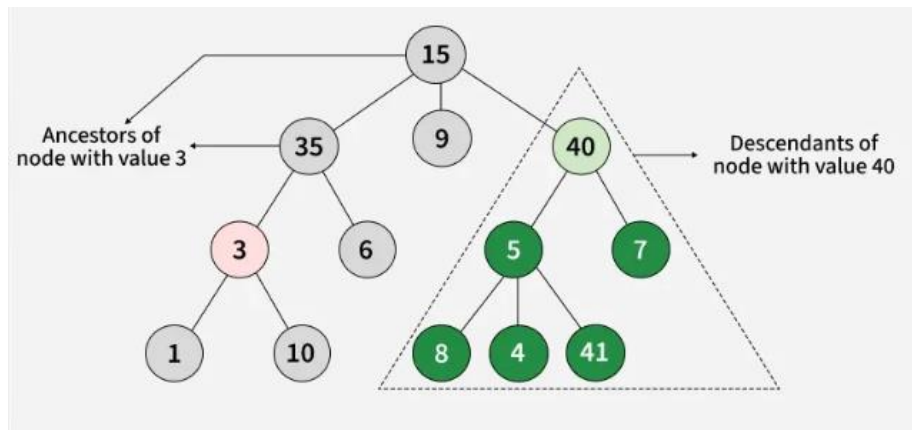


Trees:

A tree is a **hierarchical** data structure used to organize and represent data in a **parent-child** relationship.

It consists of nodes, where the topmost node is called the **root**, and every other node can have one or more **child nodes**.





Basic Terminologies In Tree Data Structure:

- **Parent Node:** A node that is an immediate predecessor of another node. Example: 35 is the parent of 3 and 6.
- **Child Node:** A node that is an immediate successor of another node. Example: 3 and 6 are children of 35.
- **Root Node:** The topmost node in a tree, which does not have a parent. Example: 15 is the root node.
- **Leaf Node (External Node):** Nodes that do not have any children. Example: 1, 10, 12, 5, 7, 7 are leaf nodes.
- **Ancestor:** Any node on the path from the root to a given node (excluding the node itself). Example: 15 and 35 are ancestors of 10.
- **Descendant:** A node x is a descendant of another node y if y is an ancestor of x . Example: 1, 10, and 6 are descendants of 35.
- **Sibling:** Nodes that share the same parent. Example: 1 and 10 are siblings, and 5 and 7 are siblings.
- **Level of a Node:** The number of edges in the path from the root to that node. The root node is at level 0.

- **Internal Node:** A node with at least one child.
- **Neighbor of a Node:** The parent or children of a node.
- **Subtree:** A node and all its descendants form a subtree.

Why Tree is considered a non-linear data structure?

Data in a tree is not stored sequentially (i.e., not in a linear order). Instead, it is organized across multiple levels, forming a hierarchical structure. Because of this arrangement, a tree is classified as a non-linear data structure.

Representation of a Node in Tree Data Structure:

A tree can be represented using a collection of nodes. Each of the nodes can be represented with the help of class or structs.

```
// Node structure for tree
class Node {
    int data;
    List<Node> children;

    Node(int x) {
        data = x;
        children = new ArrayList<>();
    }
}
```

Importance of Tree Data Structure:

- Trees are useful for storing data that naturally forms a hierarchy.
- File systems on computers are structured as trees, with folders containing subfolders and files.
- The DOM (Document Object Model) of an HTML page is a tree: The <html> tag is the root. <head> and <body> are its children. These tags can have their own child nodes, forming a hierarchical structure.
- Trees help in efficient data organization and retrieval for hierarchical relationships.

Types of Tree Data Structures :

Tree data structures can be classified based on the number of children each node can have.

Binary Tree:

Each node can have a maximum of two children.

- **Full Binary Tree** – every node has either 0 or 2 children.
- **Complete Binary Tree** – all levels are fully filled except possibly the last, which is filled from left to right.

- **Balanced Binary Tree** – height difference between left and right subtrees of every node is minimal.

Ternary Tree:

Each node can have at most three children, often labeled as left, middle, and right.

N-ary Tree (or Generic Tree):

- Each node can have any number of children.
- Each node typically contains: Some data, A list of references to its children (duplicates are not allowed).
- Unlike linked lists, nodes store references to multiple child nodes.

Please refer [Types of Trees in Data Structures](#) for details.

Basic Operations Of Tree Data Structure:

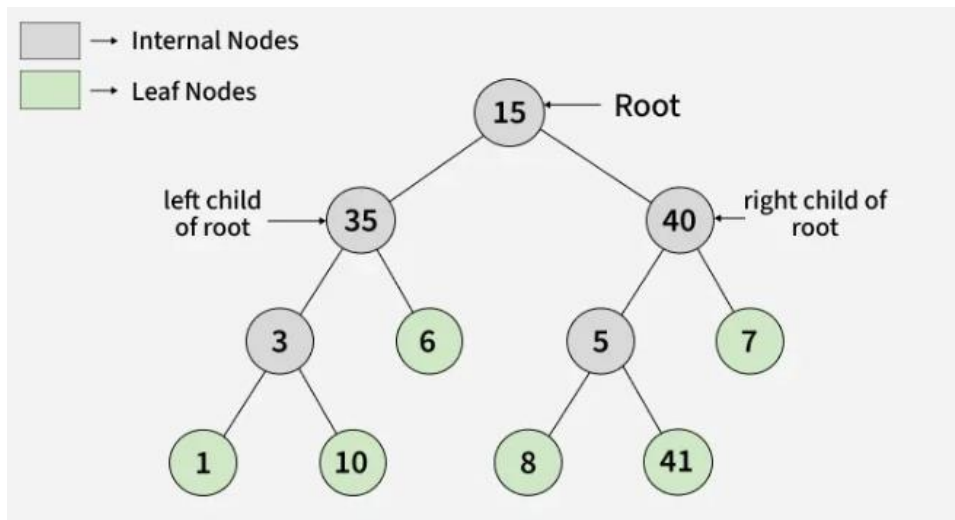
- **Create** – create a tree in the data structure.
- **Insert** – Inserts data in a tree.
- **Search** – Searches specific data in a tree to check whether it is present or not.
- **Traversal: Traversal:**
Depth-First-Search Traversal
Breadth-First-Search Traversal

Properties of Tree Data Structure:

- **Number of edges:** An edge is the connection between two nodes. A tree with N nodes will always have N - 1 edges. There is exactly one path from any node to any other node in the tree.
- **Depth of a node:** The depth of a node is the length of the path from the root to that node. Each edge in the path adds 1 unit to the length. Equivalently, it is the number of edges from the root to the node.
- **Height of the tree:** The height of the tree is the length of the longest path from the root to any leaf node.
- **Degree of a node:** The degree of a node is the number of subtrees attached to it (i.e., the number of children it has).
A leaf node has a degree of 0.
The degree of the tree is the maximum degree among all nodes in the tree.

Binary Tree :

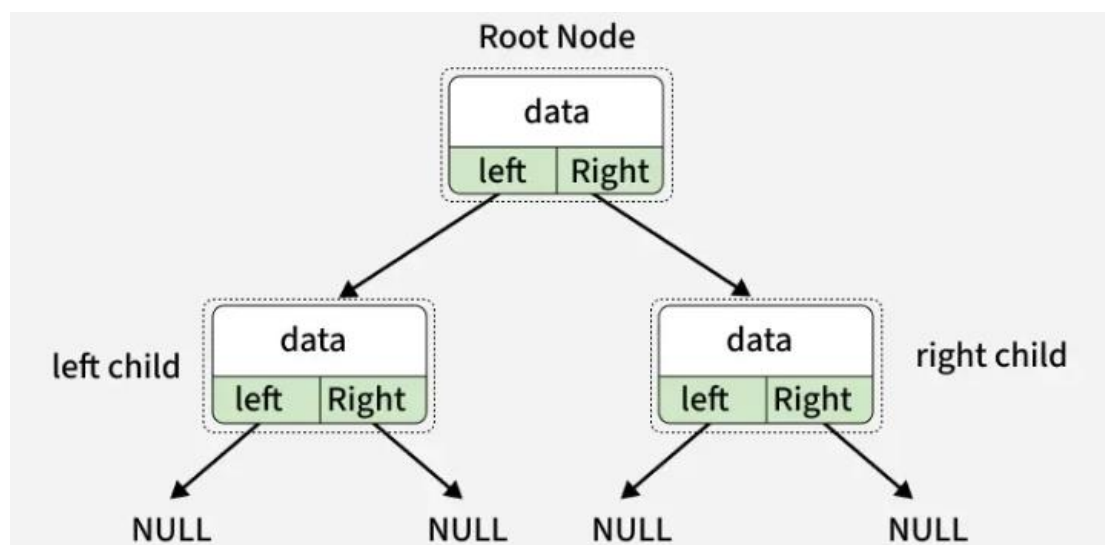
Binary Tree is a non-linear and hierarchical data structure where each node has **at most two children** referred to as the left child and the right child. The topmost node in a binary tree is called the root, and the bottom-most nodes(having no children) are called leaves.



Representation of Binary Tree

Each node in a Binary Tree has three parts:

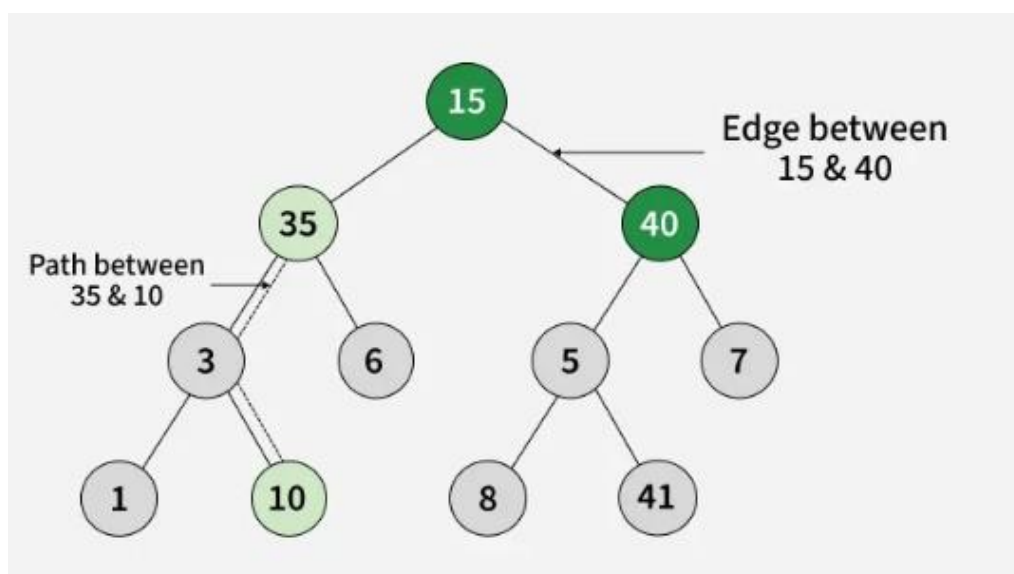
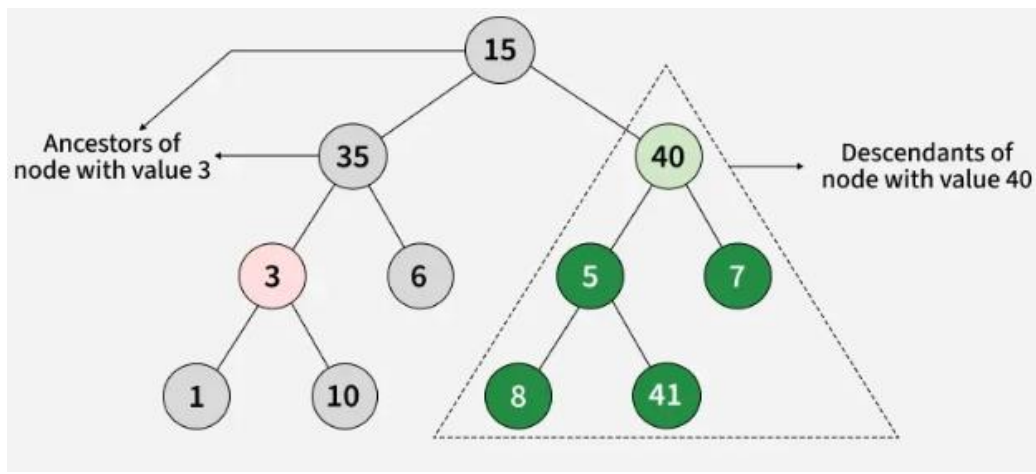
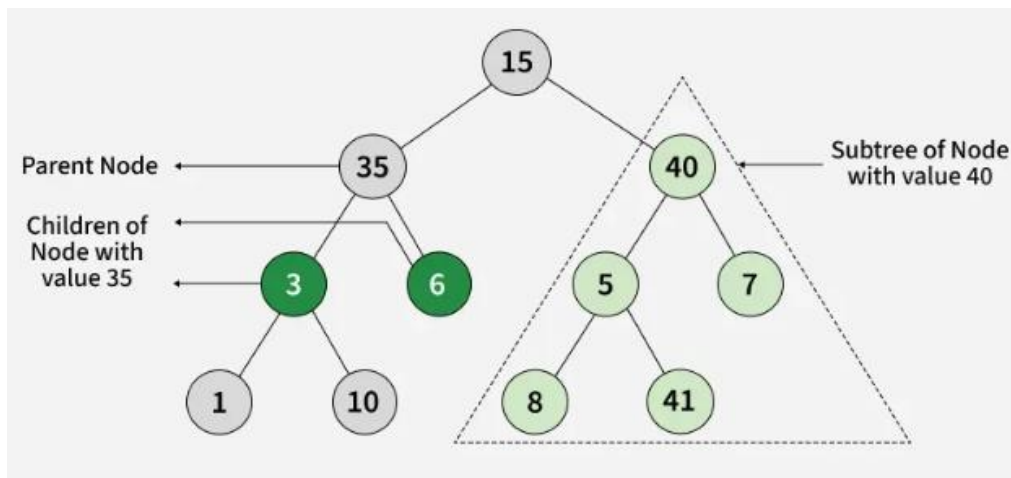
- Data
- Pointer to the left child
- Pointer to the right child

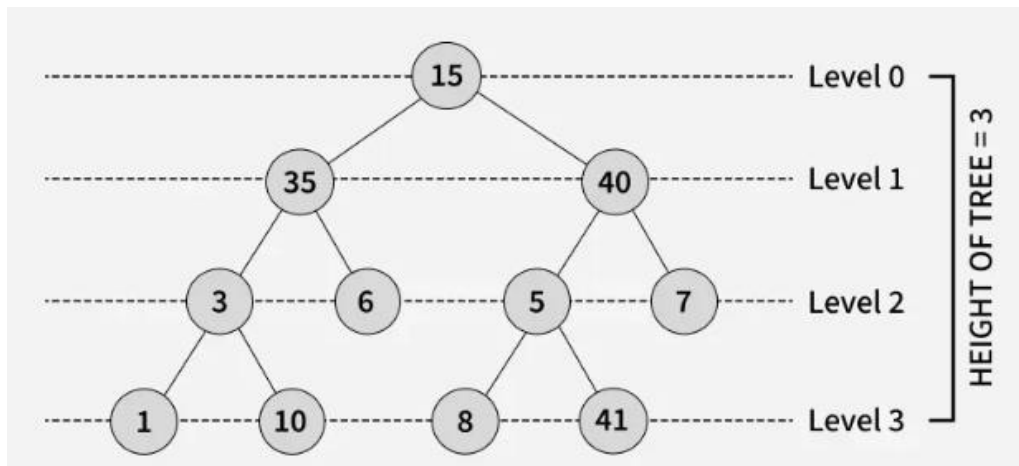


Terminologies in Binary Tree

- **Parent Node:** A node that is the **direct ancestor** of a node(its child node).
- **Child Node:** A node that is the **direct descendant** of another node (its parent).
- **Ancestors of a node:** All nodes on the path from the root to that node (including the node itself).
- **Descendants of a node:** All nodes that lie in the subtree rooted at that node (including the node itself).
- **Subtree of a node:** A tree consisting of that node as root and all its descendants.
- **Edge:** The link/connection between a parent node and its child node.
- **Path in a binary tree:** A sequence of nodes connected by edges from one node to another.
- **Leaf Node:** A node that does not have any children or both children are null.
- **Internal Node:** A node that has at least one child.

- **Depth/Level of a Node:** The number of edges in the path from root to that node. The depth/level of the **root** node is zero.
- **Height of a Binary Tree:** The number of edges on the longest path from root to a leaf. The diagram below shows all these terms in a binary tree.





Create/Declare a Node of a Binary Tree

Syntax to declare a Node of Binary Tree in different languages:

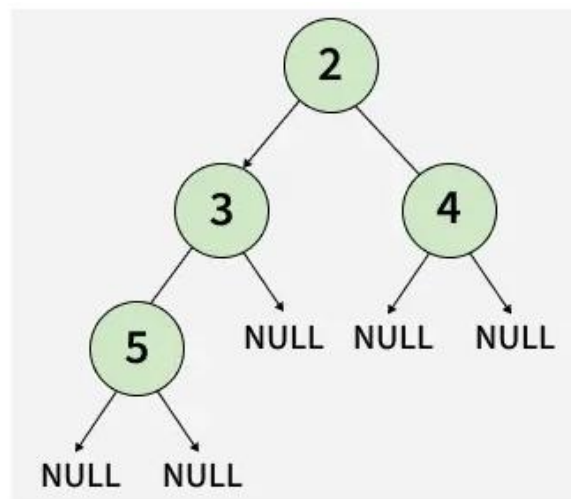
// Node structure

```

class Node {
    int key;
    Node left, right;
    public Node(int item)
    {
        key = item;
        left = right = null;
    }
}
  
```

Creating a Binary Tree:

In this example, we will explore how to create a Binary Tree with four nodes 2, 3, 4, 5.



```
// Node structure
class Node {
    int data;
    Node left, right;
    Node(int d) {
        data = d;
        left = null;
        right = null;
    }
}

class GfG {
    public static void main(String[] args) {
        // Initialize and allocate memory for tree nodes
        Node firstNode = new Node(2);
        Node secondNode = new Node(3);
        Node thirdNode = new Node(4);
        Node fourthNode = new Node(5);

        // Connect binary tree nodes
        firstNode.left = secondNode;
        firstNode.right = thirdNode;
        secondNode.left = fourthNode;
    }
}
```

In the above code, we have created four tree nodes **firstNode**, **secondNode**, **thirdNode** and **fourthNode** with values **2**, **3**, **4** and **5** respectively, and then established links between those nodes wrt the given tree structure.

Properties of Binary Tree

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

Please refer [Properties of Binary Tree](#) for more details.

Operations On Binary Tree

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

- 1. Traversal:** Depth-First Search (DFS) Traversal and Breadth-First Search (BFS) Traversal
- 2. Search:** Search a node in Binary Tree
- 3. Insertion and Deletion:** Prerequisite: Level Order Traversal, Insert in a Binary Tree and Delete from a Binary Tree

Advantages of Binary Tree

- **Efficient Search:** Binary Search Trees (a variation of Binary Tree) are efficient when searching for a specific element, as each node has at most two child nodes when compared to linked list and arrays

- **Memory Efficient:** Binary trees require lesser memory as compared to other tree data structures, therefore memory-efficient.
- Binary trees are relatively easy to implement and understand as each node has at most two children, left child and right child.

Disadvantages of Binary Tree

- **Limited structure:** Binary trees are limited to two child nodes per node, which can limit their usefulness in certain applications. For example, if a tree requires more than two child nodes per node, a different tree structure may be more suitable.
- **Space inefficiency:** Binary trees can be space inefficient when compared to other data structures like arrays and linked list. This is because each node requires two child references or pointers, which can be a significant amount of memory overhead for large trees.

Applications of Binary Tree

- Binary Tree can be used to represent hierarchical data.
- Huffman Coding trees are used in data compression algorithms.
- Useful for indexing segmented at the database is useful in storing cache in the system,
- Binary trees can be used to implement decision trees, a type of machine learning algorithm used for classification and regression analysis.

Binary Tree Traversal:

Binary trees are fundamental data structures in computer science and understanding their traversal is crucial for various applications. Traversing a binary tree means visiting all the nodes in a specific order. There are several traversal methods, each with its unique applications and benefits. This article will explore the main types of binary tree traversal: **in-order, pre-order, post-order, and level-order**.

Types of Binary Tree Traversal:

- 1. In-Order Binary Tree Traversal
- 2. Pre-Order Binary Tree Traversal
- 3. Post-Order Binary Tree Traversal
- 4. Level-Order Binary Tree Traversal

1. In-Order Binary Tree Traversal:

In in-order traversal, the left child is visited first, followed by the node itself, and then the right child. This can be visualized as **Left - Root - Right**.

2. Pre-Order Binary Tree Traversal:

In pre-order traversal, the node is visited first, followed by its left child and then its right child. This can be visualized as **Root - Left - Right**.

3. Post-Order Binary Tree Traversal:

In post-order traversal, the left child is visited first, then the right child, and finally the node itself. This can be visualized as **Left - Right - Root**.

4. Level-Order Binary Tree Traversal

In level-order traversal, the nodes are visited level by level, starting from the root node and then moving to the next level. This can be visualized as **Level 1 - Level 2 - Level 3 –**

Binary tree representation:

Binary tree is a **tree data structure (non-linear)** in which each node can have at most two children which are referred to as the **left** child and the **right** child. The topmost node in a binary tree is called the **root**, and the bottom-most nodes are called **leaves**. A binary tree can be visualized as a hierarchical structure with the root at the top and the leaves at the bottom. Binary trees can be represented in multiple ways, each with its own advantages, depending on the use case. Let's explore the two common methods: linked node representation and array implementation.

Representation of Binary Trees

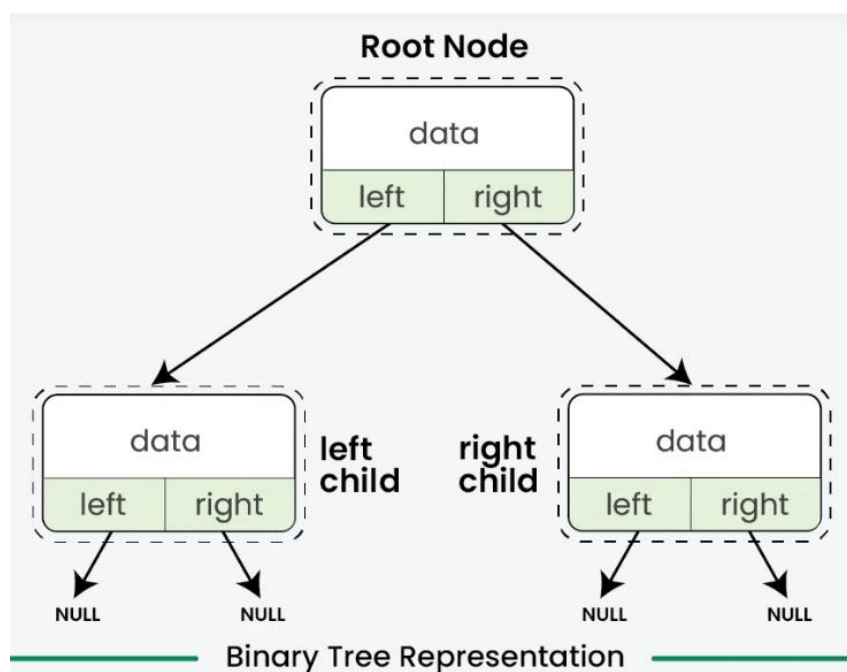
There are two primary ways to represent binary trees:

1. Linked Node Representation
2. Array Representation

1. Linked Node Representation:

This is the simplest way to represent a binary tree. Each node contains **data** and pointers to its **left** and **right** children.

This representation is mostly used to represent binary tree with multiple advantages. The most common advantages are given below.



Advantages:

- It can easily grow or shrink as needed, so it uses only the memory it needs.
- Adding or removing nodes is straightforward and requires only pointer adjustments.
- Only uses memory for the nodes that exist, making it efficient for sparse trees.

Disadvantages:

- Needs extra memory for pointers.
- Finding a node can take longer because you have to start from the root and follow pointers.

Create/Declare a Node of a Binary Tree:

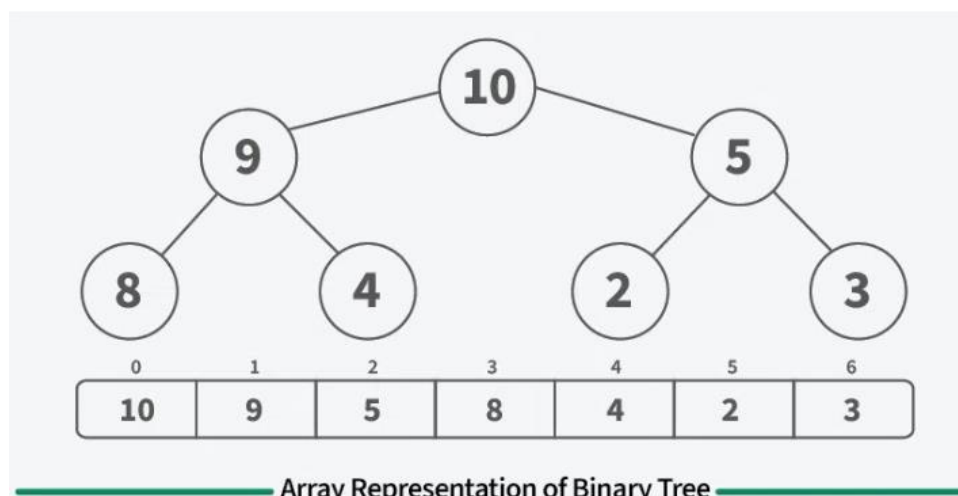
Syntax to declare a node of "linked list representation of binary tree" in different languages:

```
// Node structure of each node of the binary tree.  
class Node {  
    int data;  
    Node left, right;  
  
    Node(int val) {  
        data = val;  
        left = null;  
        right = null;  
    }  
}
```

2. Array Representation

Array Representation is another way to represent binary trees, especially useful when the tree is complete (all levels are fully filled except possibly the last, which is filled from left to right). In this method:

- The tree is stored in an array.
- For any node at index **i**:
 - Left Child: Located at $2 * i + 1$
 - Right Child: Located at $2 * i + 2$
- Root Node: Stored at index **0**



Advantages:

- Easy to navigate parent and child nodes using index calculations, which is fast

- Easier to implement, especially for complete binary trees.

Disadvantages:

- You have to set a size in advance, which can lead to wasted space.
- If the tree is not complete binary tree then many slots in the array might be empty, this will result in wasting memory
- Not as flexible as linked representations for dynamic trees.

Threaded binary tree :

A threaded binary tree is a type of binary tree data structure where the empty left and right child pointers in a binary tree are replaced with threads that link nodes directly to their in-order predecessor or successor, thereby providing a way to traverse the tree without using recursion or a stack.

Threaded binary trees can be useful when space is a concern, as they can eliminate the need for a stack during traversal. However, they can be more complex to implement than standard binary trees.

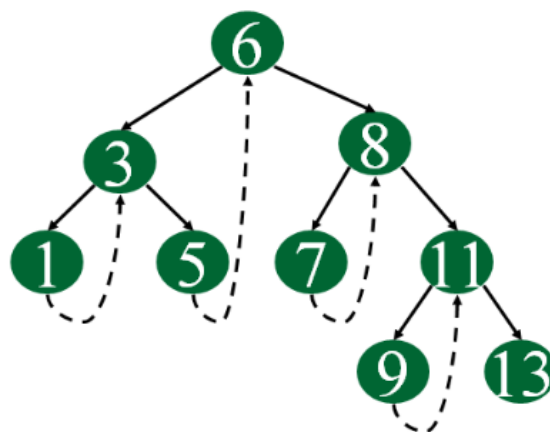
There are two types of threaded binary trees.

Single Threaded: Where a NULL right pointers is made to point to the inorder successor (if successor exists)

Double Threaded: Where both left and right NULL pointers are made to point to inorder predecessor and inorder successor respectively. The predecessor threads are useful for reverse inorder traversal and postorder traversal.

The threads are also useful for fast accessing ancestors of a node.

Following diagram shows an example Single Threaded Binary Tree. The dotted lines represent threads.



Inorder Traversal using Threads:

Following is code for inorder traversal in a threaded binary tree.

```

/ Utility function to find leftmost node in a tree rooted
// with n
Node leftMost(Node n)

```

```

{
    if (n == null)
        return null;

    while (n.left != null)
        n = n.left;

    return n;
}

```

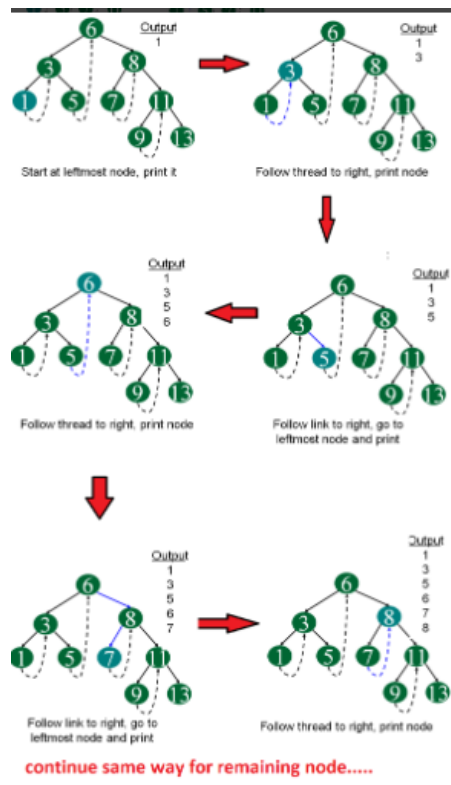
// C code to do inorder traversal in a threaded binary tree

```

static void inOrder(Node root)
{
    Node cur = leftMost(root);
    while (cur != null) {
        System.out.printf("%d ", cur.data);

        // If this node is a thread node, then go to
        // inorder successor
        if (cur.rightThread)
            cur = cur.right;
        else // Else go to the leftmost child in right
            // subtree
            cur = leftmost(cur.right);
    }
}

```



Advantages of Threaded Binary Tree

- In this Tree it enables linear traversal of elements.
- It eliminates the use of stack as it perform linear traversal, so save memory.
- Enables to find parent node without explicit use of parent pointer
- Threaded tree give forward and backward traversal of nodes by in-order fashion
- Nodes contain pointers to in-order predecessor and successor
- For a given node, we can easily find inorder predecessor and successor. So, searching is much more easier.
- In threaded binary tree there is no NULL pointer present. Hence memory wastage in occupying NULL links is avoided.
- The threads are pointing to successor and predecessor nodes. This makes us to obtain predecessor and successor node of any node quickly.
- There is no need of stack while traversing the tree, because using thread links we can reach to previously visited nodes.

Disadvantages of Threaded Binary Tree

- Every node in threaded binary tree need extra information(extra memory) to indicate whether its left or right node indicated its child nodes or its inorder predecessor or successor. So, the node consumes extra memory to implement.
- Insertion and deletion are way more complex and time consuming than the normal one since both threads and ordinary links need to be maintained.
- Implementing threads for every possible node is complicated.
- **Increased complexity:** Implementing a threaded binary tree requires more complex algorithms and data structures than a regular binary tree. This can make the code harder to read and debug.
- **Extra memory usage:** In some cases, the additional pointers used to thread the tree can use up more memory than a regular binary tree. This is especially true if the tree is not fully balanced, as threading a skewed tree can result in a large number of additional pointers.
- **Limited flexibility:** Threaded binary trees are specialized data structures that are optimized for specific types of traversal. While they can be more efficient than regular binary trees for these types of operations, they may not be as useful in other scenarios. For example, they cannot be easily modified (e.g. inserting or deleting nodes) without breaking the threading.
- **Difficulty in parallelizing:** It can be challenging to parallelize operations on a threaded binary tree, as the threading can introduce data dependencies that make it difficult to process nodes independently. This can limit the performance gains that can be achieved through parallelism.

Applications of threaded binary tree :

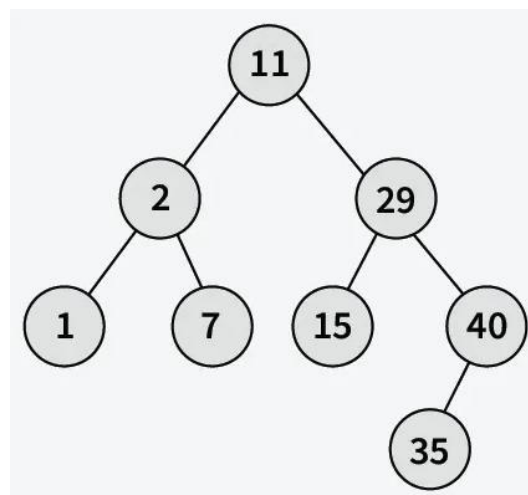
- **Expression evaluation:** Threaded binary trees can be used to evaluate arithmetic expressions in a way that avoids recursion or a stack. The tree can be constructed from the input expression, and then traversed in-order or pre-order to perform the evaluation.
- **Database indexing:** In a database, threaded binary trees can be used to index data based on a specific field (e.g. last name). The tree can be constructed with the indexed values as keys, and then traversed in-order to retrieve the data in sorted order.
- **Symbol table management:** In a compiler or interpreter, threaded binary trees can be used to store and manage symbol tables for variables and functions. The tree can be constructed with the symbols as keys, and then traversed in-order or pre-order to perform various operations on the symbol table.
- **Disk-based data structures:** Threaded binary trees can be used in disk-based data structures (e.g. B-trees) to improve performance. By threading the tree, it can be traversed in a way that minimizes disk seeks and improves locality of reference.
- **Navigation of hierarchical data:** In certain applications, threaded binary trees can be used to navigate hierarchical data structures, such as file systems or web site directories. The tree can be constructed from the hierarchical data, and then traversed in-order or pre-order to efficiently access the data in a specific order.

Binary Search Tree (BST) :

A **Binary Search Tree (BST)** is a special type of binary tree that maintains its elements in a sorted order. It is a non-linear, hierarchical data structure, where each node can have at most two children, and elements are organized in a parent-child relationship. For every node in the BST:

- All nodes in its left subtree have values less than the node's value.
- All nodes in its right subtree have values greater than the node's value.

This property ensures that each comparison allows the operation to skip about half of the remaining tree, making BST operations much faster than linear structures like arrays or linked lists.



Key Properties

- Unique ordering of elements means duplicates are usually not allowed.
- Inorder traversal of a BST gives sorted order of elements.
- Average height: $O(\log n)$ (for balanced BST).
- Worst case height: $O(n)$ (when tree becomes skewed).

Operations in BST

- **Search** : Finds whether a given key exists in the BST. Time Complexity on average is $O(\log n)$ and worst case is $O(n)$
- **Insertion** : Insert a new node while maintaining BST property. Compare key with current node and move left/right recursively or iteratively. Time Complexity on average is $O(\log n)$ and worst case is $O(n)$
- **Deletion** : Remove a node while keeping BST valid. Node has no children means remove directly. Node has one child means replace node with its child. Node has two children means replace node with inorder successor/predecessor and delete that successor/predecessor. Time Complexity: average $O(\log n)$ and $O(n)$ worst case
- **Traversals** : The four common tree traversals are Inorder (Left, Root, Right) which gives nodes in sorted order for a BST, Preorder (Root, Left, Right), Postorder (Left, Right, Root), and Level-order, which traverses the tree level by level using a queue.

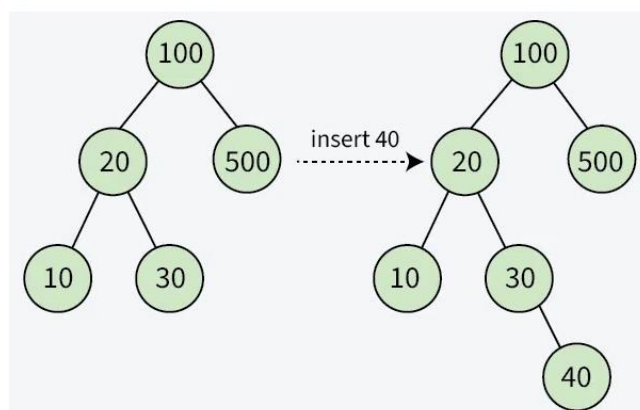
Application of Binary Search Tree

- Searching and indexing (e.g., maps, sets).
- Dynamic sorting and range queries.
- Implementing symbol tables in compilers.
- Used in advanced structures (AVL Tree, Red-Black Tree, Splay Tree).

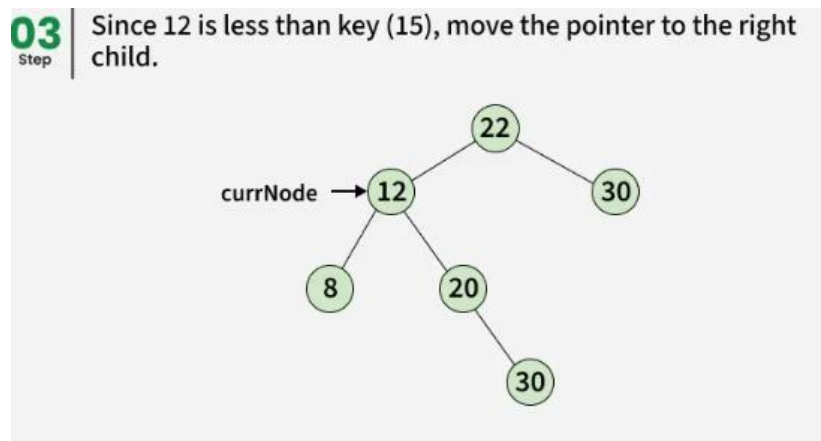
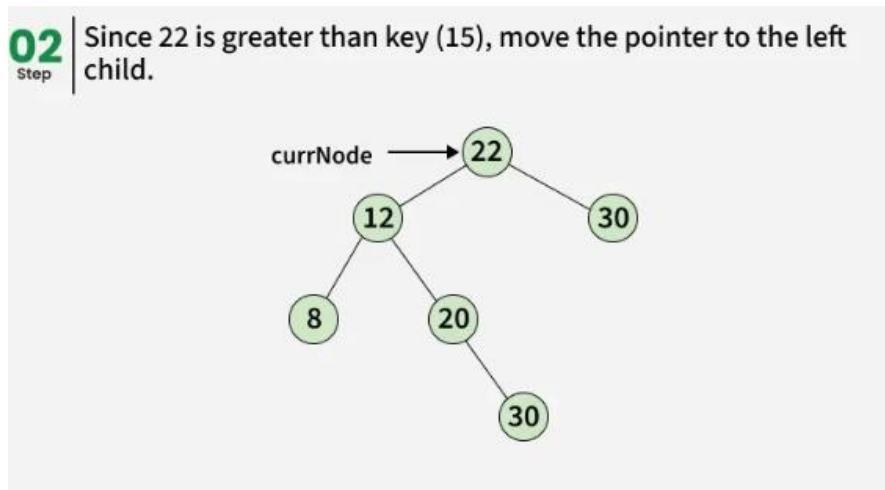
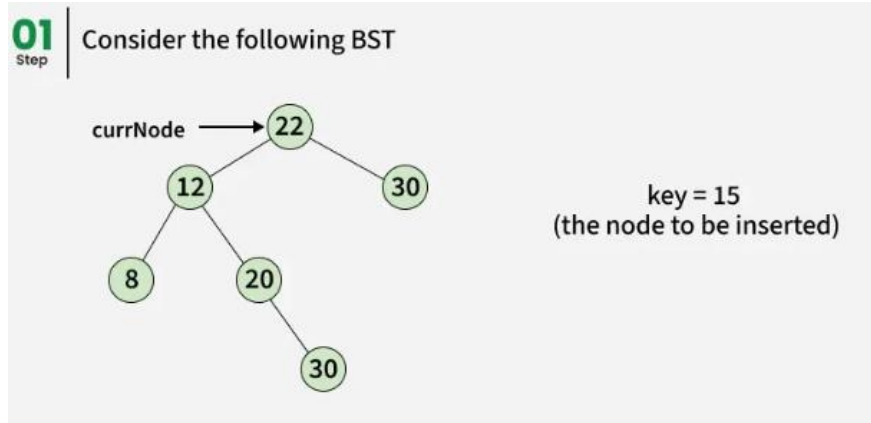
Insertion in Binary Search Tree (BST):

Given the **root** of a Binary Search Tree, we need to insert a new node with given value in the **BST**. All the nodes have distinct values in the BST and we may assume that the new value to be inserted is not present in BST.

example:

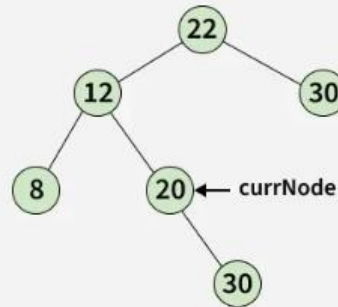


A new key is inserted at the position that maintains the BST property. We start from the root and move downward: if the key is smaller, go left; if larger, go right. We continue until we find an unoccupied spot where the node can be placed without violating the BST property, and insert it there as a new leaf.



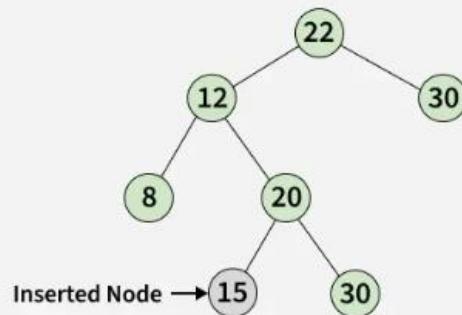
04
Step

Again 20 is greater than key (15), move the pointer to the left child.



05
Step

Now, the pointer refers to NULL. Hence, insert key(15) at this position.



Time Complexity: $O(h)$

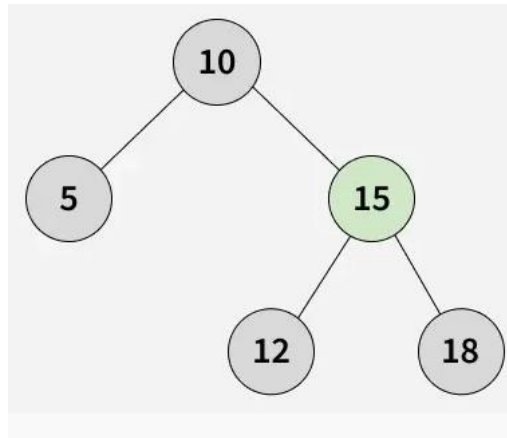
- The worst-case time complexity of insert operations is $O(h)$ where h is the height of the Binary Search Tree.
- In the worst case, we may have to travel from the root to the deepest leaf node. The height of a skewed tree may become n and the time complexity of insertion operation may become $O(n)$.

Auxiliary Space: $O(h)$, due to recursive stack.

Deletion in Binary Search Tree (BST):

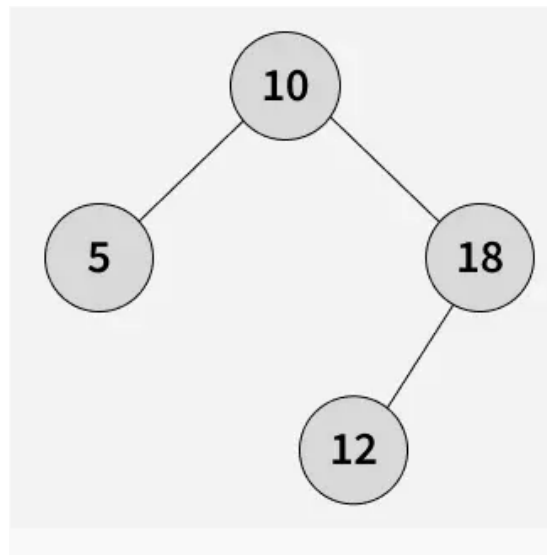
Given the **root** of a Binary Search Tree (BST) and an integer x , delete the node with value x from the BST while maintaining the BST property.

Input: $x = 15$



Output: [[10], [5, 18], [N, N, 12, N]]

Explanation: The node with value x (15) is deleted from BST.

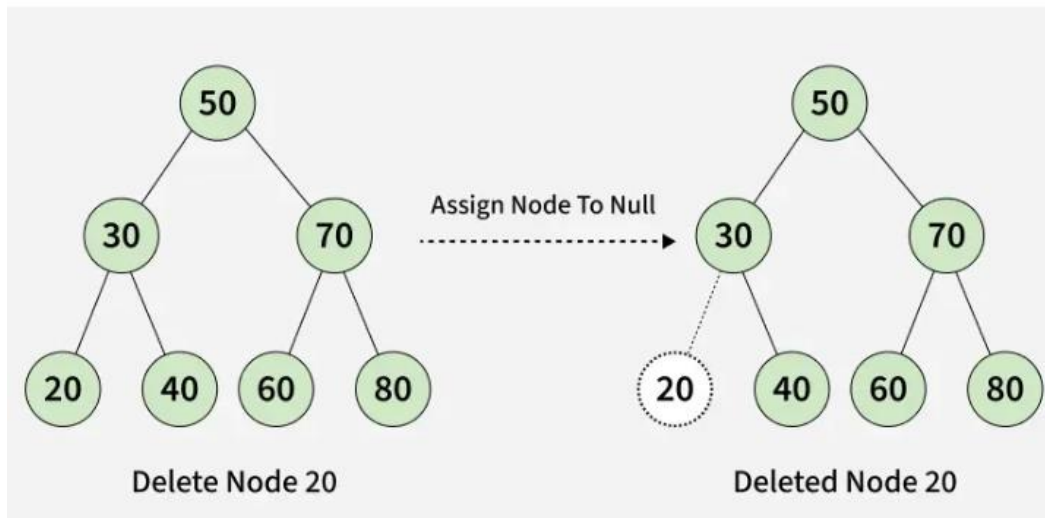


Deleting a node in a BST means removing the target node while ensuring that the tree remains a valid BST. Depending on the structure of the node to be deleted, there are three possible scenarios:

Case 1: Node has No Children (Leaf Node)

If the target node is a leaf node, it can be directly removed from the tree since it has no child to maintain.

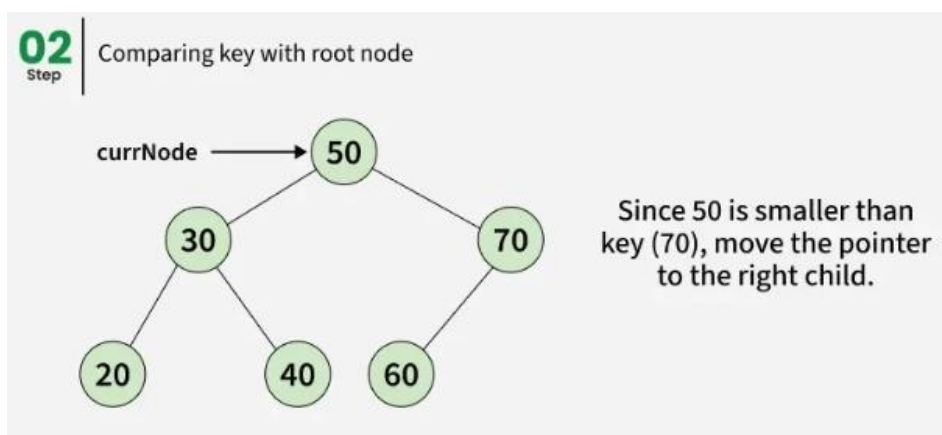
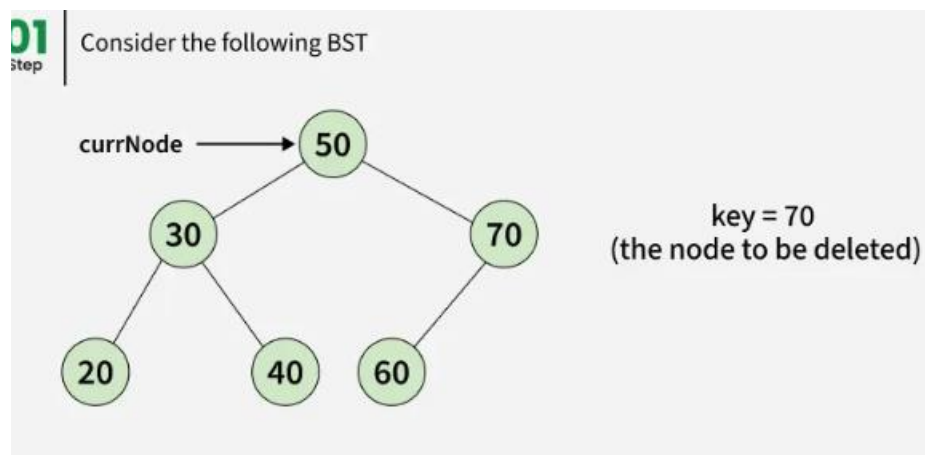
Working:



Case 2: Node has One Child(Left or Right Child)

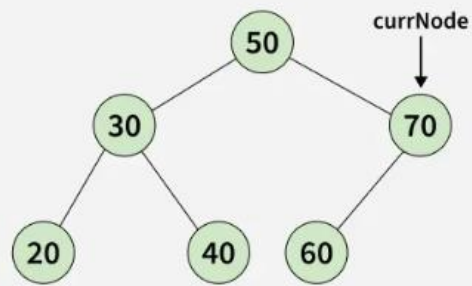
If the target node has only one child, we remove the node and connect its parent directly to its only child. This way, the tree remains valid after deletion of target node.

Working:



03
Step

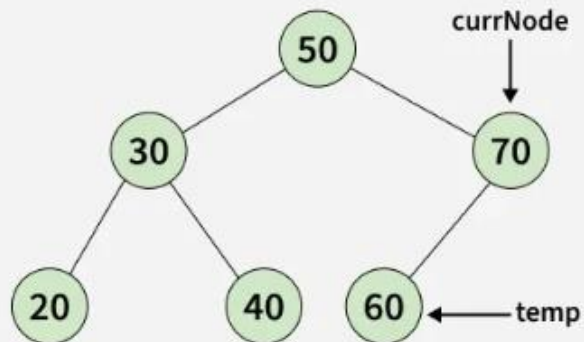
Comparing key with right child root node



Found the key node(70)

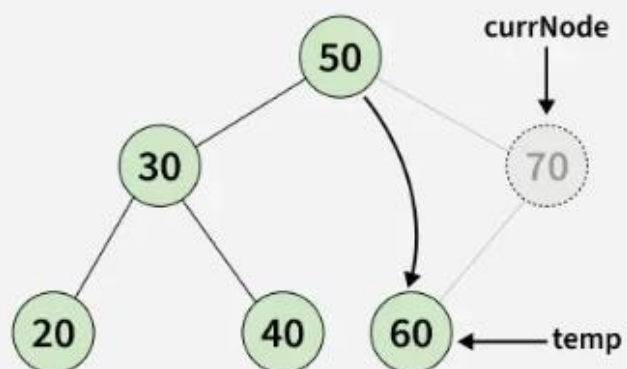
04
Step

Node 70 have only one child (temp=currNode->left).



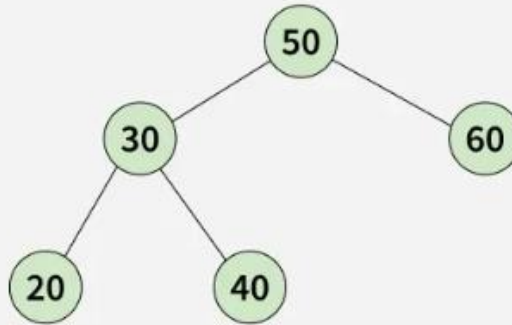
05
Step

Delete the node and return temp.



06
Step

Final BST



Case 3: Node has Two Children

If the target node has two children, deletion is slightly more complex.

To maintain the BST property, we need to find a replacement node for the target. The replacement can be either:

- The inorder successor — the smallest value in the right subtree, which is the next greater value than the target node.
- The inorder predecessor — the largest value in the left subtree, which is the next smaller value than the target node.

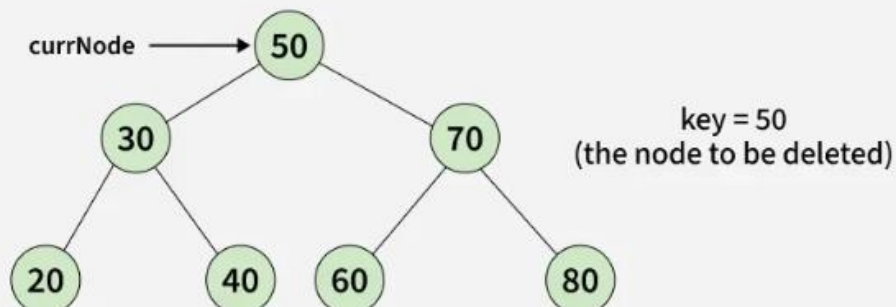
Once the replacement node is chosen, we replace the target node's value with that node's value, and then delete the replacement node, which will now fall under Case 1 (no children) or Case 2 (one child).

Note: Inorder predecessor can also be used.

Working:

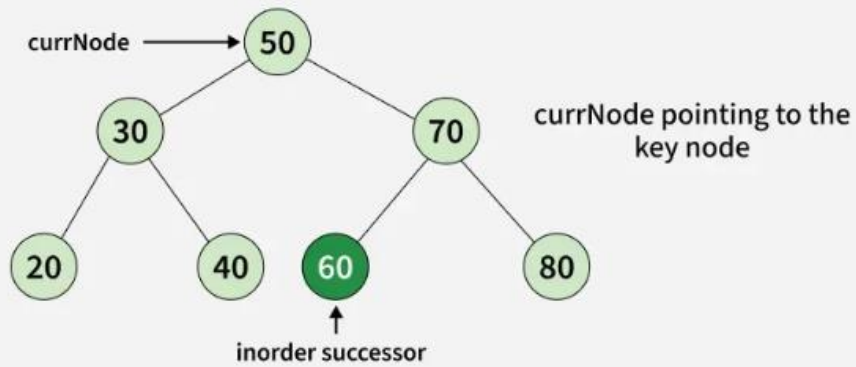
01
Step

Consider the following BST



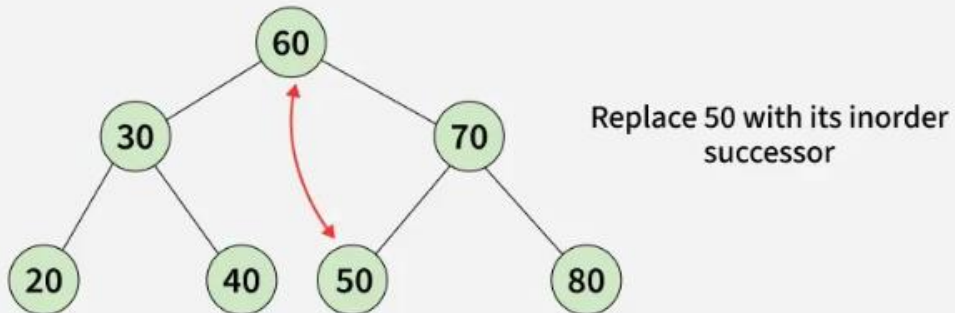
02
Step

Find the inorder successor



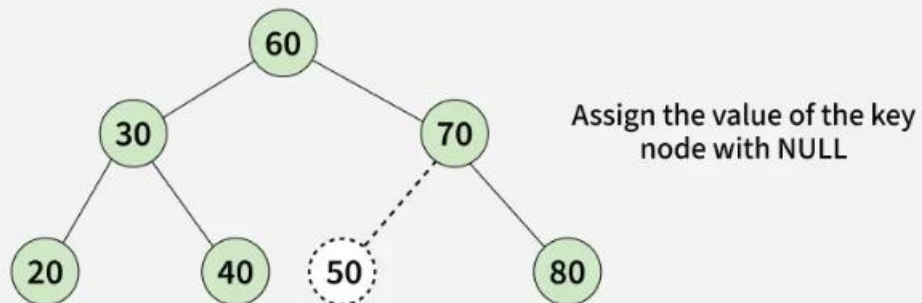
03
Step

Swapping the key node with its inorder successor



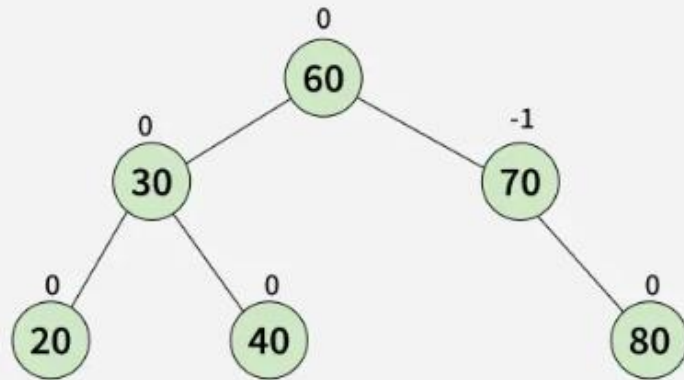
04
Step

Deletion of the key node (50)



05
Step

Final BST



The deletion process in BST depends on the number of children of the node.

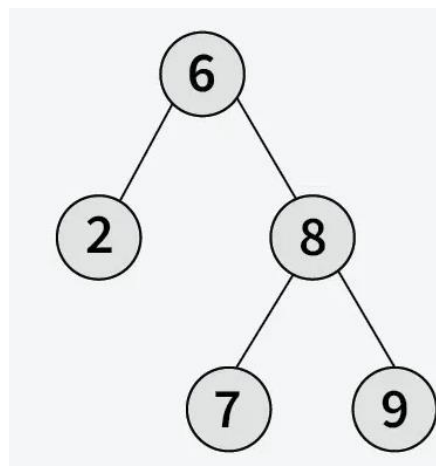
- No children means simply remove the node.
- One child means remove the node and connect its parent to the node's only child.
- Two children means replace the node with its inorder successor/predecessor and delete that node.

Searching in Binary Search Tree (BST):

Given the **root** of a Binary Search Tree and a value **key**, find if key is present in the BST or not.

Note: The key may or may not be present in the BST.

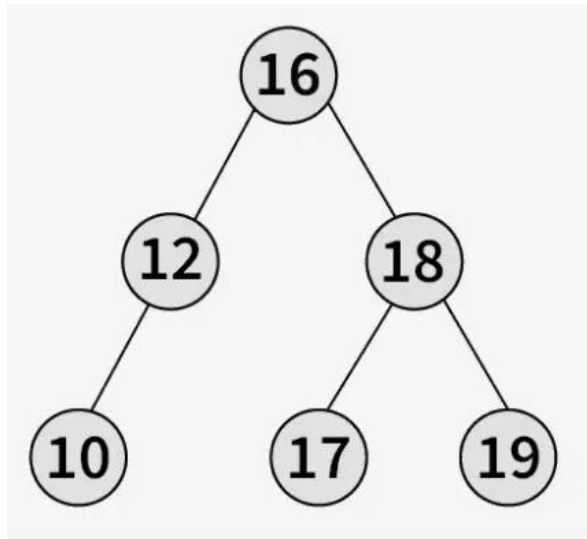
Input: key = 7



Output: true

Explanation: 7 is present in the BST.

Input: key = 14



Output: false

Explanation: 14 is not present in the BST.

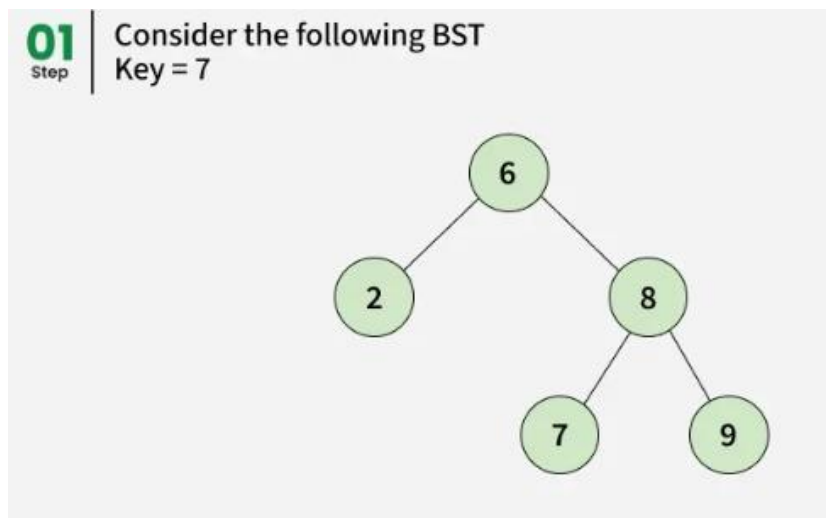
How to search a value in Binary Search Tree:

Let's say we want to search for the number **key**, We start at the root. Then:

- We compare the value to be searched with the value of the root.
- If it's equal we are done with the search.
- If it's smaller we know that we need to go to the left subtree.
- If it's greater we search in the right subtree.
- Repeat the above step till no more traversal is possible
- If at any iteration, key is found, return True. If the node is null, return False.

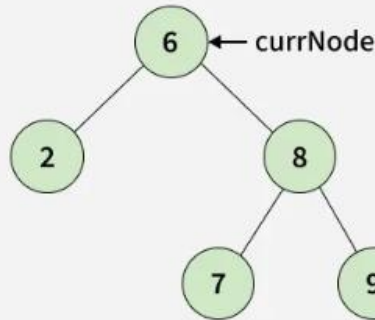
Illustration of searching a key in BST:

See the illustration below for a better understanding:



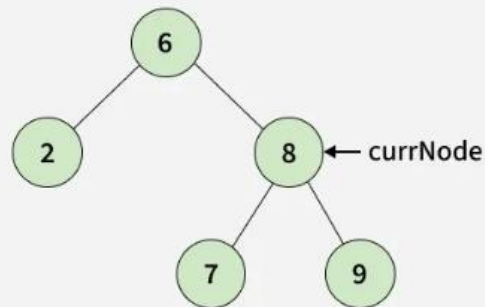
02
Step

Compare key with current node, i.e., 6.
As $6 < 7$, search in the right subtree of 6.



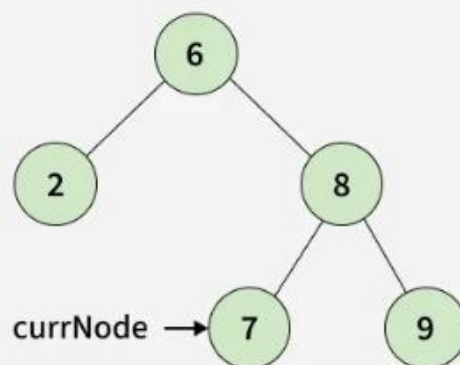
03
Step

Compare key with current node, i.e., 8.
As $8 > 7$, search in the left subtree of 8.



04
Step

Compare key with current node, i.e., 7.
As 7 is equal to key(7), we have found the key.



Time complexity: $O(h)$, where h is the height of the BST.

Auxiliary Space: $O(h)$ This is because of the space needed to store the recursion stack.

We can avoid the auxiliary space and recursion overhead with the help of iterative implementation. Below is the iterative implementation that works in $O(h)$ time and $O(1)$ auxiliary space.

AVL tree:

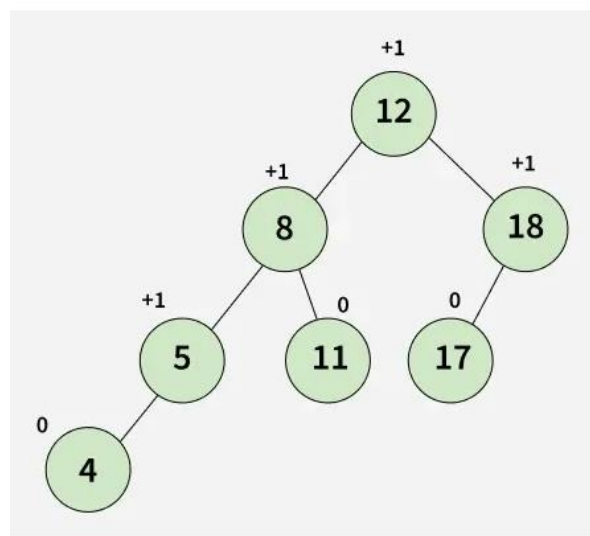
An AVL tree defined as a self-balancing Binary Search Tree (BST) where the difference between heights of left and right subtrees for any node cannot be more than one.

Balance Factor = left subtree height - right subtree height

For a Balanced Tree(for every node): $-1 \leq \text{Balance Factor} \leq 1$

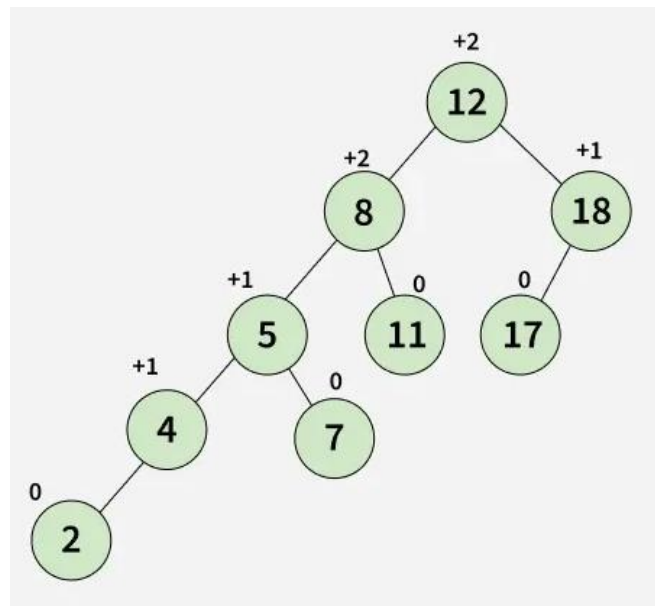
Example of an AVL Tree:

The balance factors for different nodes are: 12 : +1, 8 : +1, 18 : +1, 5 : +1, 11 : 0, 17 : 0 and 4 : 0. Since all differences are lies between -1 to +1, so the tree is an AVL tree.



Example of a BST which is not an AVL Tree:

The Below Tree is **not** an AVL Tree as the balance factor for nodes **8** and **12** is more than 1.



Important Points about AVL Tree:

- **Rotations:** rotations are designed to restore balance in **$O(1)$** time while ensuring the overall time complexity remains **$O(\log n)$** . AVL Trees use **four cases** to rebalance themselves after insertions and deletions: Left-Left (LL), Right-Right (RR), Left-Right (LR) and Right-Left (RL)
- **Insertion and Deletion:** While insertion is followed by upward traversals to check balance and apply rotations, deletion can be more complex due to multiple rotations possibly being required. AVL Trees may require multiple rebalancing steps during deletion, unlike Red-Black Trees which limit this better.
- **Use Cases:** AVL Trees are particularly useful when you need frequent and efficient lookups, like in database indexing, memory-intensive applications, or where predictable time complexity is crucial.
- **Drawbacks Compared to Other Trees:** Although faster in lookups than Red-Black Trees, AVL Trees might incur slightly more overhead on insertions and deletions due to stricter balancing requirements. As a result, Red-Black Trees are more common in standard libraries like TreeMap or TreeSet in Java or map in C++ STL.
- **In-order Traversal:** An in-order traversal of an AVL Tree still gives you elements in sorted order, just like any Binary Search Tree.

Operations on an AVL Tree:

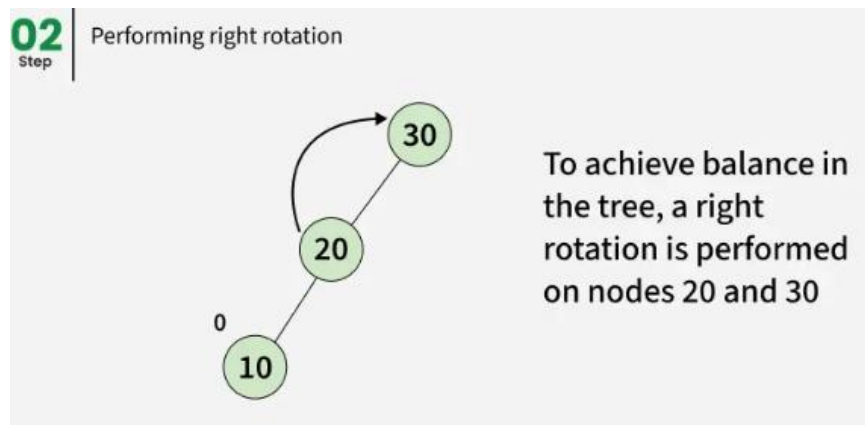
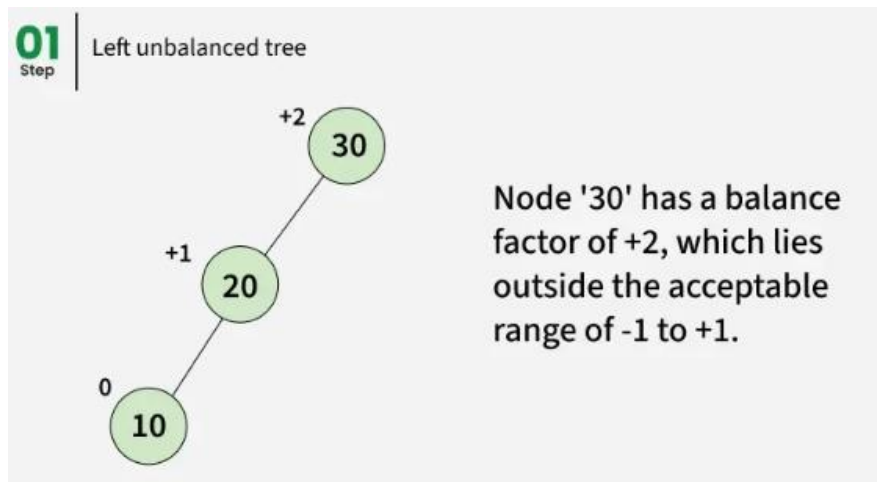
- **Searching :** It is same as normal Binary Search Tree (BST) as an AVL Tree is always a BST. So we can use the same implementation as BST. The advantage here is time complexity is $O(\log n)$
- **Insertion :** It does rotations along with normal BST insertion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after insertion
- **Deletion :** It also does rotations along with normal BST deletion to make sure that the balance factor of the impacted nodes is less than or equal to 1 after deletion.

Rotating the subtrees (Used in Insertion and Deletion)

An AVL tree may rotate in one of the following four ways to keep itself balanced while making sure that the BST properties are maintained.

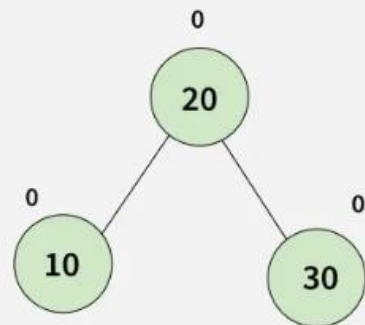
1. Left-Left Case:

- Occurs when a node is inserted into the left subtree of the left child, causing the balance factor to become **more than +1**.
- Fix:** Perform a single **right** rotation.



03
Step

Balanced tree



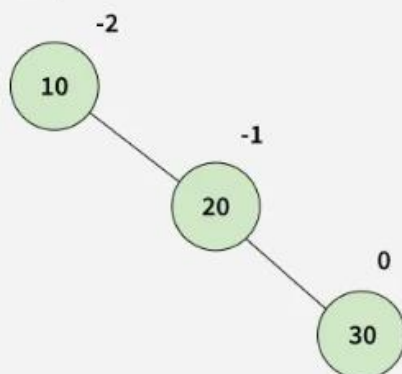
The tree is balanced now, with all node balance factors within the valid range.

2. Right-Right Case:

- Occurs when a node is inserted into the right subtree of the right child, making the balance factor **less than -1**.
- Fix:** Perform a single **left** rotation.

01
Step

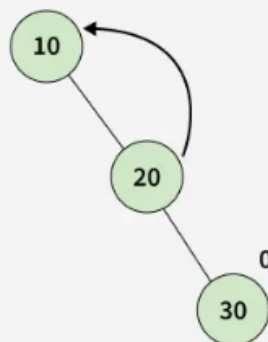
Right unbalanced tree



Node '10' has a balance factor of -2, which lies outside the acceptable range of -1 to +1.

02
Step

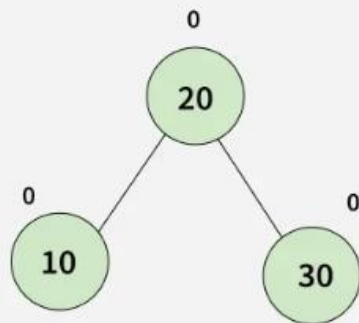
Performing left rotation



To achieve balance in the tree, a left rotation is performed on nodes 10 and 20

03
Step

Balanced tree



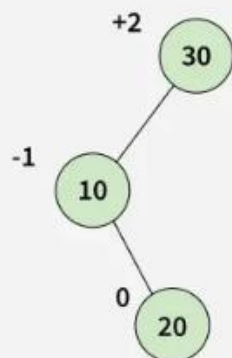
The tree is balanced now, with all node balance factors within the valid range.

3. Left-Right Case:

- **Occurs when** a node is inserted into the right subtree of the left child, which disturbs the balance factor of an ancestor node, making it **left-heavy**.
- **Fix:** Perform a left rotation on the left child, followed by a right rotation on the node.

01
Step

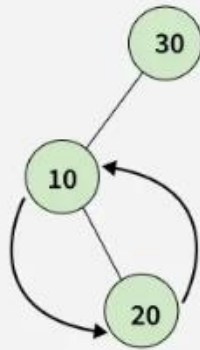
Unbalanced tree



Node '30' has a balance factor of +2, which lies outside the acceptable range of -1 to +1.

02
Step

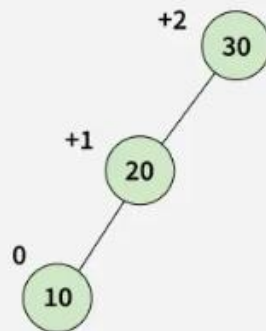
Performing left rotation



To achieve balance in the tree, a left rotation is performed on nodes 10 and 20

03
Step

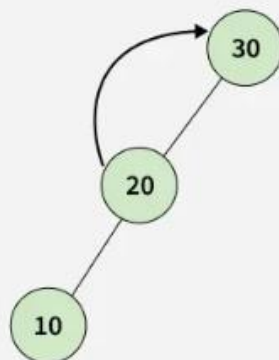
Unbalanced tree



we need to perform right rotation on nodes 30 and 20 to balanced the tree

04
Step

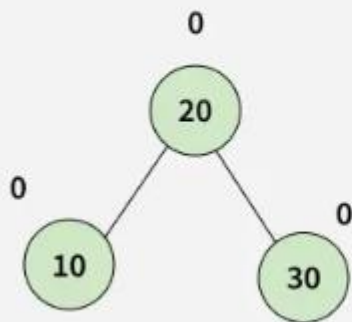
Performing right rotation



To achieve balance in the tree, a right rotation is performed on nodes 20 and 30

05
Step

Balanced tree



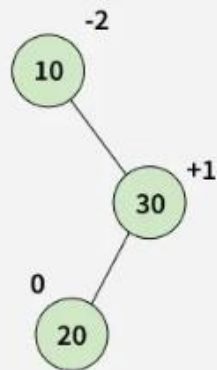
The tree is balanced now, with all node balance factors within the valid range.

4. Right-Left Case:

- Occurs when a node is inserted into the left subtree of the right child, which disturbs the balance factor of an ancestor node, making it **right-heavy**.
- Fix:** Perform a right rotation on the right child, followed by a left rotation on the node.

01
Step

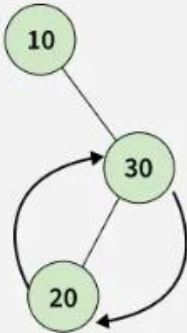
Unbalanced tree



Node '10' has a balance factor of -2, which lies outside the acceptable range of -1 to +1.

02
Step

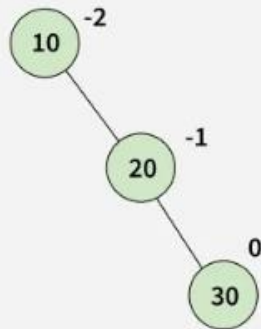
Performing right rotation



To achieve balance in the tree, a right rotation is performed on nodes 30 and 20

03
Step

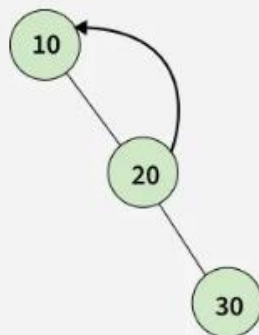
Unbalanced tree



we need to perform left rotation on nodes 10 and 20 to balanced the tree

04
Step

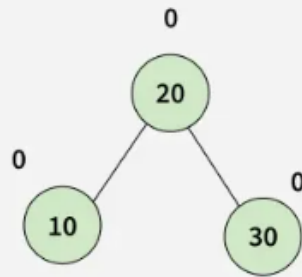
Performing left rotation



To achieve balance in the tree, a left rotation is performed on nodes 10 and 20

05
Step

Balanced tree



The tree is balanced now, with all node balance factors within the valid range.

Applications of AVL Tree:

1. AVL Tree is used as a first example self balancing BST in teaching DSA as it is easier to understand and implement compared to Red Black
2. Applications, where insertions and deletions are less common but frequent data lookups along with other operations of BST like sorted traversal, floor, ceil, min and max.
3. Red Black tree is more commonly implemented in language libraries like map in C++, set in C++, TreeMap in Java and TreeSet in Java.
4. AVL Trees can be used in a real time environment where predictable and consistent performance is required.

Advantages of AVL Tree:

1. AVL trees can self-balance themselves and therefore provides time complexity as **$O(\log n)$** for **search, insert and delete**.
2. As it is a balanced BST, so items can be traversed in sorted order.
3. Since the balancing rules are strict compared to Red Black Tree, AVL trees in general have relatively less height and hence the search is faster.
4. AVL tree is relatively less complex to understand and implement compared to Red Black Trees.

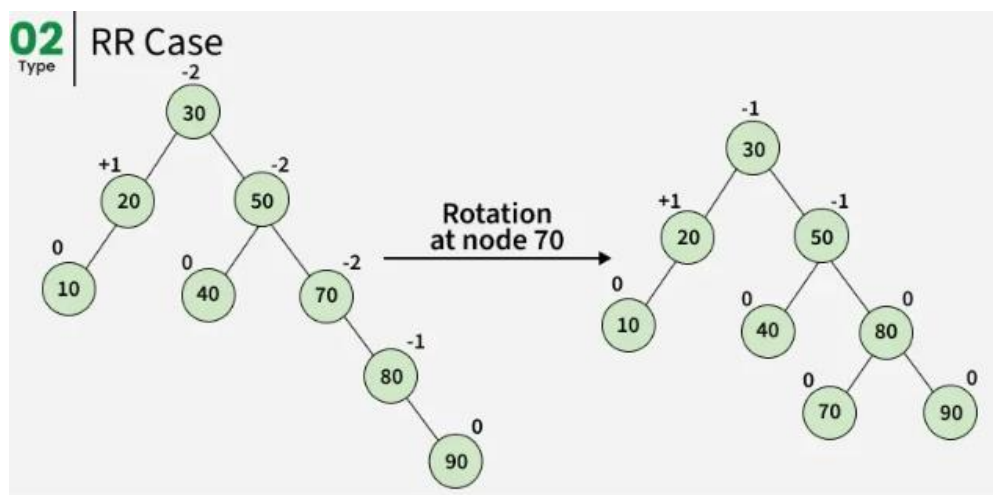
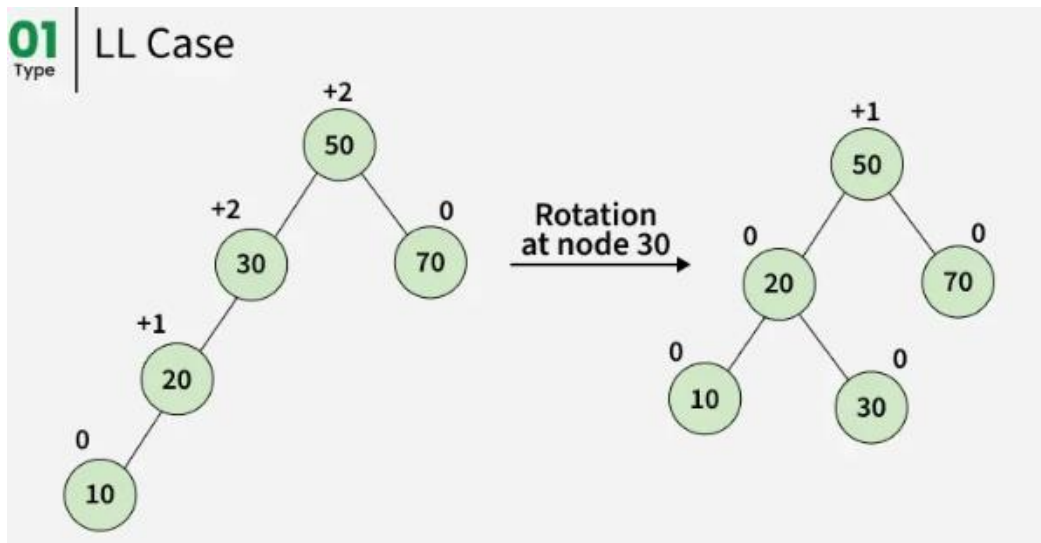
Disadvantages of AVL Tree:

1. It is difficult to implement compared to normal BST.
 2. Less used compared to Red-Black trees. Due to its rather strict balance.
 3. AVL trees provide complicated insertion and removal operations as more rotations are performed.
- Insertion in an AVL Tree
 - Deletion in an AVL Tree
 - Red Black Tree

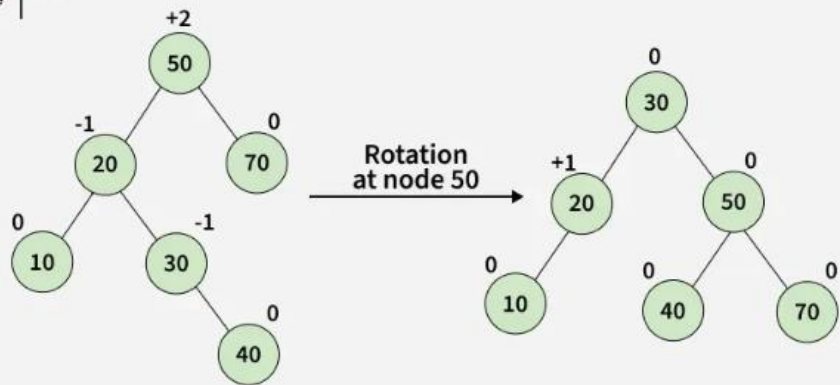
Insertion in an AVL Tree:

AVL tree is a self-balancing Binary Search Tree (**BST**) where the difference between heights of left and right subtrees cannot be more than **one** for all nodes. Insertion in an AVL Tree follows the same basic rules as in a Binary Search Tree (BST):

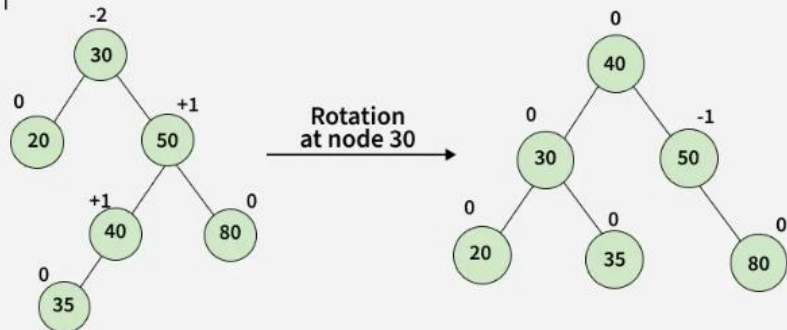
- A new key is placed in its correct position based on BST rules (left < node < right).
- After the insertion, the **balance factor** of each node is checked during the path back up to the root. If any node becomes **unbalanced** (i.e., its balance factor becomes less than -1 or greater than +1), a **rotation** is required to restore the AVL property.



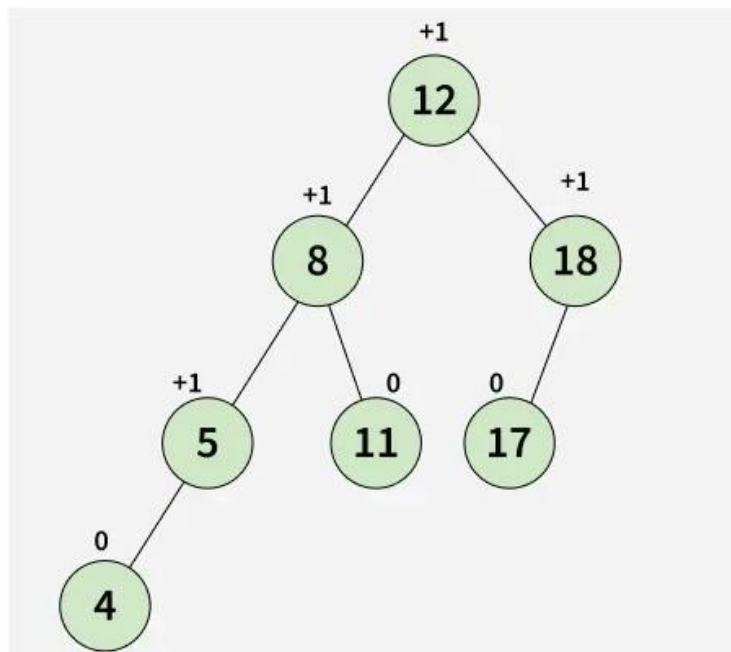
03 Type LR Case



04 Type RL Case

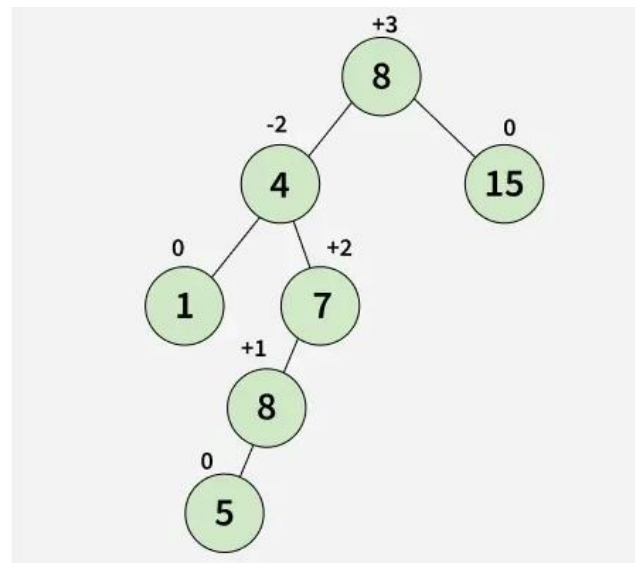


Example of AVL Tree:



The above tree is AVL tree because the differences between the heights of left and right subtrees for every node are lies in the range -1 to +1.

Example of a Tree that is NOT an AVL Tree:



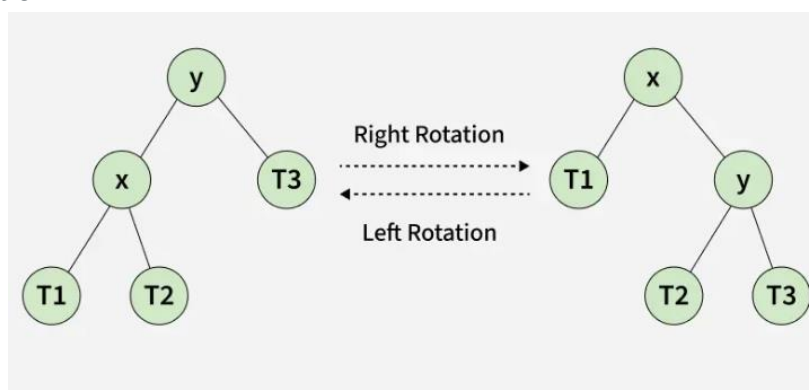
The above tree is not an AVL tree because the differences between the heights of the left and right subtrees for 8 and 12 are greater than 1.

Why AVL Trees? Most of the BST operations (e.g., search, max, min, insert, delete, floor and ceiling) take $O(h)$ time where h is the height of the BST. The cost of these operations may become $O(n)$ for a **skewed Binary tree**. If we make sure that the height of the tree remains $O(\log(n))$ after every insertion and deletion, then we can guarantee an upper bound of $O(\log(n))$ for all these operations. The height of an AVL tree is always $O(\log(n))$ where n is the number of nodes in the tree.

Insertion in AVL Tree:

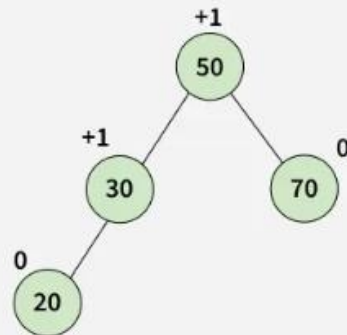
To make sure that the given tree remains AVL after every insertion, we must augment the standard BST insert operation to perform some re-balancing. Following are two basic operations that can be performed to balance a BST without violating the BST property ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

- Left Rotation
- Right Rotation



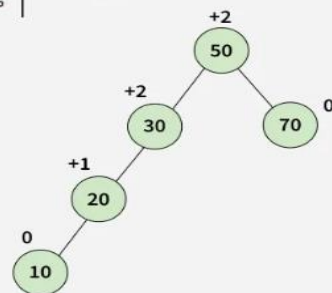
01
Step

Consider the following BST



02
Step

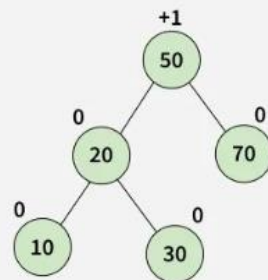
Inserting node 10



- Inserting node 10 at the left of node 20
- Balance factor (30) = +2
- LL Case required at node 30

03
Step

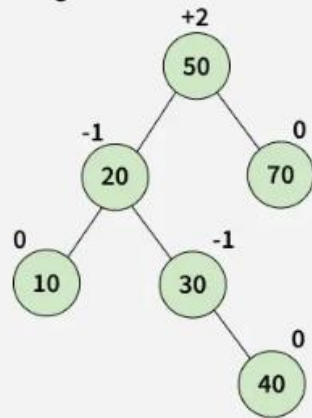
Performing LL Case



- Balance factor for every node is inside the range of -1 to +1

04
Step

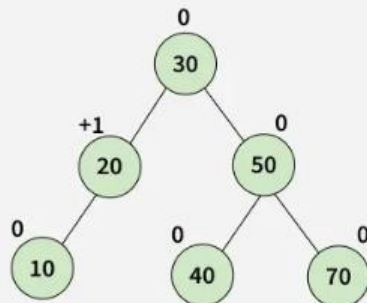
Inserting node 40



- Inserting node 40 at the right of node 30
- Balance factor (50) = +2
- LR Case required at node 50

05
Step

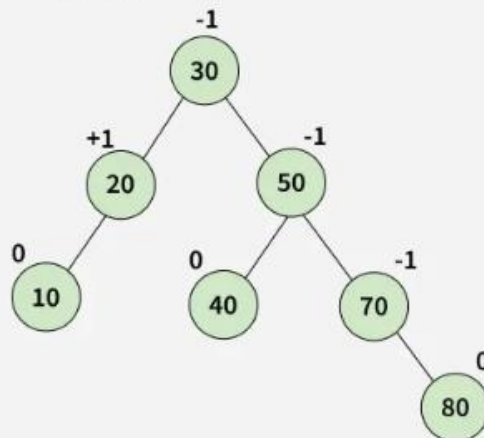
Performing LR Case



- Balance factor for every node is inside the range of -1 to +1

06
Step

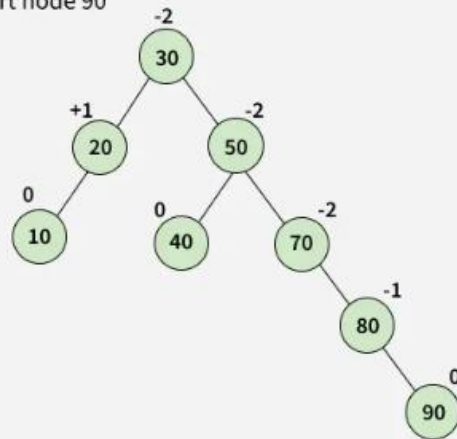
Insert node 80



- Inserting node 80 at the right of node 70

07
Step

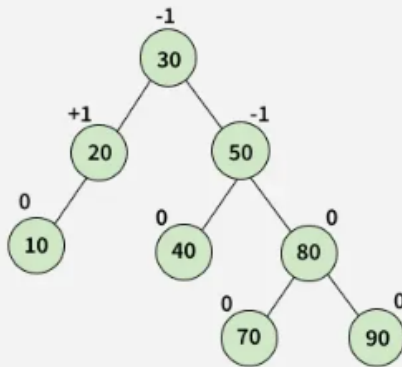
Insert node 90



- Inserting node 90 at the right of node 80
- Balance factor (70) = -2
- RR Case required at node 70

08
Step

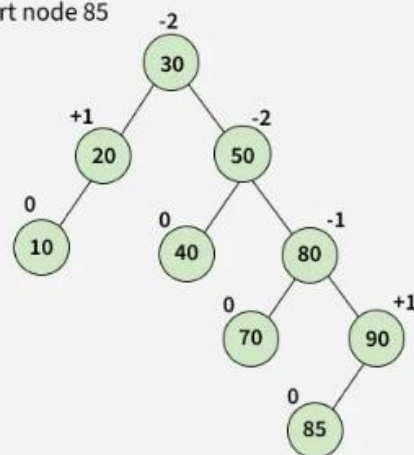
Performing RR Case



- Balance factor for every node is inside the range of -1 to +1

09
Step

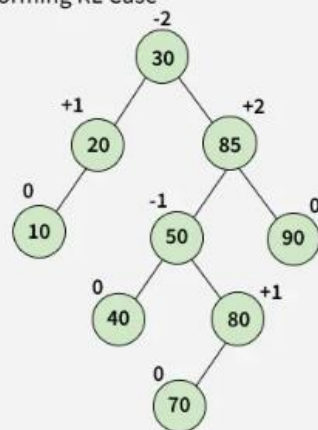
Insert node 85



- Inserting node 85 at the left of node 90
- Balance factor (50) = -2
- RL Case required at node 50

10
Step

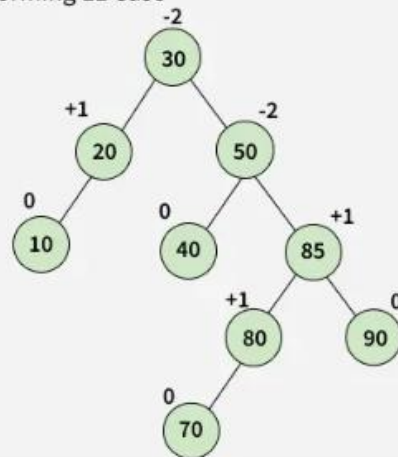
Performing RL Case



- Balance factor (85) = +2
- LL Case required at node 85

11
Step

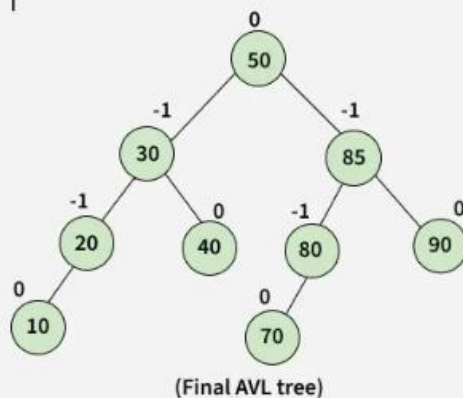
Performing LL Case



- Balance factor (50) = -2
- RR Case required at node 50

12
Step

Performing RR Case



- Balance factor for every node is inside the range of -1 to +1

Approach: The idea is to use recursive BST insert, after insertion, we get pointers to all ancestors one by one in a bottom-up manner. So we don't need a parent pointer to travel up. The recursive code itself travels up and visits all the ancestors of the newly inserted node. Follow the steps mentioned below to implement the idea:

- Perform the normal BST insertion.
- The current node must be one of the ancestors of the newly inserted node. Update the **height** of the current node.

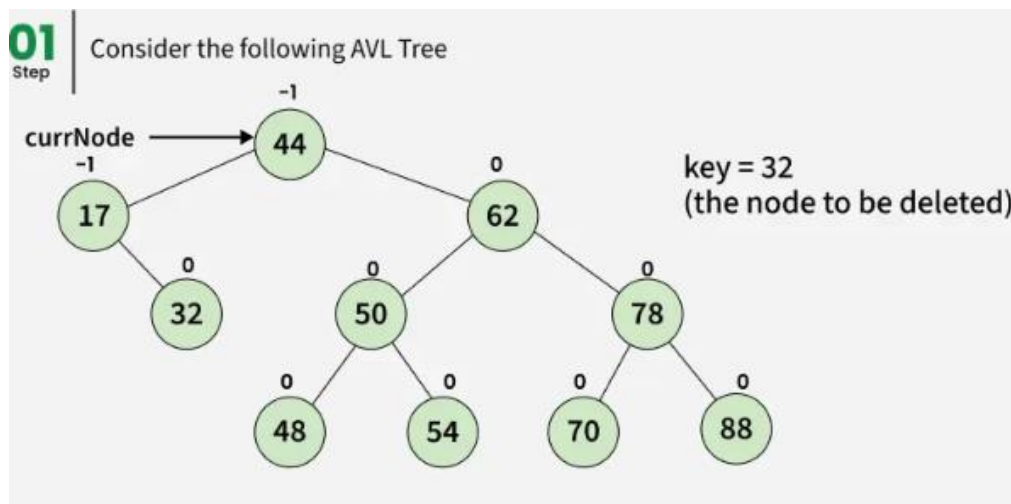
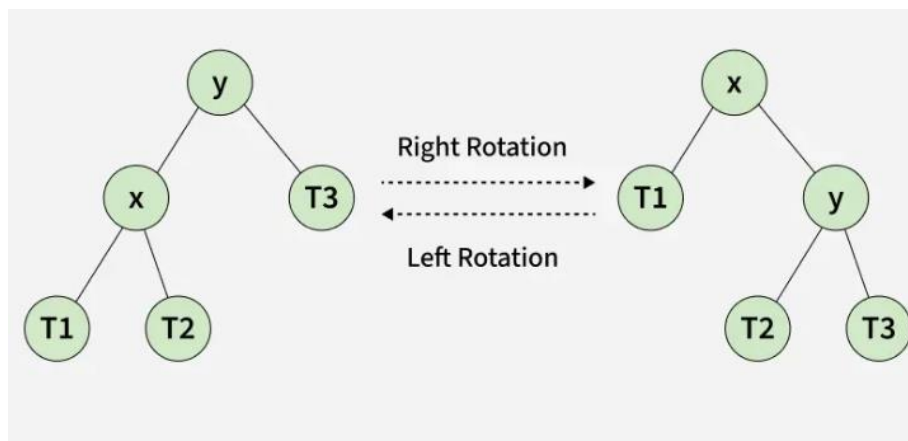
- Get the balance factor (left subtree height - right subtree height) of the current node.
- If the balance factor is greater than **1**, then the current node is unbalanced and we are either in the **Left Left case** or **left Right case**. To check whether it is **left left case** or not, compare the newly inserted key with the key in the **left subtree root**.
- If the balance factor is less than **-1**, then the current node is unbalanced and we are either in the **Right Right case** or **Right-Left case**. To check whether it is the **Right Right case** or not, compare the newly inserted key with the key in the **right subtree root**.

Deletion in an AVL Tree

Steps to follow for deletion.

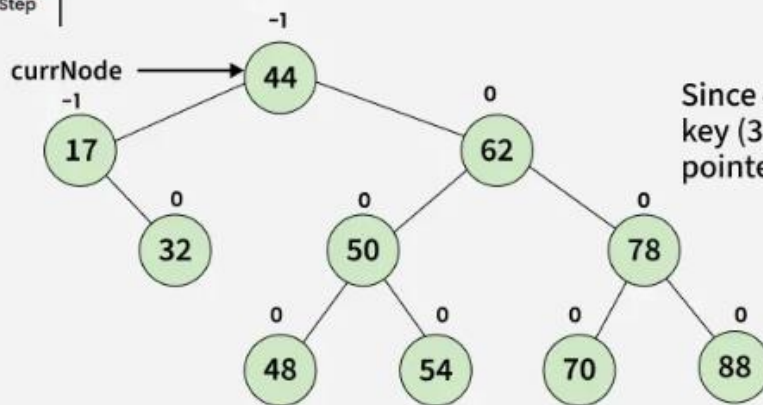
To make sure that the given tree remains AVL after every deletion, we must augment the standard BST delete operation to perform some re-balancing. Following are two basic operations that can be performed to re-balance a BST without violating the BST property ($\text{keys}(\text{left}) < \text{key}(\text{root}) < \text{keys}(\text{right})$).

1. Left Rotation
2. Right Rotation



02
Step

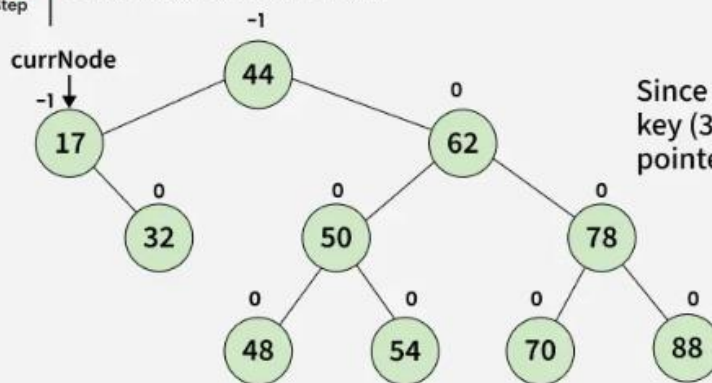
Comparing key with root node



Since 40 is greater than key (32), move the pointer to the left child.

03
Step

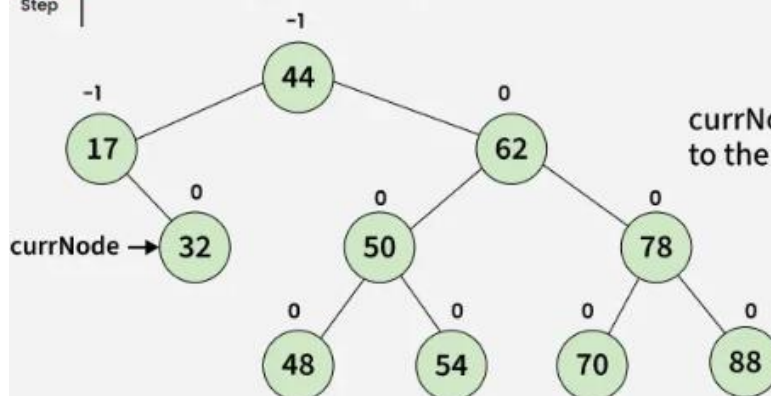
Comparing key with node(17)



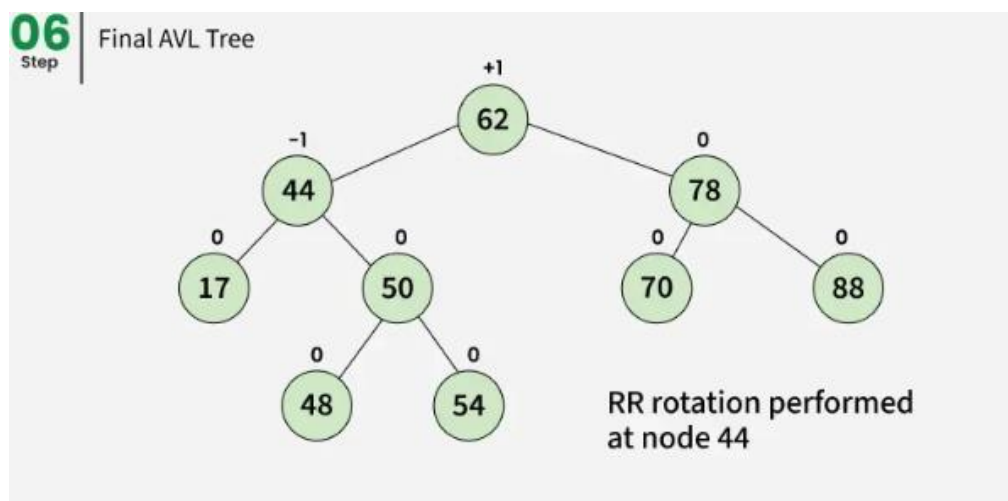
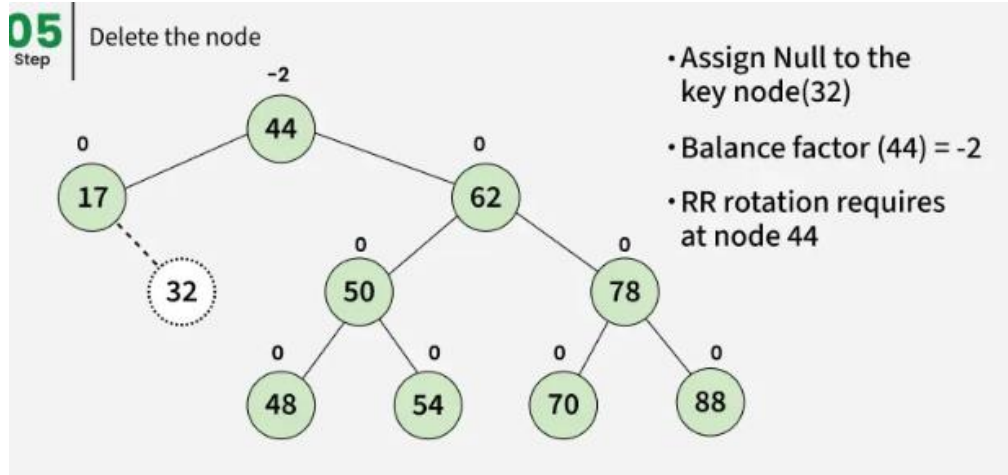
Since 17 is smaller than key (32), move the pointer to the right child.

04
Step

Comparing key with node(32)



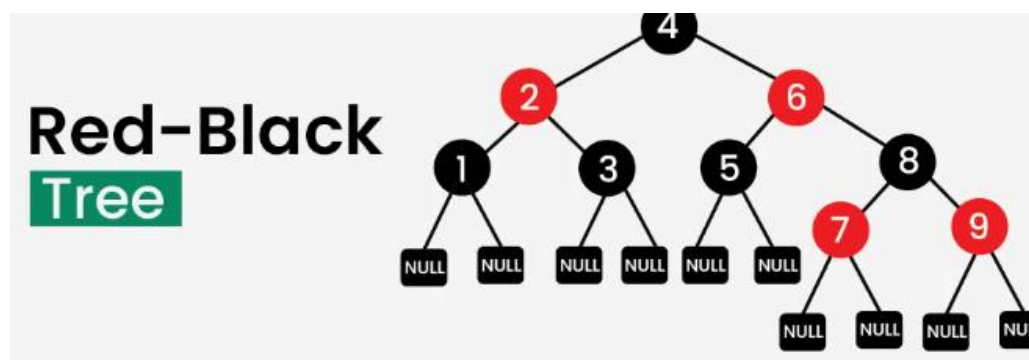
currNode pointing to the key node (32)



Red-Black Tree :

A Red-Black Tree is a self-balancing [binary search tree](#) where the height of the tree is never beyond $O(\log n)$. The main advantage of limiting height is efficient search, insert and delete operations. All of these operations happen in $O(\log n)$ time whereas in a normal binary search tree, these operations may take $O(n)$ time.

- Each node has an additional attribute: a color, which can be either **red** or **black**.
- The colors are used to maintain balance during insertions and deletions, ensuring efficient data retrieval and manipulation.

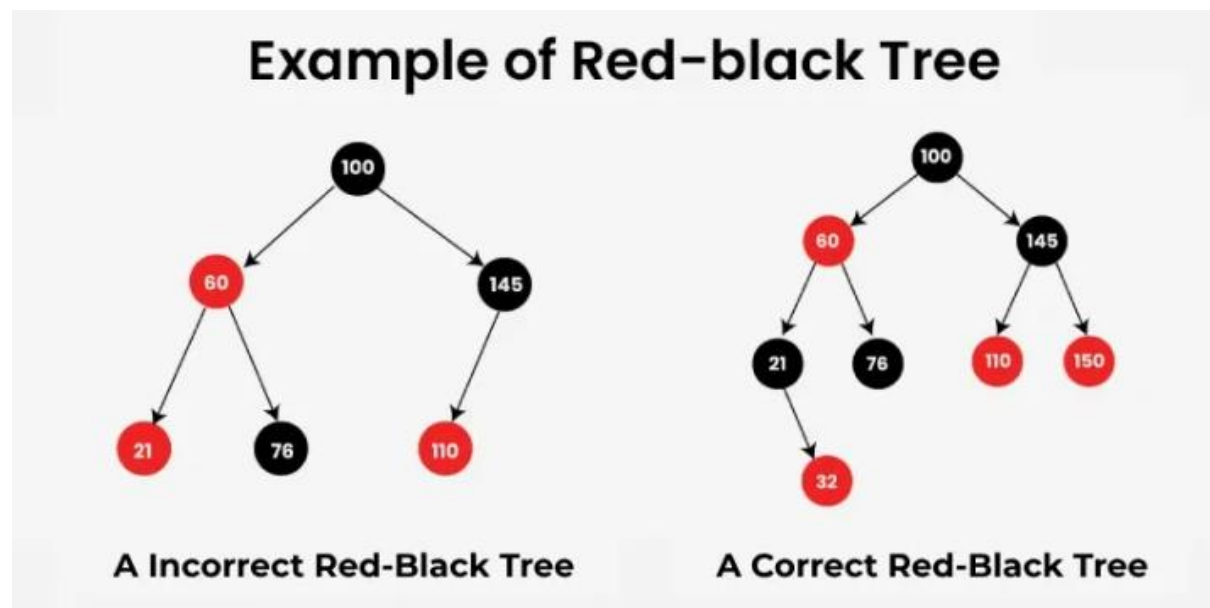


Properties of Red-Black Trees

A Red-Black Tree have the following properties:

1. **Node Color:** Each node is either red or **black**.
2. **Root Property:** The root of the tree is always **black**.
3. **Red Property:** Red nodes cannot have red children (no two consecutive red nodes on any path).
4. **Black Property:** Every path from a node to its descendant null nodes (leaves) has the same number of **black** nodes.
5. **Leaf Property:** All leaves (NIL nodes) are **black**.

These properties ensure that the longest path from the root to any leaf is no more than twice as long as the shortest path, maintaining the tree's balance and efficient performance.



The **Correct Red-Black Tree** in above image ensures that every path from the root to a leaf node has the same number of black nodes. In this case, there is one (excluding the root node). The **Incorrect Red Black Tree** does not follow the red-black properties as **two red nodes** are adjacent to each other. Another problem is that one of the paths to a leaf node has zero black nodes, whereas the other two contain a black node.

Why Red-Black Trees?

Most of the BST operations (e.g., search, max, min, insert, delete.. etc) take **$O(h)$** time where h is the height of the BST. The cost of these operations may become **$O(n)$** for a skewed Binary Search Tree. In Red Black Tree, we make sure that the height of the tree remains **$O(\log n)$** after every insertion and deletion, then we can guarantee an upper bound of **$O(\log n)$** for all these operations.

Sr. No.	Algorithm	Time Complexity
1.	Search	$O(\log n)$

Sr. No.	Algorithm	Time Complexity
2.	Insert	$O(\log n)$
3.	Delete	$O(\log n)$

Basic Operations on Red-Black Tree:

The basic operations on a Red-Black Tree include:

1. Insertion
2. Search
3. Deletion
4. Rotation

1. Insertion

Inserting a new node in a Red-Black Tree involves a two-step process: performing a standard binary search tree (BST) insertion, followed by fixing any violations of Red-Black properties.

Insertion Steps:

1. **BST Insert:** Insert the new node like in a standard BST.
2. **Fix Violations:**
 - If the parent of the new node is **black**, no properties are violated.
 - If the parent is **red**, the tree might violate the Red Property, requiring fixes.

Fixing Violations During Insertion

After inserting the new node as a **red** node, we might encounter several cases depending on the colors of the node's parent and uncle (the sibling of the parent):

- **Case 1: Uncle is Red:** Recolor the parent and uncle to **black**, and the grandparent to **red**. Then move up the tree to check for further violations.
- **Case 2: Uncle is Black:**
 - **Sub-case 2.1: Node is a right child:** Perform a left rotation on the parent.
 - **Sub-case 2.2: Node is a left child:** Perform a right rotation on the grandparent and recolor appropriately.

2. Searching

Searching for a node in a Red-Black Tree is similar to searching in a standard **Binary Search Tree (BST)**. The search operation follows a straightforward path from the **root** to a **leaf**, comparing the target value with the current node's value and moving left or right accordingly.

Search Steps

1. **Start at the Root:** Begin the search at the root node.
2. **Traverse the Tree:**
 - If the target value is equal to the current node's value, the node is found.
 - If the target value is less than the current node's value, move to the left child.
 - If the target value is greater than the current node's value, move to the right child.

3. **Repeat:** Continue this process until the target value is found or a NIL node is reached (indicating the value is not present in the tree).

3. Deletion

Deleting a node from a Red-Black Tree also involves a two-step process: performing the BST deletion, followed by fixing any violations that arise.

Deletion Steps

1. **BST Deletion:** Remove the node using standard BST rules.
2. **Fix Double Black:**
 - If a black node is deleted, a "double black" condition might arise, which requires specific fixes.

Fixing Violations During Deletion

When a black node is deleted, we handle the double black issue based on the sibling's color and the colors of its children:

- **Case 1: Sibling is Red:** Rotate the parent and recolor the sibling and parent.
- **Case 2: Sibling is Black:**
 - **Sub-case 2.1: Sibling's children are black:** Recolor the sibling and propagate the double black upwards.
 - **Sub-case 2.2: At least one of the sibling's children is red:**
 - **If the sibling's far child is red:** Perform a rotation on the parent and sibling, and recolor appropriately.
 - **If the sibling's near child is red:** Rotate the sibling and its child, then handle as above.

4. Rotation

Rotations are fundamental operations in maintaining the balanced structure of a Red-Black Tree (RBT). They help to preserve the properties of the tree, ensuring that the longest path from the root to any leaf is no more than twice the length of the shortest path. Rotations come in two types: **left rotations** and **right rotations**.

1. Left Rotation

A left rotation at node xx moves xx down to the left and its right child yy up to take xx 's place.

Before Rotation:

```
x
 \
  y
 /\
a  b
```

After Left Rotation:


```

    y
   /\
  x  b
   \
    a

```

Left Rotation Steps:

1. Set y to be the right child of x .
2. Move y 's left subtree to x 's right subtree.
3. Update the parent of x and y .
4. Update x 's parent to point to y instead of x .
5. Set y 's left child to x .
6. Update x 's parent to y .

2. Right Rotation

A right rotation at node xx moves xx down to the right and its left child yy up to take xx 's place.

Before Right Rotation:

```

    x
   /
  y
 /\
a  b

```

After Right Rotation:

```

    y
   /\
  a  x
   /
  b

```

Right Rotation Steps:

1. Set y to be the left child of x .
2. Move y 's right subtree to x 's left subtree.
3. Update the parent of x and y .
4. Update x 's parent to point to y instead of x .
5. Set y 's right child to x .
6. Update x 's parent to y .

When to Perform Rotations?

Rotations in Red-Black Trees are typically performed during insertions and deletions to maintain the properties of the tree. Below are the scenarios for rotations:

1. Fixing Violations after Insertion

When a new node is inserted, it is always colored red. This can create violations of Red-Black Tree properties, specifically:

- The root must be **black**.
- **Red** nodes cannot have **red** children.

Case Analysis for Fixing Insertions:

- **Case 1: Recoloring and Propagating Upwards**
 - If the parent and uncle of the new node are both **red**, recolor the parent and uncle to **black**, and the grandparent to **red**. Then, recursively apply the fix-up to the grandparent.
- **Case 2: Rotation and Recoloring**
 - If the new node's uncle is **black** and the new node is the right child of a left child (or vice versa), perform a rotation to move the new node up and align it.
 - If the new node's uncle is **black** and the new node is the left child of a left child (or right of a right), perform a rotation and recolor the parent and grandparent to fix the violation.

2. Fixing Violations after Deletion

After deletion, the tree might need fixing to restore properties:

- When a black node is removed, or a red node is replaced by a black node, a double-black situation can arise.

Case Analysis for Fixing Deletions:

- **Case 1: Sibling is Red**
 - Recolor the sibling and the parent, and perform a rotation.
- **Case 2: Sibling is Black with Black Children**
 - Recolor the sibling to red and move the problem up to the parent.
- **Case 3: Sibling is Black with at least one Red Child**
 - Rotate and recolor to fix the double-black issue.

Advantages of Red-Black Trees:

- **Balanced:** Red-Black Trees are self-balancing, meaning they automatically maintain a balance between the heights of the left and right subtrees. This ensures that search, insertion, and deletion operations take $O(\log n)$ time in the worst case.
- **Efficient search, insertion, and deletion:** Due to their balanced structure, Red-Black Trees offer efficient operations. Search, insertion, and deletion all take $O(\log n)$ time in the worst case.
- **Simple to implement:** The rules for maintaining the Red-Black Tree properties are relatively simple and straightforward to implement.
- **Widely used:** Red-Black Trees are a popular choice for implementing various data structures, such as maps, sets, and priority queues.

Disadvantages of Red-Black Trees:

- **More complex than other balanced trees:** Compared to simpler balanced trees like AVL trees, Red-Black Trees have more complex insertion and deletion rules.
- **Constant overhead:** Maintaining the Red-Black Tree properties adds a small overhead to every insertion and deletion operation.

Applications of Red-Black Trees:

- **Implementing maps and sets:** Red-Black Trees are used in libraries like TreeMap and TreeSet in Java, and map and set in C++.
- **File systems:** Red-Black Trees are used in some file systems to manage file and directory structures.
- **In-memory databases:** Red-Black Trees are sometimes used in in-memory databases to store and retrieve data efficiently.
- **Graphics and game development:** Red-Black Trees can be used in graphics and game development for tasks like collision detection and pathfinding.

Splay tree :

Splay tree is a self-adjusting binary search tree data structure, which means that the tree structure is adjusted dynamically based on the accessed or inserted elements. In other words, the tree automatically reorganizes itself so that frequently accessed or inserted elements become closer to the root node.

1. The splay tree was first introduced by Daniel Dominic Sleator and Robert Endre Tarjan in 1985. It has a simple and efficient implementation that allows it to perform search, insertion, and deletion operations in $O(\log n)$ amortized time complexity, where n is the number of elements in the tree.
2. The basic idea behind splay trees is to bring the most recently accessed or inserted element to the root of the tree by performing a sequence of tree rotations, called splaying. Splaying is a process of restructuring the tree by making the most recently accessed or inserted element the new root and gradually moving the remaining nodes closer to the root.
3. Splay trees are highly efficient in practice due to their self-adjusting nature, which reduces the overall access time for frequently accessed elements. This makes them a good choice for applications that require fast and dynamic data structures, such as caching systems, data compression, and network routing algorithms.
4. However, the main disadvantage of splay trees is that they do not guarantee a balanced tree structure, which may lead to performance degradation in worst-case scenarios. Also, splay trees are not suitable for applications that require guaranteed worst-case performance, such as real-time systems or safety-critical systems.

Overall, splay trees are a powerful and versatile data structure that offers fast and efficient access to frequently accessed or inserted elements. They are widely used in various applications and provide an excellent tradeoff between performance and simplicity.

- The key feature of a splay tree is that each time an element is accessed, it is moved to the root of the tree, creating a more balanced structure for subsequent accesses.

- Splay trees are characterized by their use of rotations, which are local transformations of the tree that change its shape but preserve the order of the elements.
- Rotations are used to bring the accessed element to the root of the tree, and also to rebalance the tree if it becomes unbalanced after multiple accesses.

Operations in a splay tree:

- **Insertion:** To insert a new element into the tree, start by performing a regular binary search tree insertion. Then, apply rotations to bring the newly inserted element to the root of the tree.
- **Deletion:** To delete an element from the tree, first locate it using a binary search tree search. Then, if the element has no children, simply remove it. If it has one child, promote that child to its position in the tree. If it has two children, find the successor of the element (the smallest element in its right subtree), swap its key with the element to be deleted, and delete the successor instead.
- **Search:** To search for an element in the tree, start by performing a binary search tree search. If the element is found, apply rotations to bring it to the root of the tree. If it is not found, apply rotations to the last node visited in the search, which becomes the new root.
- **Rotation:** The rotations used in a splay tree are either a Zig or a Zig-Zig rotation. A Zig rotation is used to bring a node to the root, while a Zig-Zig rotation is used to balance the tree after multiple accesses to elements in the same subtree.

Here's a step-by-step explanation of the rotation operations:

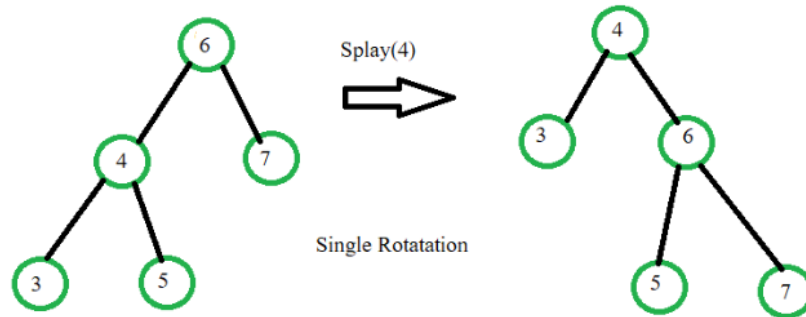
- **Zig Rotation:** If a node has a right child, perform a right rotation to bring it to the root. If it has a left child, perform a left rotation.
- **Zig-Zig Rotation:** If a node has a grandchild that is also its child's right or left child, perform a double rotation to balance the tree. For example, if the node has a right child and the right child has a left child, perform a right-left rotation. If the node has a left child and the left child has a right child, perform a left-right rotation.
- **Note:** The specific implementation details, including the exact rotations used, may vary depending on the exact form of the splay tree.

Rotations in Splay Tree

- Zig Rotation
- Zag Rotation
- Zig - Zig Rotation
- Zag - Zag Rotation
- Zig - Zag Rotation
- Zag - Zig Rotation

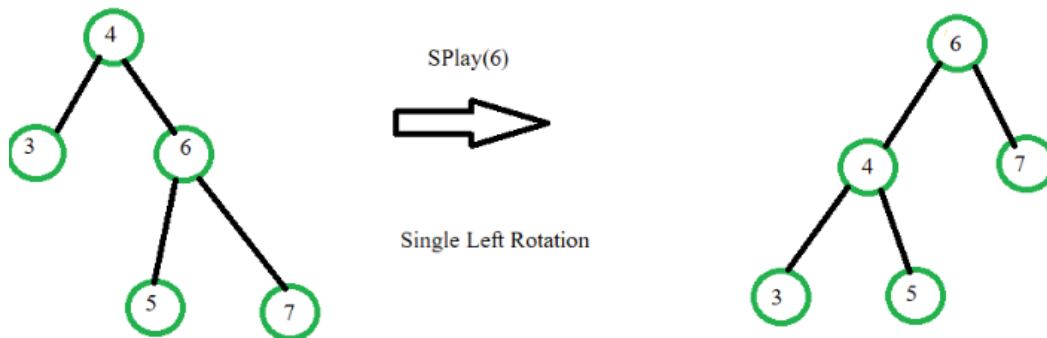
1) Zig Rotation:

The Zig Rotation in splay trees operates in a manner similar to the single right rotation in AVL Tree rotations. This rotation results in nodes moving one position to the right from their current location. For example, consider the following scenario:



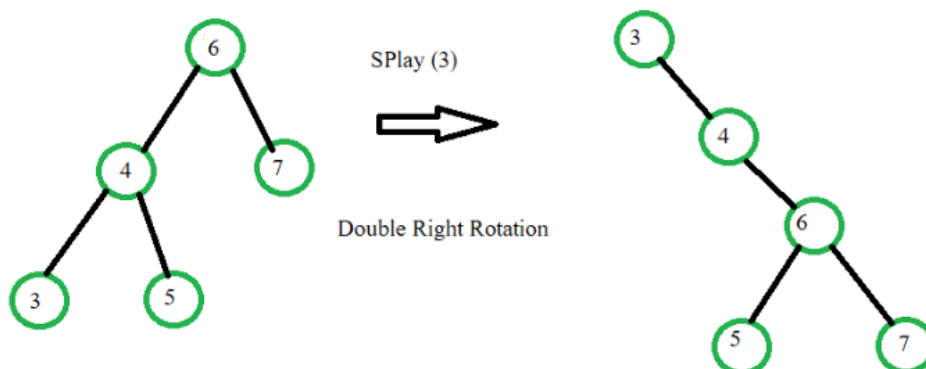
2) Zag Rotation:

The Zag Rotation in splay trees operates in a similar fashion to the single left rotation in AVL Tree rotations. During this rotation, nodes shift one position to the left from their current location. For instance, consider the following illustration:



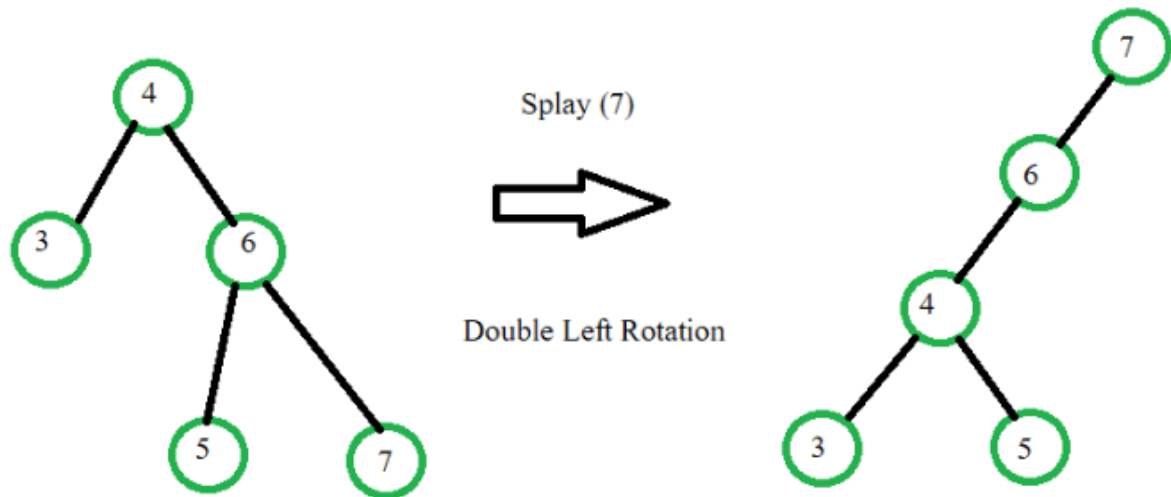
3) Zig-Zig Rotation:

The Zig-Zig Rotation in splay trees is a double zig rotation. This rotation results in nodes shifting two positions to the right from their current location. Take a look at the following example for a better understanding:



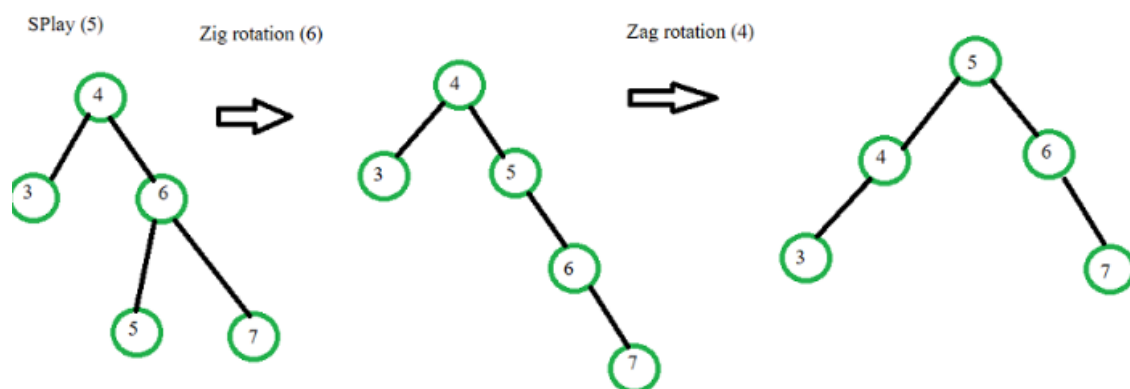
4) Zag-Zag Rotation:

In splay trees, the Zag-Zag Rotation is a double zag rotation. This rotation causes nodes to move two positions to the left from their present position. For example:



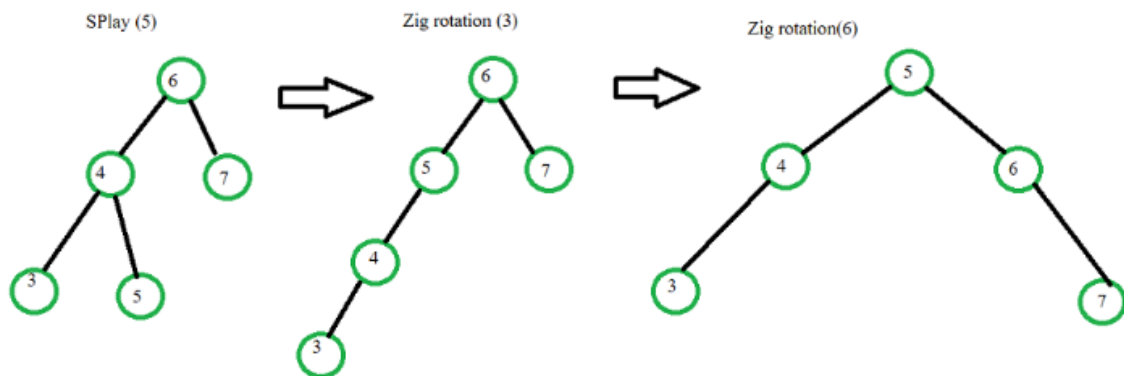
5) Zig-Zag Rotation:

The Zig-Zag Rotation in splay trees is a combination of a zig rotation followed by a zag rotation. As a result of this rotation, nodes shift one position to the right and then one position to the left from their current location. The following illustration provides a visual representation of this concept:



6) Zag-Zig Rotation:

The Zag-Zig Rotation in splay trees is a series of zag rotations followed by a zig rotation. This results in nodes moving one position to the left, followed by a shift one position to the right from their current location. The following illustration offers a visual representation of this concept:



Drawbacks of splay tree data structure:

- **Unbalanced Trees:** Splay trees can become unbalanced and inefficient if the tree is repeatedly rotated in the same direction.
- **Memory Usage:** Splay trees can use a lot of memory compared to other data structures because each node contains additional information.
- **Complexity:** Splay trees can have a high time complexity for basic operations such as insertion and deletion because the trees need to be reorganized after every operation.
- **Reorganization Overhead:** The splaying operation required in every operation can be time-consuming and result in a high overhead.
- **Limited Use Cases:** Splay trees are not suitable for all data structures and have limited use cases because they don't handle duplicate keys efficiently.

Applications of the splay tree:

- **Caching:** Splay trees can be used to implement cache memory management, where the most frequently accessed items are moved to the top of the tree for quicker access.
- **Database Indexing:** Splay trees can be used to index databases for faster searching and retrieval of data.
- **File Systems:** Splay trees can be used to store file system metadata, such as the allocation table, directory structure, and file attributes.
- **Data Compression:** Splay trees can be used to compress data by identifying and encoding repeating patterns.
- **Text Processing:** Splay trees can be used in text processing applications, such as spell-checkers, where words are stored in a splay tree for quick searching and retrieval.
- **Graph Algorithms:** Splay trees can be used to implement graph algorithms, such as finding the shortest path in a weighted graph.
- **Online Gaming:** Splay trees can be used in online gaming to store and manage high scores, leaderboards, and player statistics.

Sure, here are some advantages and disadvantages of splay trees, as well as some recommended books for learning more about the topic:

Advantages of Splay Trees:

Splay trees have amortized time complexity of $O(\log n)$ for many operations, making them faster than many other balanced tree data structures in some cases.

Splay trees are self-adjusting, meaning that they automatically balance themselves as items are inserted and removed. This can help to avoid the performance degradation that can occur when a tree becomes unbalanced.

Disadvantages of Splay Trees:

Splay trees can have worst-case time complexity of $O(n)$ for some operations, making them less predictable than other balanced tree data structures like AVL trees or red-black trees.

Splay trees may not be suitable for certain applications where predictable performance is required.

UNIT III

Graph and its representations:

A **Graph** is a non-linear data structure consisting of vertices and edges. The vertices are sometimes also referred to as nodes and the edges are lines or arcs that connect any two nodes in the graph. More formally a Graph is composed of a set of vertices(**V**) and a set of edges(**E**). The graph is denoted by **G(V, E)**.

Representations of Graph

Here are the two most common ways to represent a graph : For simplicity, we are going to consider only **unweighted graphs** in this post.

1. Adjacency Matrix
2. Adjacency List

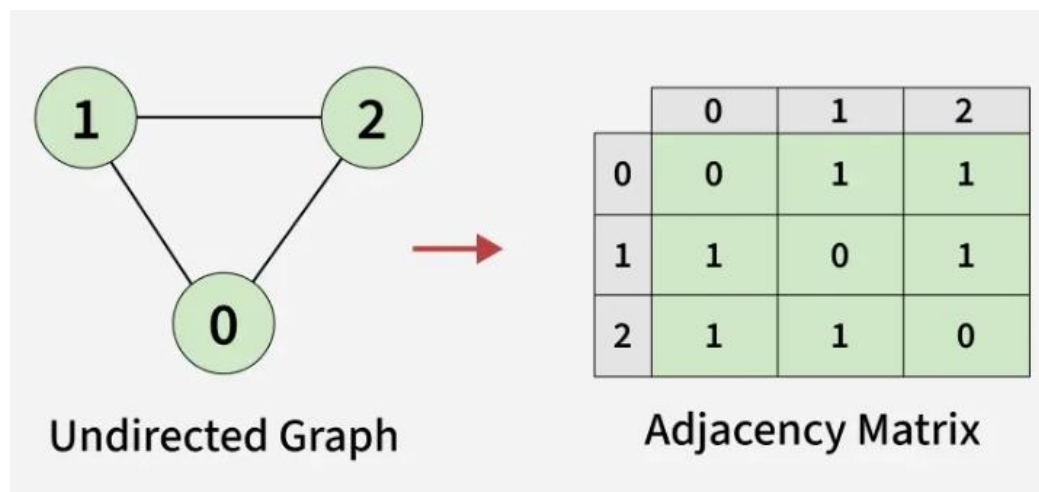
Adjacency Matrix Representation

An adjacency matrix is a way of representing a graph as a boolean matrix of (0's and 1's).

Let's assume there are n vertices in the graph So, create a 2D matrix $\text{adjMat}[n][n]$ having dimension $n \times n$.

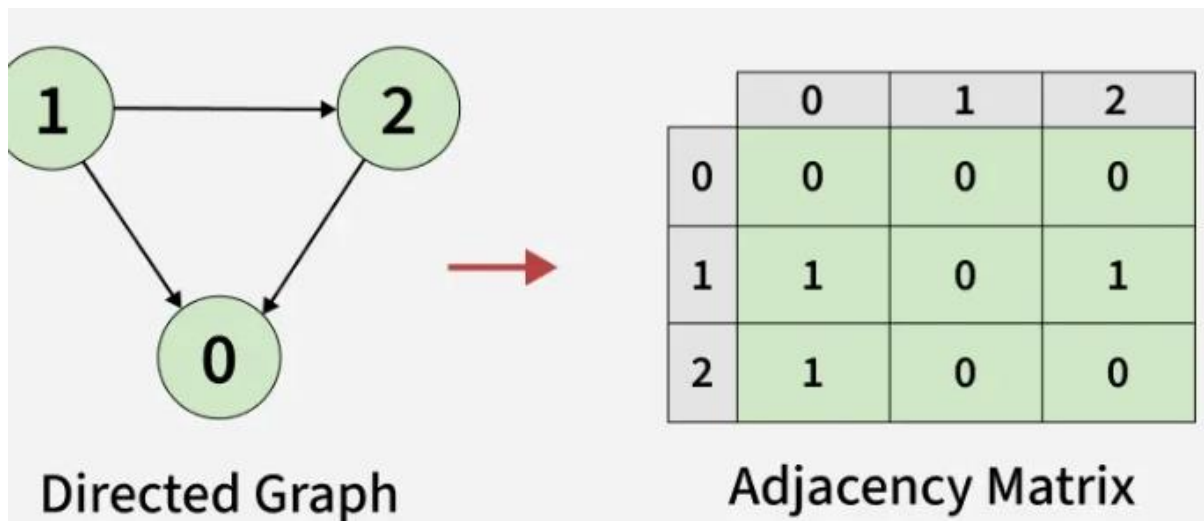
- If there is an edge from vertex i to j , mark $\text{adjMat}[i][j]$ as 1.
- If there is no edge from vertex i to j , mark $\text{adjMat}[i][j]$ as 0.

Representation of Undirected Graph as Adjacency Matrix:



- We use an adjacency matrix to represent connections between vertices.
- Initially, the entire matrix is filled with 0s, meaning no edges exist.
- There is an edge between vertex 0 and vertex 1, so we set $\text{mat}[0][1] = 1$ and $\text{mat}[1][0] = 1$.
- There is an edge between vertex 0 and vertex 2, so we set $\text{mat}[0][2] = 1$ and $\text{mat}[2][0] = 1$.
- There is an edge between vertex 1 and vertex 2, so we set $\text{mat}[1][2] = 1$ and $\text{mat}[2][1] = 1$.

Representation of Directed Graph as Adjacency Matrix:



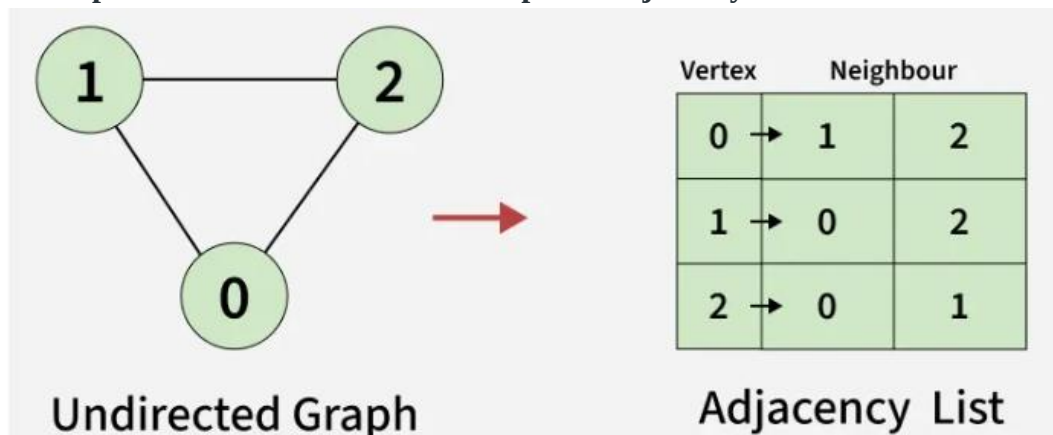
- Initially, the entire matrix is filled with 0s, meaning no edges exist.
- Unlike an undirected graph, we do not set $\text{mat}[\text{destination}][\text{source}]$ because the edge goes in only one direction.
- There is an edge between vertex 1 and vertex 0, so we set $\text{mat}[1][0] = 1$.
- There is an edge between vertex 2 and vertex 0, so we set $\text{mat}[2][0] = 1$.
- There is an edge between vertex 1 and vertex 2, so we set $\text{mat}[1][2] = 1$.

Adjacency List Representation

An array of Lists is used to store edges between two vertices. The size of array is equal to the number of vertices (i.e, n). Each index in this array represents a specific vertex in the graph. The entry at the index i of the array contains a linked list containing the vertices that are adjacent to vertex i. Let's assume there are n vertices in the graph So, create an array of list of size n as $\text{adjList}[n]$.

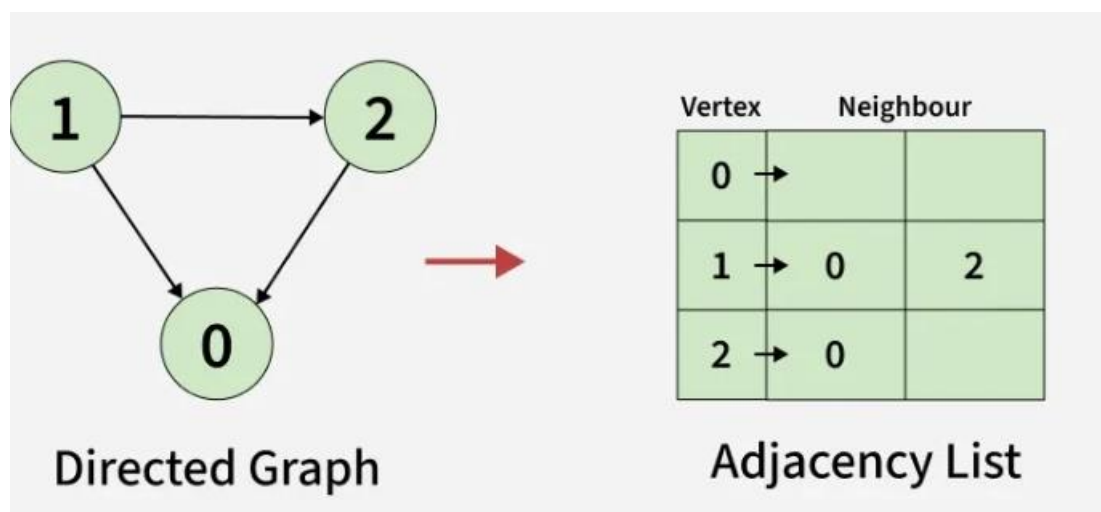
- $\text{adjList}[0]$ will have all the nodes which are connected (neighbour) to vertex 0.
- $\text{adjList}[1]$ will have all the nodes which are connected (neighbour) to vertex 1 and so on.

Representation of Undirected Graph as Adjacency list:



- We use an array of lists (or vector of lists) to represent the graph.
- The size of the array is equal to the number of vertices (here, 3).
- Each index in the array represents a vertex.
- Vertex 0 has two neighbours (1 and 2).
- Vertex 1 has two neighbours (0 and 2).
- Vertex 2 has two neighbours (0 and 1).

Representation of Directed Graph as Adjacency list:



- We use an array of lists (or vector of lists) to represent the graph.
- The size of the array is equal to the number of vertices (here, 3).
- Each index in the array represents a vertex.
- Vertex 0 has no neighbours
- Vertex 1 has two neighbours (0 and 2).
- Vertex 2 has 1 neighbours (0).

Graph Traversals DFS and BFS:

Graph traversal, or graph search, is the process of systematically visiting every vertex (node) in a graph data structure, often to find paths, detect cycles, or analyze network connectivity. The two primary algorithms are Depth-First Search (DFS), which explores as deep as possible along each branch before backtracking using a stack, and Breadth-First Search (BFS), which explores neighbor nodes level-by-level using a queue.

Key Graph Traversal Techniques

- **Depth-First Search (DFS)**: Starts at a root node and explores as far as possible along each branch before backtracking.
 - **Data Structure**: Stack (or recursion).
 - **Use Cases**: Topological sorting, pathfinding, detecting cycles, and solving puzzles like mazes.
- **Breadth-First Search (BFS)**: Starts at a root node and explores all neighboring nodes at the present depth level before moving on to the nodes at the next depth level.

Depth First Search or DFS for a Graph:

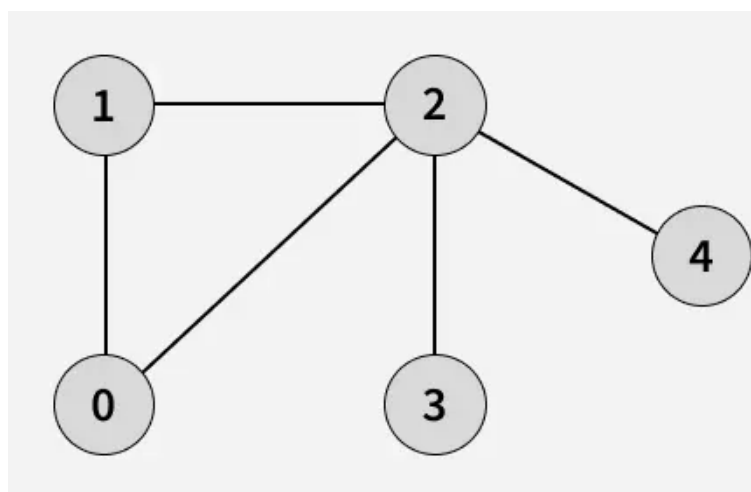
Given a graph, traverse the graph using Depth First Search and find the order in which nodes are visited.

Depth First Search (DFS) is a graph traversal method that starts from a source vertex and explores each path completely before backtracking and exploring other paths. To avoid revisiting nodes in graphs with cycles, a visited array is used to track visited vertices.

Note: There can be multiple DFS traversals of a graph according to the order in which we pick adjacent vertices. Here we pick vertices as per the insertion order.

Example:

Input: $adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]$



Output: [0, 1, 2, 3, 4]

Explanation: The source vertex is 0. We visit it first, then we visit its adjacent.

Start at 0: Mark as visited. Print 0.

Move to 1: Mark as visited. Print 1.

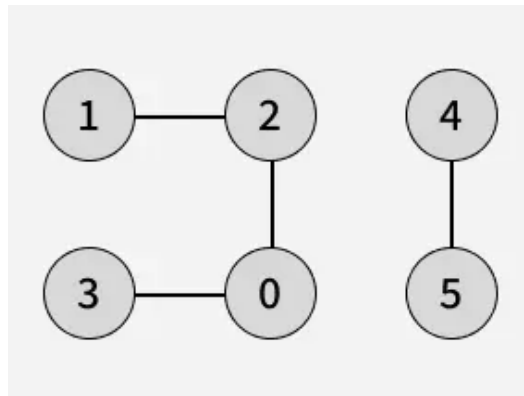
Move to 2: Mark as visited. Print 2.

Move to 3: Mark as visited. Print 3.(backtrack to 2)

Move to 4: Mark as visited. Print 4(backtrack to 2, then backtrack to 1, then to 0).

Note that there can be more than one DFS Traversals of a Graph. For example, after 1, we may pick adjacent 2 instead of 0 and get a different DFS.

Input: $\text{adj}[][] = [[2, 3], [2], [0, 1], [0], [5], [4]]$



Output: [0, 2, 1, 3, 4, 5]

Explanation: DFS Steps:

Start at 0: Mark as visited. Print 0.

Move to 2: Mark as visited. Print 2.

Move to 1: Mark as visited. Print 1 (backtrack to 2, then backtrack to 0).

Move to 3: Mark as visited. Print 3 (backtrack to 0).

Start with 4.

Start at 4: Mark as visited. Print 4.

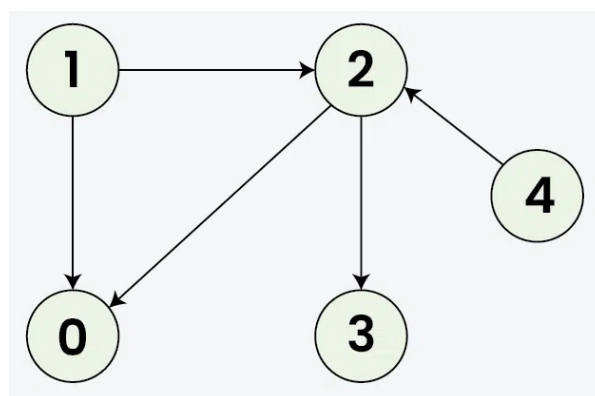
Move to 5: Mark as visited. Print 5. (backtrack to 4)

Depth First Search or DFS on Directed Graph:

Depth-First Search (DFS) is a basic algorithm used to explore graph structures. In directed graphs, **DFS** can **start from a specific point** and explore all the connected nodes. It can also be used to make sure every part of the graph is visited, even if the graph has disconnected sections. This article explains how DFS works when starting from a single point and how it can be used to explore an entire graph, including disconnected parts.

Example:

Input: $V = 5, E = 5, \text{edges} = \{\{1, 2\}, \{1, 0\}, \{2, 0\}, \{2, 3\}, \{4, 2\}\}, \text{source} = 1$

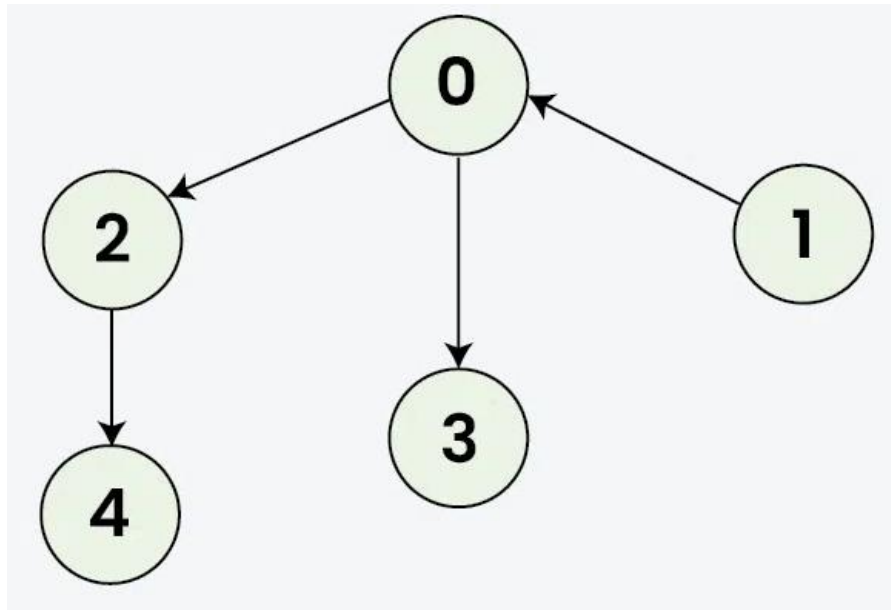


Output: 1 2 0 3

Explanation: DFS Steps:

- Start at 1: Mark as visited. Output: 1
- Move to 2: Mark as visited. Output: 2
- Move to 0: Mark as visited. Output: 0 (backtrack to 2)
- Move to 3: Mark as visited. Output: 3 (backtrack to 2)
- All neighbors of 2 explored (backtrack to 1)
- All neighbors of 1 explored (end of traversal)

Input: V = 5, E = 4, edges = {{0, 2}, {0, 3}, {2, 4}, {1, 0}}, source = 2



Output: 2 4

Explanation: DFS Steps:

- Start at 2: Mark as visited. Output: 2
- Move to 4: Mark as visited. Output: 4 (backtrack to 2)
- All neighbors of 2 explored (backtrack to start)

DFS from a Given Source in Directed Graph:

Depth-First Search (DFS) from a given source is a method of exploring a directed graph by starting at a specific vertex and traversing each node as far as we can go down in the path. If we reach a vertex that has no unvisited neighbors, we backtrack to the previous vertex to explore any other paths that haven't been visited yet. This approach is particularly useful for tasks such as finding paths, checking connectivity, and exploring all reachable nodes from a starting point.

How It Works:

- Keep a boolean **visited** array to keep track of which vertices have already been visited. This will help in not entering in infinite loops when there are cycles in the graph.
- Use Recursion to visited all unvisited neighbours of source node:
 - First marks the current vertex as visited and processes the current vertex (for example, by printing its value).
 - Then, recursively visits each unvisited neighbor of the current vertex.

- If a vertex has no unvisited neighbors, backtracks to the previous vertex to explore other unvisited paths.

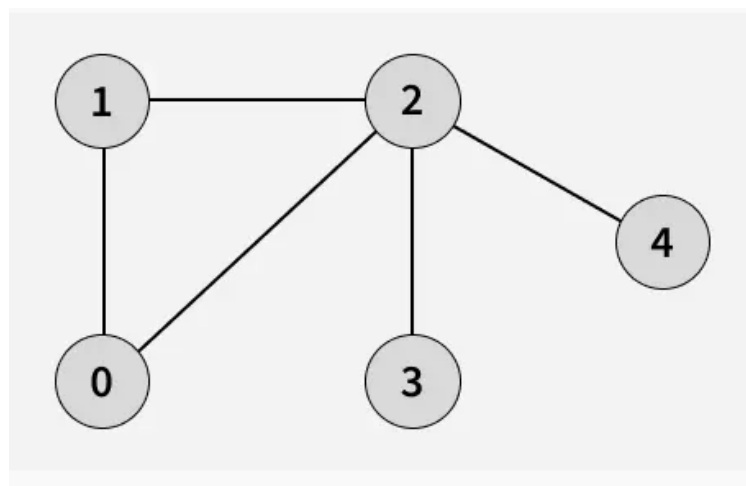
Breadth First Search or BFS for a Graph:

Breadth First Search (BFS) is a graph traversal algorithm that starts from a source node and explores the graph level by level. First, it visits all nodes directly adjacent to the source. Then, it moves on to visit the adjacent nodes of those nodes, and this process continues until all reachable nodes are visited.

- BFS is different from DFS in a way that closest vertices are visited before others. We mainly traverse vertices level by level.
- Popular graph algorithms like Dijkstra's shortest path, Kahn's Algorithm, and Prim's algorithm are based on BFS.
- BFS itself can be used to detect cycle in a directed and undirected graph, find shortest path in an unweighted graph and many more problems.

Examples:

Input: `adj[][] = [[1, 2], [0, 2], [0, 1, 3, 4], [2], [2]]`



Output: [0, 1, 2, 3, 4]

Explanation: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 - Print: 0

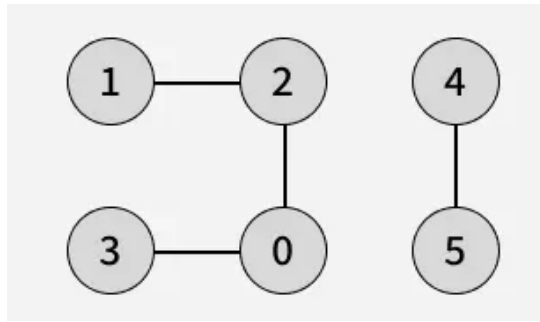
Visit 1 (neighbor of 0) - Print: 1

Visit 2 (next neighbor of 0) - Print: 2

Visit 3 (first neighbor of 2 that hasn't been visited yet) - Print: 3

Visit 4 (next neighbor of 2) - Print: 4

Input: `adj[][] = [[2, 3], [2], [0, 1], [0], [5], [4]]`



Output: [0, 2, 3, 1, 4, 5]

Explanation: Starting from 0, the BFS traversal proceeds as follows:

Visit 0 - Print: 0

Visit 2 (first neighbor of 0) - Print: 2

Visit 3 (next neighbor of 0) - Print: 3

Visit 1 (neighbor of 2 that hasn't been visited yet) - Print: 1

Start another BFS traversal with source as 4:

Visit 4 - Print: 4

Visit 5 (neighbor of 4) - Print: 5

BFS of a Disconnected Undirected Graph:

In a disconnected graph, some vertices may not be reachable from a single source. To ensure all vertices are visited in BFS traversal, we iterate through each vertex, and if any vertex is unvisited, we perform a BFS starting from that vertex being the source. This way, BFS explores every connected component of the graph.

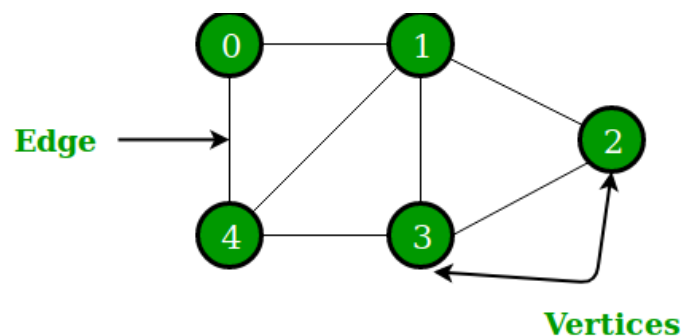
Applications of BFS in Graphs

BFS has various applications in graph theory and computer science, including:

- Shortest Path Finding
- Cycle Detection
- Connected Components
- Network Routing

Applications of Graphs:

A graph is a non-linear data structure, which consists of vertices(or nodes) connected by edges(or arcs) where edges may be directed or undirected.



- In **Computer science** graphs are used to represent the flow of computation.

- **Google maps** uses graphs for building transportation systems, where intersection of two(or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
- In **Facebook**, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of **undirected graph**.
- In **World Wide Web**, web pages are considered to be the vertices. There is an edge from a page u to other page v if there is a link of page v on page u. This is an example of **Directed graph**. It was the basic idea behind [Google Page Ranking Algorithm](#).
- In **Operating System**, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle then a deadlock will occur.
- In **mapping system** we use graph. It is useful to find out which is an excellent place from the location as well as your nearby location. In GPS we also use graphs.
- **Facebook** uses graphs. Using graphs suggests mutual friends. it shows a list of the following pages, friends, and contact list.
- **Microsoft Excel** uses DAG means Directed Acyclic Graphs.
- In the **Dijkstra algorithm**, we use a graph. we find the smallest path between two or many nodes.
- On **social media** sites, we use graphs to track the data of the users. liked showing preferred post suggestions, recommendations, etc.
- Graphs are used in biochemical applications such as structuring of protein, **DNA** etc.

Minimum spanning tree:

spanning tree is defined as a tree-like subgraph of a connected, undirected graph that includes all the vertices of the graph. Or, to say in Layman's words, it is a subset of the edges of the graph that forms a tree (**acyclic**) where every node of the graph is a part of the tree. The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

Minimum Spanning Tree for Directed Graph



Properties of a Spanning Tree

The spanning tree holds the **below-mentioned principles**:

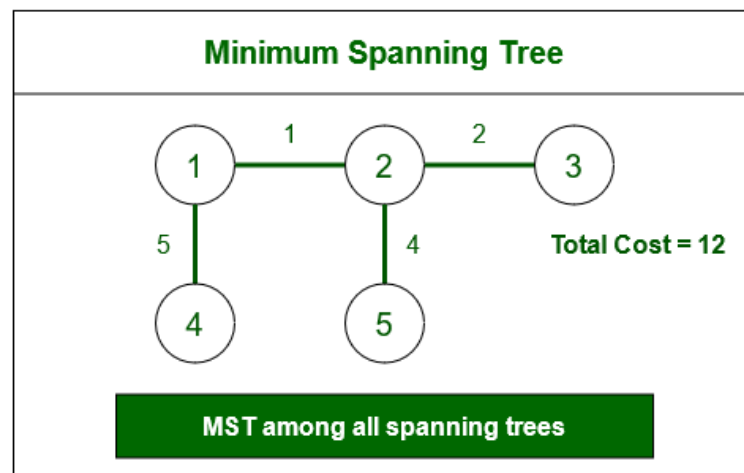
- The number of vertices (**V**) in the graph and the spanning tree is the same.
- There is a fixed number of edges in the spanning tree which is equal to one less than the total number of vertices ($E = V - 1$).
- The spanning tree should not be **disconnected**, as in there should only be a single source of component, not more than that.
- The spanning tree should be **acyclic**, which means there would not be any cycle in the tree.
- The total cost (or weight) of the spanning tree is defined as the sum of the edge weights of all the edges of the spanning tree.
- There can be many possible spanning trees for a graph.

Minimum Spanning Tree

A **minimum spanning tree (MST)** is defined as a **spanning tree** that has the minimum weight among all the possible spanning trees.

The minimum spanning tree has all the properties of a spanning tree with an added constraint of having the minimum possible weights among all possible spanning trees. Like a spanning tree, there can also be many possible MSTs for a graph.

- Let's look at the MST of the above example Graph,



Minimum Spanning Tree

Algorithms to find Minimum Spanning Tree

There are several algorithms to find the minimum spanning tree from a given graph, some of them are listed below:

Kruskal's Minimum Spanning Tree Algorithm

This is one of the popular algorithms for finding the minimum spanning tree from a connected, undirected graph. This is a [greedy algorithm](#). The algorithm workflow is as below:

- First, it sorts all the edges of the graph by their weights,
- Then starts the iterations of finding the spanning tree.
- At each iteration, the algorithm adds the next lowest-weight edge one by one, such that the edges picked until now does not form a cycle.

Applications of Minimum Spanning Trees:

- **Network design:** Spanning trees can be used in network design to find the minimum number of connections required to connect all nodes. Minimum spanning trees, in particular, can help minimize the cost of the connections by selecting the cheapest edges.
- **Image processing:** Spanning trees can be used in image processing to identify regions of similar intensity or color, which can be useful for segmentation and classification tasks.
- **Biology:** Spanning trees and minimum spanning trees can be used in biology to construct phylogenetic trees to represent evolutionary relationships among species or genes.
- **Social network analysis:** Spanning trees and minimum spanning trees can be used in social network analysis to identify important connections and relationships among individuals or groups.

Minimum spanning Tree using Kruskal's Algorithm:

minimum spanning tree (MST) or minimum weight spanning tree for a weighted, connected, and undirected graph is a [spanning tree](#) (no cycles and connects all vertices) that has minimum weight. The weight of a spanning tree is the sum of all edges in the tree.

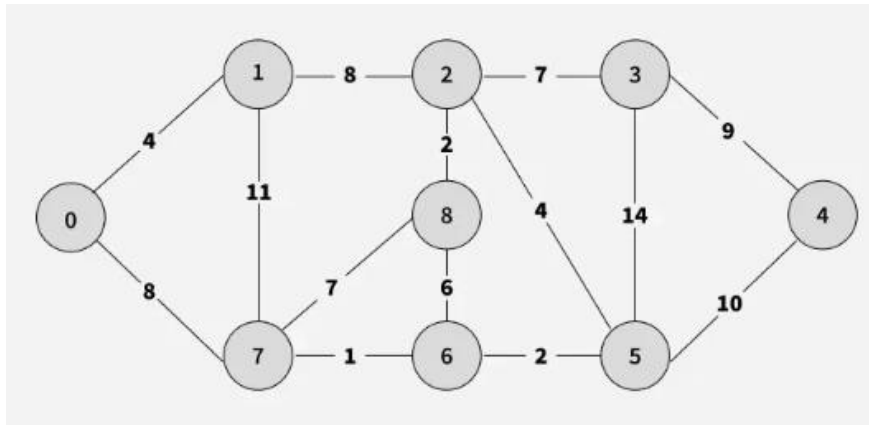
Below are the steps for finding MST using Kruskal's algorithm:

- Sort all the edges in a non-decreasing order of their weight.
- Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If the cycle is not formed, include this edge. Else, discard it. It uses the [Disjoint Sets](#) to detect cycles.
- Repeat step 2 until there are $(V-1)$ edges in the spanning tree.

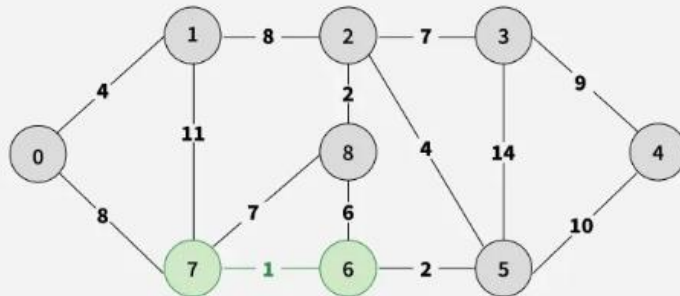
It picks the minimum weighted edge at first and the maximum weighted edge at last. Thus we can say that it makes a locally optimal choice in each step in order to find the optimal solution. Hence this is a Greedy Algorithm.

Illustration:

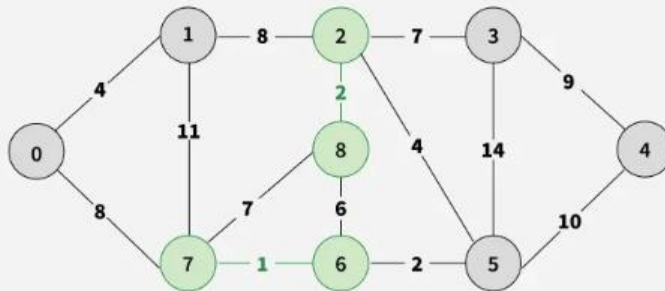
The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having $(9 - 1) = 8$ edges.



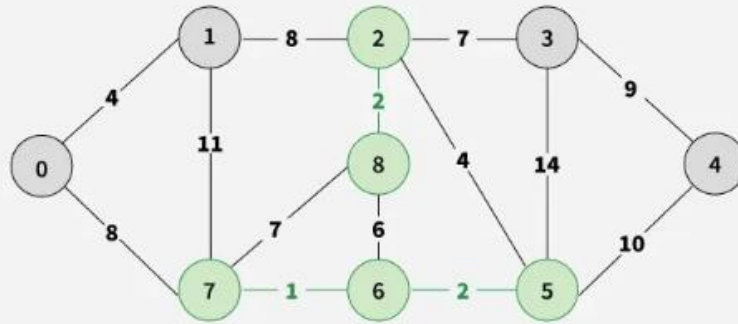
02 Step | Pick all sorted edges one by one.
Pick edge 7-6. No cycle is formed, include it.



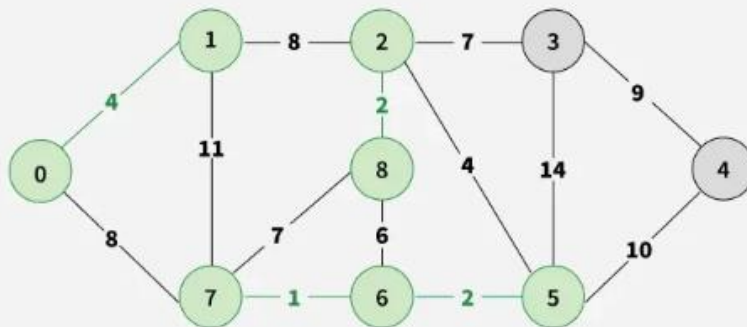
03 Step | Pick edge 8-2. No cycle is formed, include it.



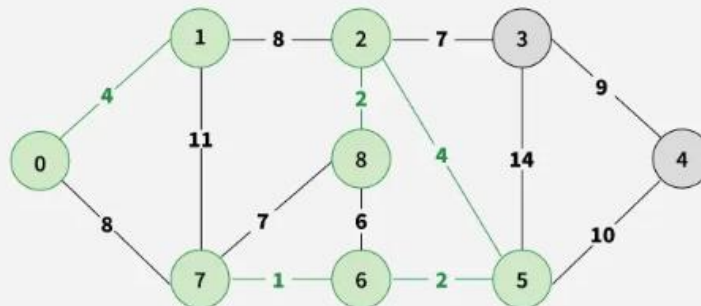
04 Step | Pick edge 6-5. No cycle is formed, include it.



05 Step | Pick edge 0-1. No cycle is formed, include it.

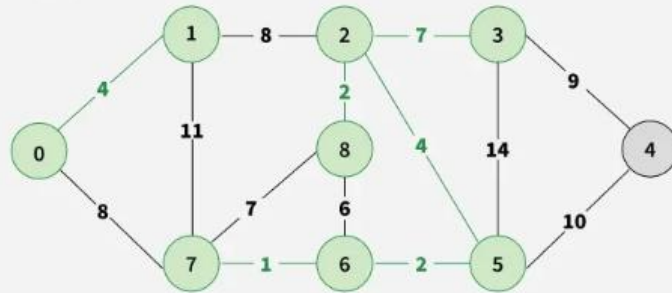


06 Step | Pick edge 2-5. No cycle is formed, include it.



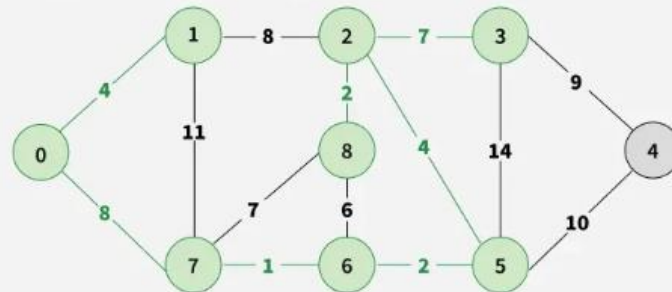
07
Step

Pick edge 8-6. Since including this edge results in the cycle, discard it.
Pick edge 2-3: No cycle is formed, include it.



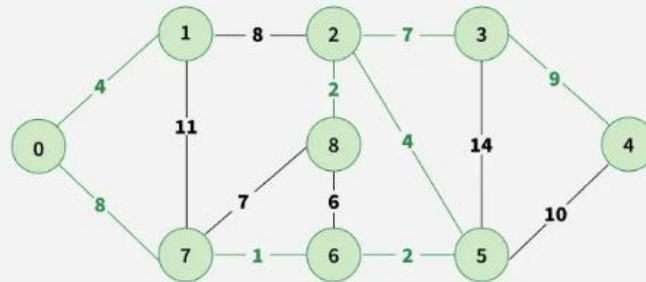
08
Step

Pick edge 7-8. Since including this edge results in the cycle, discard it.
Pick edge 0-7: No cycle is formed, include it.



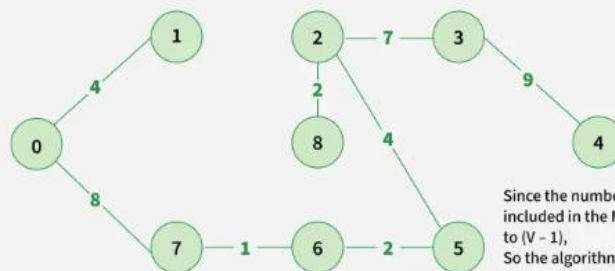
09
Step

Pick edge 1-2. Since including this edge results in the cycle, discard it.
Pick edge 3-4: No cycle is formed, include it.



10
Step

Final obtained graph minimum spanning tree.



Since the number of edges included in the MST equals to $(V - 1)$, So the algorithm stops here. Total cost of MST = (37)

Time Complexity: $O(E * \log E)$ or $O(E * \log V)$

- Sorting of edges takes $O(E * \log E)$ time.
- After sorting, we iterate through all edges and apply the find-union algorithm. The find and union operations can take at most $O(\log V)$ time.
- So overall complexity is $O(E * \log E + E * \log V)$ time.
- The value of E can be at most $O(V^2)$, so $O(\log V)$ and $O(\log E)$ are the same. Therefore, the overall time complexity is $O(E * \log E)$ or $O(E * \log V)$

Auxiliary Space: $O(E+V)$, where V is the number of vertices and E is the number of edges in the graph

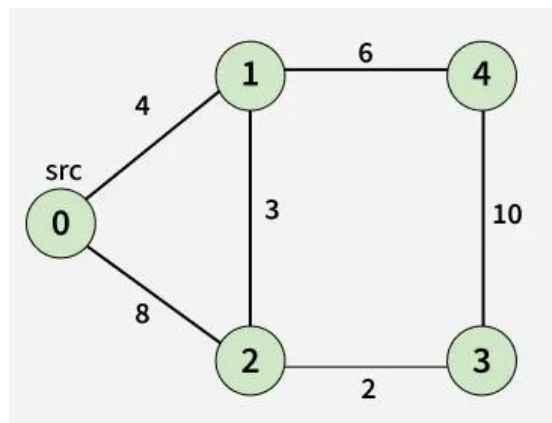
Dijkstra's algorithm for single source shortest path problem:

Given a **weighted undirected** graph and a source vertex **src**. We need to find the shortest path distances from the source vertex to all other vertices in the graph.

Note: The given graph does not contain any negative edge.

Examples:

Input: $\text{src} = 0$, $\text{adj}[][] = [[1, 4], [2, 8]],$
 $[[0, 4], [4, 6], [2, 3]],$
 $[[0, 8], [3, 2], [1, 3]],$
 $[[2, 2], [4, 10]],$
 $[[1, 6], [3, 10]]]$



Output: $[0, 4, 7, 9, 10]$

Explanation: Shortest Paths:

$0 \rightarrow 0 = 0$: Source node itself, so distance is 0.

$0 \rightarrow 1 = 4$: Direct edge from node 0 to 1 gives shortest distance 4.

$0 \rightarrow 2 = 7$: Path $0 \rightarrow 1 \rightarrow 2$ gives total cost $4 + 3 = 7$, which is smaller than direct edge 8.

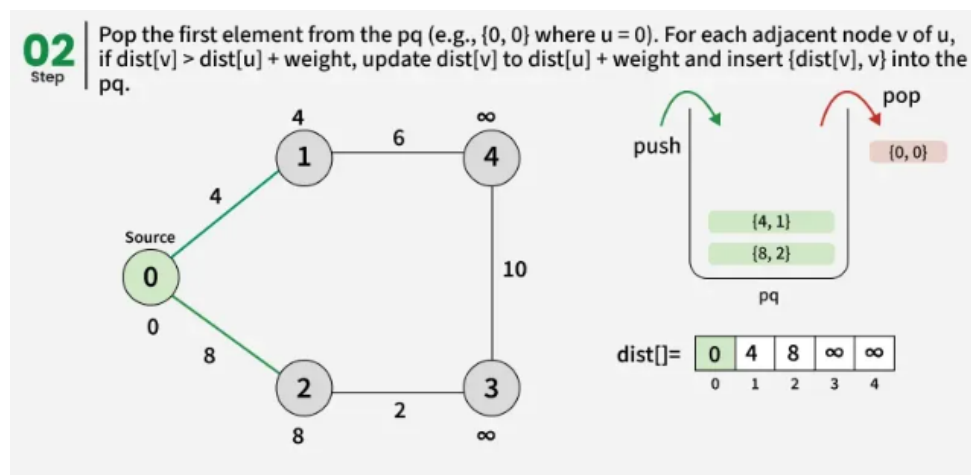
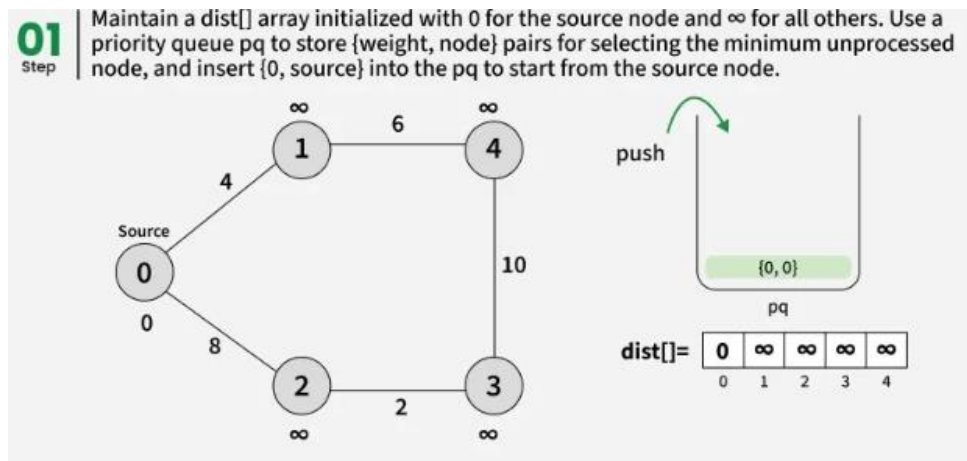
$0 \rightarrow 3 = 9$: Path $0 \rightarrow 1 \rightarrow 2 \rightarrow 3$ gives total cost $4 + 3 + 2 = 9$.

$0 \rightarrow 4 = 10$: Path $0 \rightarrow 1 \rightarrow 4$ gives total cost $4 + 6 = 10$.

Detailed Steps:

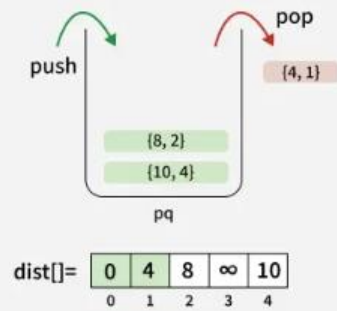
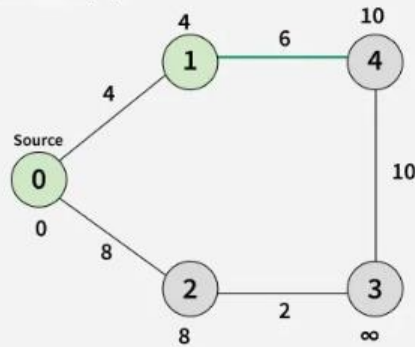
- Create a distance array $\text{dist}[]$ of size V and initialize all values to infinity (∞) since no paths are known yet.

- Set the distance of the source vertex to 0 and insert it into the priority queue.
- While the priority queue is not empty, remove the vertex with the smallest distance value.
- Check: if the popped distance is greater than the recorded distance for this vertex($\text{dist}[u]$), it means this vertex has already been processed with a smaller distance, so skip it and continue to the next iteration.
- For each neighbor v of u , check if the path through u gives a smaller distance than the current $\text{dist}[v]$. If it does, update $\text{dist}[v] = \text{dist}[u] + \text{edge weight}(d)$ and push $(\text{dist}[v], v)$ into the priority queue.
- Continue this process until the priority queue becomes empty.
- Once done, the $\text{dist}[]$ array will contain the shortest distance from the source to every vertex in the graph.



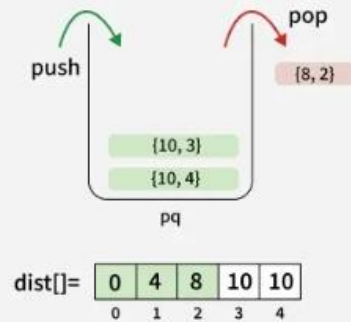
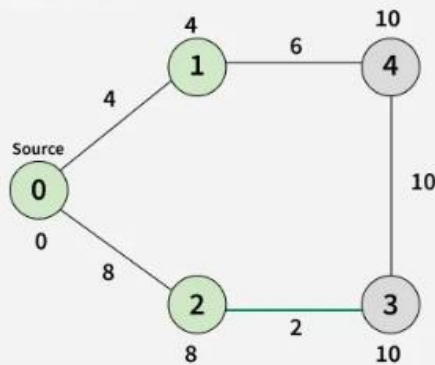
03
Step

Pop the first element from the pq (e.g., {4, 1} where $u = 1$). For each adjacent node v of u , if $\text{dist}[v] > \text{dist}[u] + \text{weight}$, update $\text{dist}[v]$ to $\text{dist}[u] + \text{weight}$ and insert $\{\text{dist}[v], v\}$ into the pq.



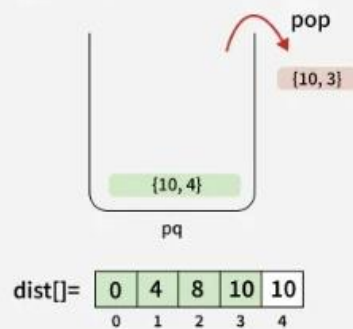
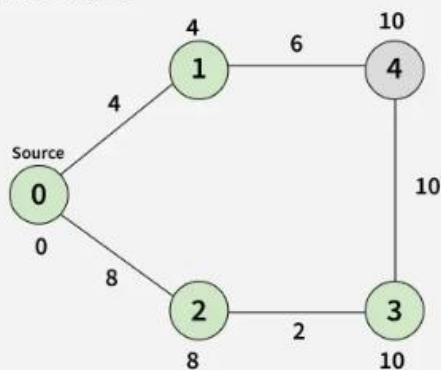
04
Step

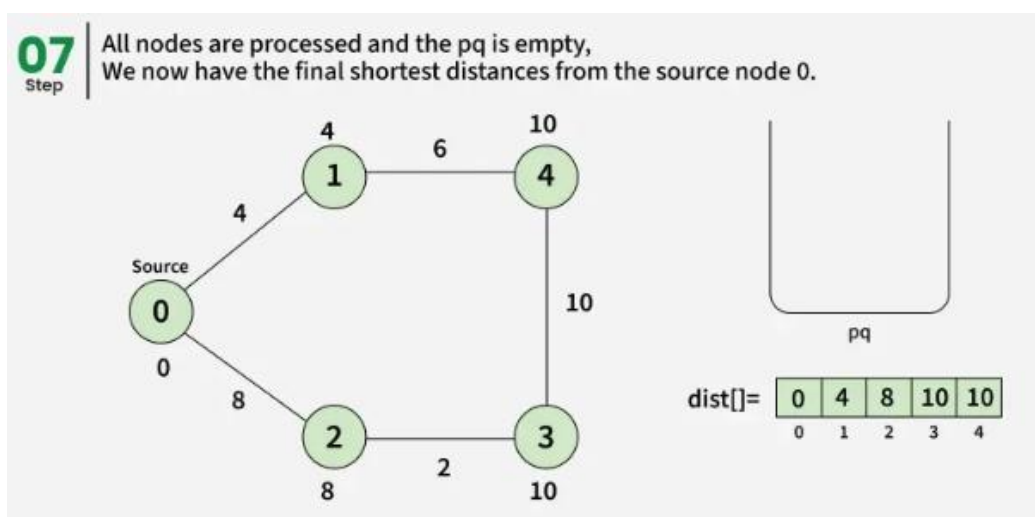
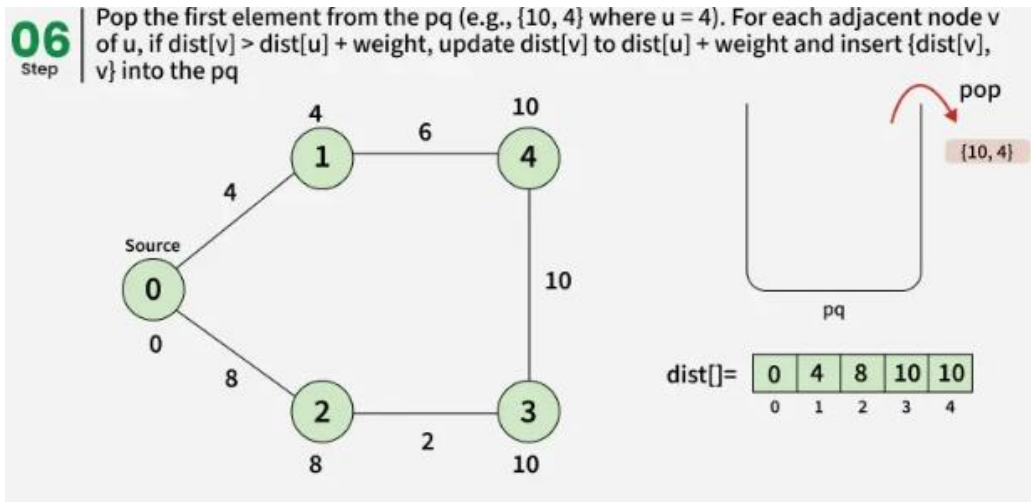
Pop the first element from the pq (e.g., {8, 2} where $u = 2$). For each adjacent node v of u , if $\text{dist}[v] > \text{dist}[u] + \text{weight}$, update $\text{dist}[v]$ to $\text{dist}[u] + \text{weight}$ and insert $\{\text{dist}[v], v\}$ into the pq.



05
Step

Pop the first element from the set (e.g., {10, 3} where $u = 3$). For each adjacent node v of u , if $\text{dist}[v] > \text{dist}[u] + \text{weight}$, update $\text{dist}[v]$ to $\text{dist}[u] + \text{weight}$ and insert $\{\text{dist}[v], v\}$ into the pq.





Time Complexity: $O((V+E) \cdot \log V)$, Where E is the number of edges and V is the number of vertices.

Auxiliary Space: $O(V+E)$

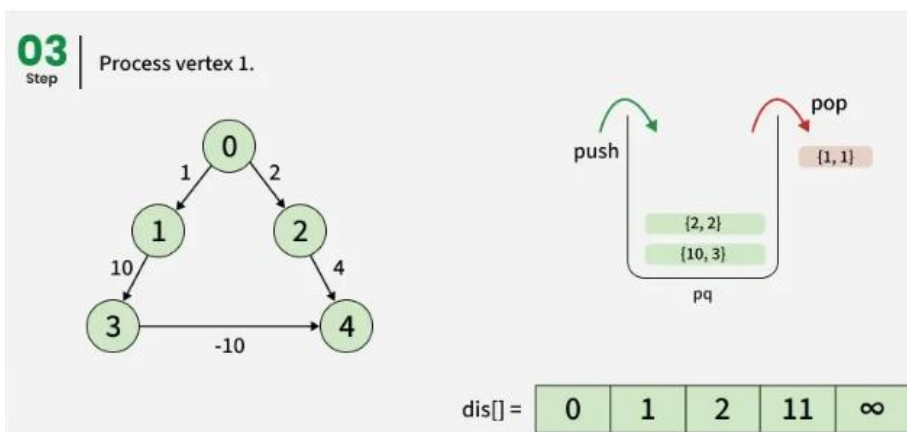
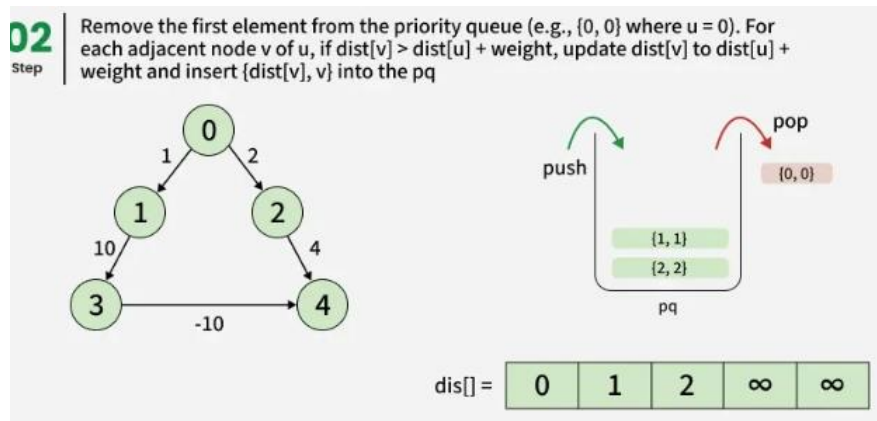
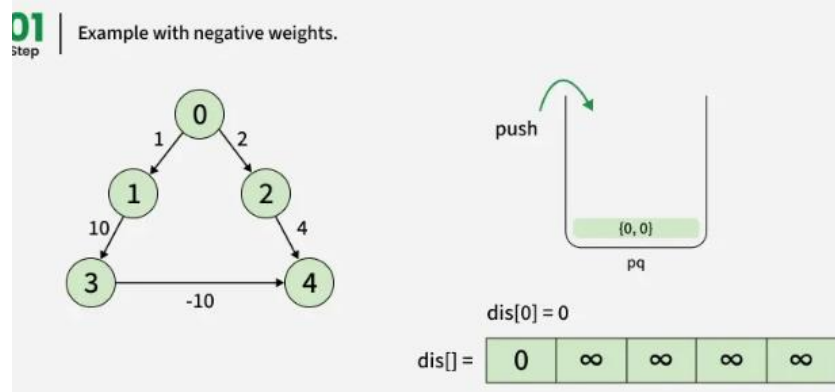
How Does it Work?

Once we pop the minimum distance item from the priority queue, we finalize its shortest distance and never consider it again. Assuming that there are no negative weight edges, if there were a shorter path to this node through any other route, that path would have to go through nodes with equal or greater than current distance, and so wouldn't be shorter

Why Does Not Work with Negative Weights:

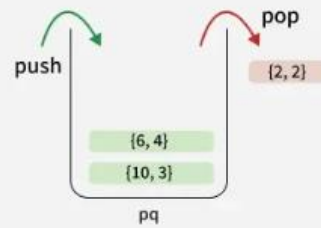
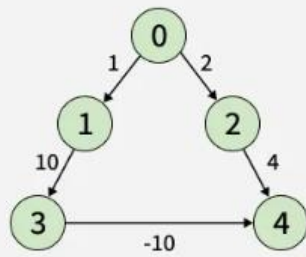
Dijkstra's algorithm assumes that once a vertex u is picked from the priority queue (meaning it currently has the smallest distance), its shortest distance is finalized - it will never change in the future. This assumption is true only if all edge weights are non-negative.

Suppose there exists an edge with a negative weight. After Dijkstra finalizes a vertex u , there might exist another path through some vertex v (processed later) that leads back to u with a smaller total distance because of the negative edge. So, the algorithm's earlier assumption that $\text{dist}[u]$ was final is no longer valid.



04
Step

Process vertex 2.

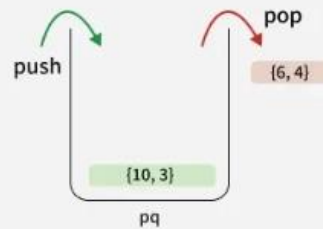
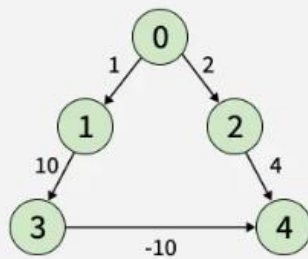


dis[] =

0	1	2	11	6
---	---	---	----	---

05
Step

Process vertex 4.

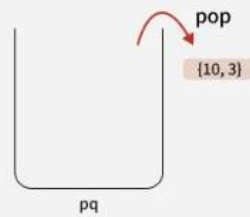
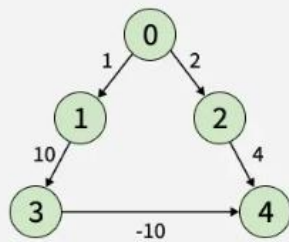


dis[] =

0	1	2	11	6
---	---	---	----	---

06
Step

Process vertex 3.



Node 4 is already processed, which means its value should never change. But because of the negative edge, node 3 updates its distance to 1, and that allows the node 4 to be processed again, which breaks Dijkstra's rule and causes the algorithm to fail.

UNIT IV

Sorting:

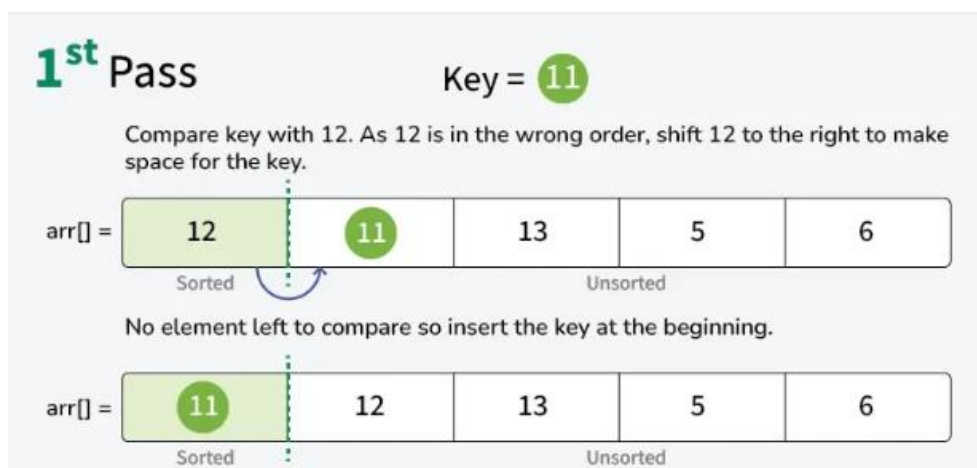
A Sorting Algorithm is used to rearrange a given array or list of elements in an order. For example, a given array [10, 20, 5, 2] becomes [2, 5, 10, 20] after sorting in increasing order and becomes [20, 10, 5, 2] after sorting in decreasing order.

- There exist different sorting algorithms for different different types of inputs, for example a binary array, a character array, an array with a large range of values or an array with many duplicates or a small vs large array.
- The algorithms may also differ according to output requirements. For example, stable sorting (or maintains original order of equal elements) or not stable.
- Sorting is provided in library implementation of most of the programming languages. These sorting functions typically are general purpose functions with flexibility of providing the expected sorting order (increasing or decreasing or by a specific key in case of objects)

Insertion sort :

Insertion sort is a simple sorting algorithm that works by iteratively inserting each element of an unsorted list into its correct position in a sorted portion of the list. It is like sorting playing cards in your hands. You split the cards into two groups: the sorted cards and the unsorted cards. Then, you pick a card from the unsorted group and put it in the right place in the sorted group.

- Start with the second element as the first element is assumed to be sorted.
- Compare the second element with the first if the second is smaller then swap them.
- Move to the third element, compare it with the first two, and put it in its correct position
- Repeat until the entire array is sorted.



2nd Pass

Key = 13

Compare key with 12, 12 is in the correct order, so no changes take place.



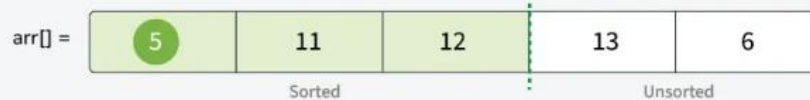
3rd Pass

Key = 5

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.



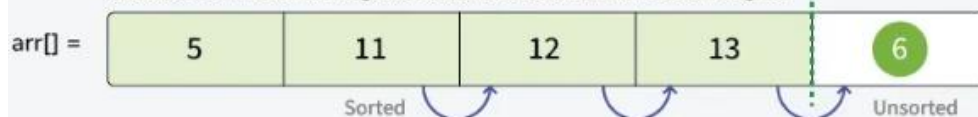
No element left to compare, so insert key at the beginning.



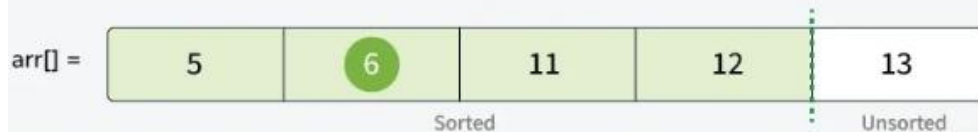
4th Pass

Key = 6

Compare key with all the elements in the sorted subarray starting with 13, if the element is in the wrong order, shift that element to the right.

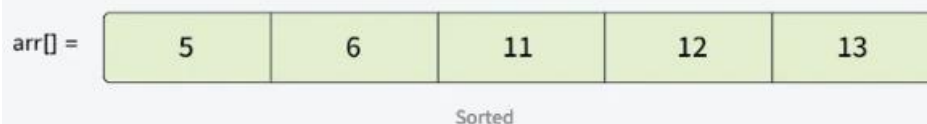


5 is in the correct order, so shifting stops. Insert key after 5.



Sorted Array

The sorted part contains the whole array. Means that the whole array is sorted.



/ Java program for implementation of Insertion Sort

```
public class InsertionSort {  
    /* Function to sort array using insertion sort */  
    void sort(int arr[])  
    {  
        int n = arr.length;  
        for (int i = 1; i < n; ++i) {  
            int key = arr[i];  
            int j = i - 1;  
            /* Move elements of arr[0..i-1], that are  
            greater than key, to one position ahead  
            of their current position */  
            while (j >= 0 && arr[j] > key) {  
                arr[j + 1] = arr[j];  
                j = j - 1;  
            }  
            arr[j + 1] = key;  
        }  
    }  
    /* A utility function to print array of size n */  
    static void printArray(int arr[])  
    {  
        int n = arr.length;  
        for (int i = 0; i < n; ++i)  
            System.out.print(arr[i] + " ");  
        System.out.println();  
    }  
  
    // Driver method  
    public static void main(String args[])
```

```

{
    int arr[] = { 12, 11, 13, 5, 6 };

    InsertionSort ob = new InsertionSort();

    ob.sort(arr);

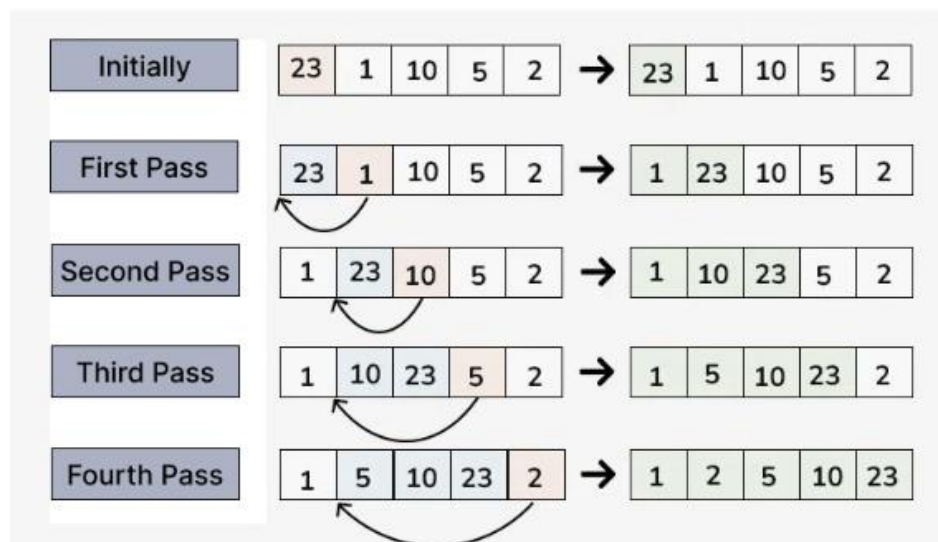
    printArray(arr);
}
}

```

Output

5 6 11 12 13

Illustration



arr = {23, 1, 10, 5, 2}

Initial:

- Current element is **23**
- The first element in the array is assumed to be sorted.
- The sorted part until **0th** index is : **[23]**

First Pass:

- Compare **1** with **23** (current element with the sorted part).
- Since **1** is smaller, insert **1** before **23**.
- The sorted part until **1st** index is: **[1, 23]**

Second Pass:

- Compare **10** with **1** and **23** (current element with the sorted part).
- Since **10** is greater than **1** and smaller than **23**, insert **10** between **1** and **23**.
- The sorted part until **2nd** index is: **[1, 10, 23]**

Third Pass:

- Compare **5** with **1**, **10**, and **23** (current element with the sorted part).
- Since **5** is greater than **1** and smaller than **10**, insert **5** between **1** and **10**.
- The sorted part until **3rd** index is : **[1, 5, 10, 23]**

Fourth Pass:

- Compare 2 with 1, 5, 10, and 23 (current element with the sorted part).
- Since 2 is greater than 1 and smaller than 5 insert 2 between 1 and 5.
- The sorted part until 4th index is: [1, 2, 5, 10, 23]

Final Array:

- The sorted array is: [1, 2, 5, 10, 23]

Complexity Analysis of Insertion Sort**Time Complexity**

- **Best case: $O(n)$** , If the list is already sorted, where n is the number of elements in the list.
- **Average case: $O(n^2)$** , If the list is randomly ordered
- **Worst case: $O(n^2)$** , If the list is in reverse order

Space Complexity

- **Auxiliary Space: $O(1)$** , Insertion sort requires **$O(1)$** additional space, making it a space-efficient sorting algorithm.

Please refer Complexity Analysis of Insertion Sort for details.

Advantages and Disadvantages of Insertion Sort

Advantages

- Simple and easy to implement.
- **Stable** sorting algorithm.
- Efficient for small lists and nearly sorted lists.
- Space-efficient as it is an in-place algorithm.
- Adoptive. the number of inversions is directly proportional to number of swaps. For example, no swapping happens for a sorted array and it takes $O(n)$ time only.

Disadvantages

- Inefficient for large lists.
- Not as efficient as other sorting algorithms (e.g., merge sort, quick sort) for most cases.

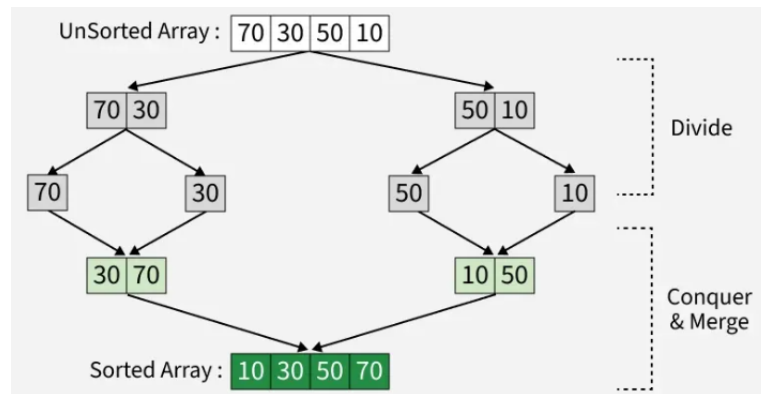
Applications of Insertion Sort

Insertion sort is commonly used in situations where:

- The list is small or nearly sorted.
- Simplicity and stability are important.
- Used as a subroutine in Bucket Sort
- Can be useful when array is already almost sorted (very few inversions)
- Since Insertion sort is suitable for small sized arrays, it is used in Hybrid Sorting algorithms along with other efficient algorithms like Quick Sort and Merge Sort. When the subarray size becomes small, we switch to insertion sort in these recursive algorithms. For example IntroSort and TimSort use insertions sort.

Merge sort:

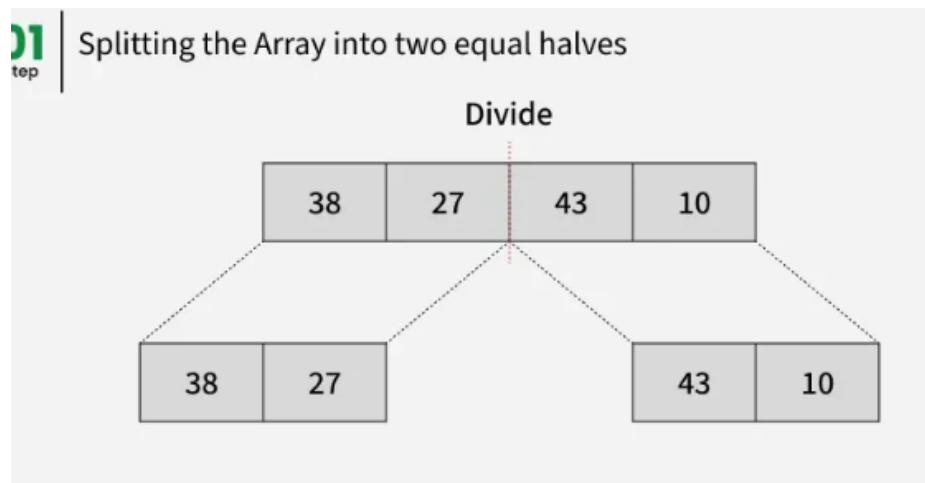
Merge sort is a popular sorting algorithm known for its efficiency and stability. It follows the Divide and Conquer approach. It works by recursively dividing the input array into two halves, recursively sorting the two halves and finally merging them back together to obtain the sorted array.



Here's a step-by-step explanation of how merge sort works:

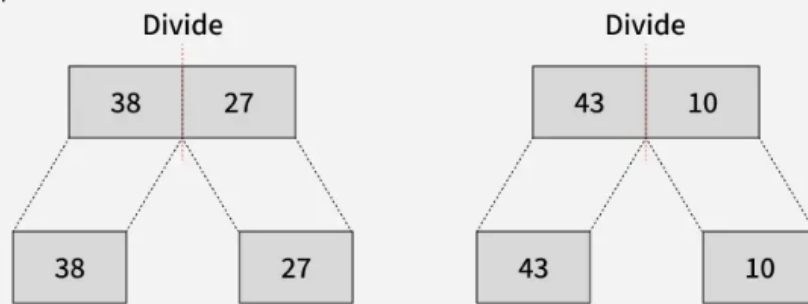
1. **Divide:** Divide the list or array recursively into two halves until it can no more be divided.
2. **Conquer:** Each subarray is sorted individually using the merge sort algorithm.
3. **Merge:** The sorted subarrays are merged back together in sorted order. The process continues until all elements from both subarrays have been merged.

Let's sort the array or list [38, 27, 43, 10] using Merge Sort



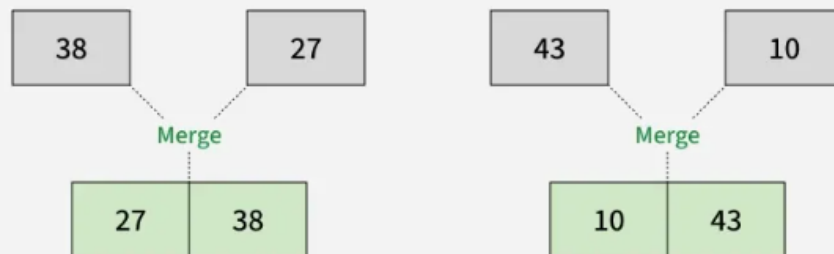
02
Step

Splitting the subarrays into two halves



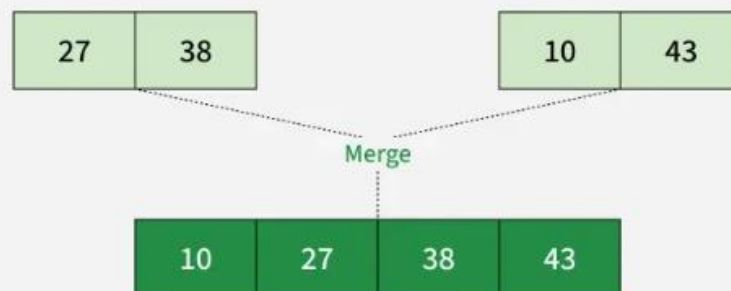
03
Step

Merging unit length cells into sorted subarrays



04
Step

Merging sorted subarrays into the sorted array



Let's look at the working of above example:

Divide:

- [38, 27, 43, 10] is divided into [38, 27] and [43, 10] .
- [38, 27] is divided into [38] and [27] .
- [43, 10] is divided into [43] and [10] .

Conquer:

- [38] is already sorted.
- [27] is already sorted.
- [43] is already sorted.

- [10] is already sorted.

Merge:

- Merge [38] and [27] to get [27, 38] .
- Merge [43] and [10] to get [10,43] .
- Merge [27, 38] and [10,43] to get the final sorted list [10, 27, 38, 43]

Therefore, the sorted list is [10, 27, 38, 43] .

```
import java.io.*;
```

```
class GfG {
```

```
    // Merges two subarrays of arr[].
```

```
    // First subarray is arr[l..m]
```

```
    // Second subarray is arr[m+1..r]
```

```
    static void merge(int arr[], int l, int m, int r){
```

```
        // Find sizes of two subarrays to be merged
```

```
        int n1 = m - l + 1;
```

```
        int n2 = r - m;
```

```
        // Create temp arrays
```

```
        int L[] = new int[n1];
```

```
        int R[] = new int[n2];
```

```
        // Copy data to temp arrays
```

```
        for (int i = 0; i < n1; ++i)
```

```
            L[i] = arr[l + i];
```

```
        for (int j = 0; j < n2; ++j)
```

```
            R[j] = arr[m + 1 + j];
```

```
        // Merge the temp arrays
```

```
        // Initial indices of first and second subarrays
```

```
        int i = 0, j = 0;
```

```
        // Initial index of merged subarray array
```

```
        int k = l;
```

```
        while (i < n1 && j < n2) {
```

```
            if (L[i] <= R[j]) {
```

```
                arr[k] = L[i];
```

```

        i++;
    }
    else {
        arr[k] = R[j];
        j++;
    }
    k++;
}

// Copy remaining elements of L[] if any
while (i < n1) {
    arr[k] = L[i];
    i++;
    k++;
}

// Copy remaining elements of R[] if any
while (j < n2) {
    arr[k] = R[j];
    j++;
    k++;
}
}

// Main function that sorts arr[l..r] using
// merge()
static void mergeSort(int arr[], int l, int r){
    if (l < r) {
        // Find the middle point
        int m = l + (r - l) / 2;

        // Sort first and second halves
        mergeSort(arr, l, m);
        mergeSort(arr, m + 1, r);
    }
}

```

```

        // Merge the sorted halves
        merge(arr, l, m, r);
    }
}

// Driver code

public static void main(String args[]){
    int arr[] = {38, 27, 43, 10};
    mergeSort(arr, 0, arr.length - 1);
    int n = arr.length;
    for (int i = 0; i < n; ++i)
        System.out.print(arr[i] + " ");
    System.out.println();
}
}

```

Output

```
10 27 38 43
```

Recurrence Relation of Merge Sort:

The recurrence relation of merge sort is:

$$T(n) = \begin{cases} \Theta(1) & \text{if } n=1 \\ 2T(n/2) + \Theta(n) & \text{if } n > 1 \end{cases}$$

- $T(n)$ Represents the total time taken by the algorithm to sort an array of size n .
- $2T(n/2)$ represents time taken by the algorithm to recursively sort the two halves of the array. Since each half has $n/2$ elements, we have two recursive calls with input size as $(n/2)$.
- $O(n)$ represents the time taken to merge the two sorted halves

Complexity Analysis of Merge Sort

Time Complexity:

- **Best Case:** $O(n \log n)$, When the array is already sorted or nearly sorted.
- **Average Case:** $O(n \log n)$, When the array is randomly ordered.
- **Worst Case:** $O(n \log n)$, When the array is sorted in reverse order.

Auxiliary Space: $O(n)$, Additional space is required for the temporary array used during merging.

Applications of Merge Sort:

- Sorting large datasets
- External sorting (when the dataset is too large to fit in memory)
- Used to solve problems like Inversion counting, Count Smaller on Right & Surpasser Count
- Merge Sort and its variations are used in library methods of programming languages. Its variation Tim Sort is used in Python, Java Android and Swift. The main reason why it is preferred to sort non-primitive types is stability which is not there in QuickSort. Arrays. sort in Java uses Quick Sort while Collections. sort uses Merge Sort.
- It is a preferred algorithm for sorting Linked lists.
- It can be easily parallelized as we can independently sort subarrays and then merge.
- The merge function of merge sort to efficiently solve the problems like union and intersection of two sorted arrays.

Advantages and Disadvantages of Merge Sort

Advantages

- **Stability** : Merge sort is a stable sorting algorithm, which means it maintains the relative order of equal elements in the input array.
- **Guaranteed worst-case performance**: Merge sort has a worst-case time complexity of $O(N \log N)$, which means it performs well even on large datasets.
- **Simple to implement**: The divide-and-conquer approach is straightforward.
- **Naturally Parallel** : We independently merge subarrays that makes it suitable for parallel processing.

Disadvantages

- **Space complexity**: Merge sort requires additional memory to store the merged sub-arrays during the sorting process.
- **Not in-place**: Merge sort is not an in-place sorting algorithm, which means it requires additional memory to store the sorted data. This can be a disadvantage in applications where memory usage is a concern.
- Merge Sort is Slower than QuickSort in general as QuickSort is more cache friendly because it works in-place.

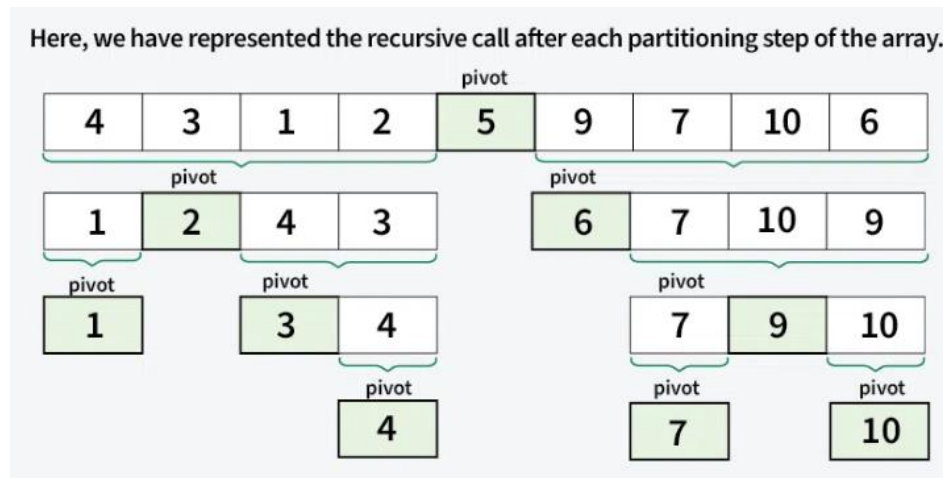
Quick sort:

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. .

There are mainly three steps in the algorithm:

1. **Choose a Pivot**: Select an element from the array as the pivot. The choice of pivot can vary (e.g., first element, last element, random element, or median).
2. **Partition the Array**: Re arrange the array around the pivot. After partitioning, all elements smaller than the pivot will be on its left, and all elements greater than the pivot will be on its right.
3. **Recursively Call**: Recursively apply the same process to the two partitioned sub-arrays.

4. **Base Case:** The recursion stops when there is only one element left in the sub-array, as a single element is already sorted.



Choice of Pivot

There are many different choices for picking pivots.

- Always pick the first (or last) element as a pivot. The below implementation picks the last element as pivot. The problem with this approach is it ends up in the worst case when array is already sorted.
- Pick a random element as a pivot. This is a preferred approach because it does not have a pattern for which the worst case happens.
- Pick the median element is pivot. This is an ideal approach in terms of time complexity as we can find median in linear time and the partition function will always divide the input array into two halves. But it takes more time on average as median finding has high constants.

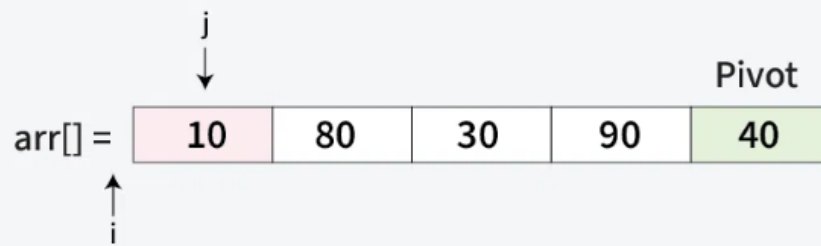
Partition Algorithm

The key process in **quickSort** is a **partition()**. There are three common algorithms to partition. All these algorithms have $O(n)$ time complexity.

1. **Naive Partition:** Here we create copy of the array. First put all smaller elements and then all greater. Finally we copy the temporary array back to original array. This requires $O(n)$ extra space.
2. **Lomuto Partition:** We have used this partition in this article. This is a simple algorithm, we keep track of index of smaller elements and keep swapping. We have used it here in this article because of its simplicity.
3. **Hoare's Partition:** This is the fastest of all. Here we traverse array from both sides and keep swapping greater element on left with smaller on right while the array is not partitioned.

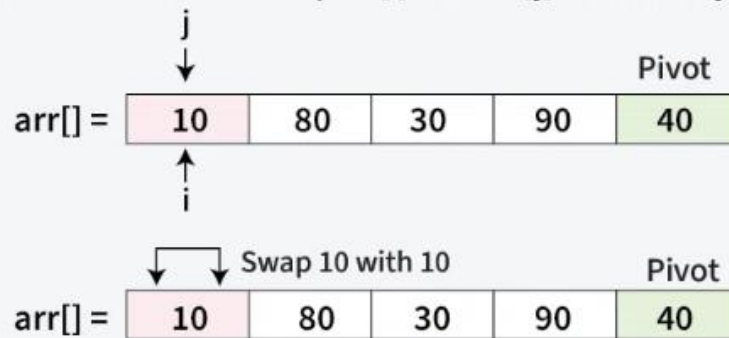
01
Step

Pivot Selection: The last element $\text{arr}[4] = 40$ is chosen as the pivot.
Initial Pointers: $i = -1$ and $j = 0$.



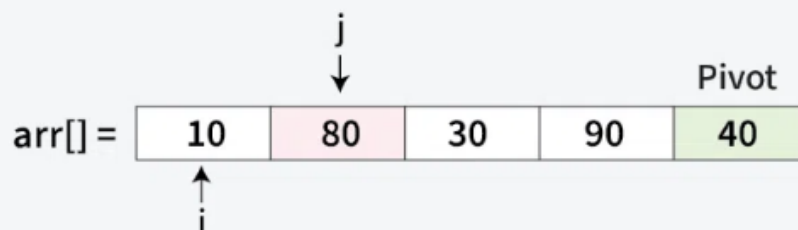
02
Step

Since, $\text{arr}[j] < \text{pivot}$ ($10 < 40$)
Increment i to 0 and swap $\text{arr}[i]$ with $\text{arr}[j]$. Increment j by 1



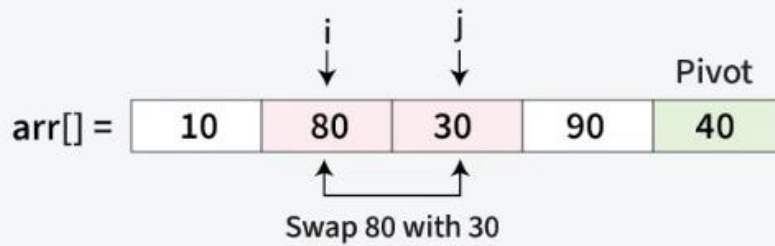
03
Step

Since, $\text{arr}[j] > \text{pivot}$ ($80 > 40$)
No swap needed. Increment j by 1



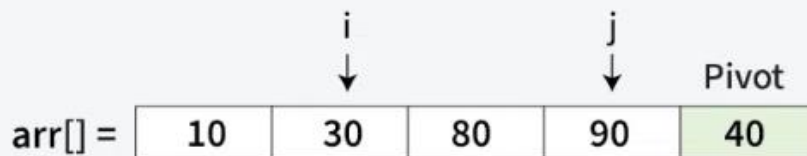
04
Step

Since, $\text{arr}[j] < \text{pivot}$ ($30 < 40$)
Increment i by 1 and swap $\text{arr}[i]$ with $\text{arr}[j]$. Increment j by 1



05
Step

Since, $\text{arr}[j] > \text{pivot}$ ($90 > 40$)
No swap needed. Increment j by 1



06
Step

Since traversal of j has ended. Now move pivot to its correct position, Swap $\text{arr}[i + 1] = \text{arr}[2]$ with $\text{arr}[4] = 40$.

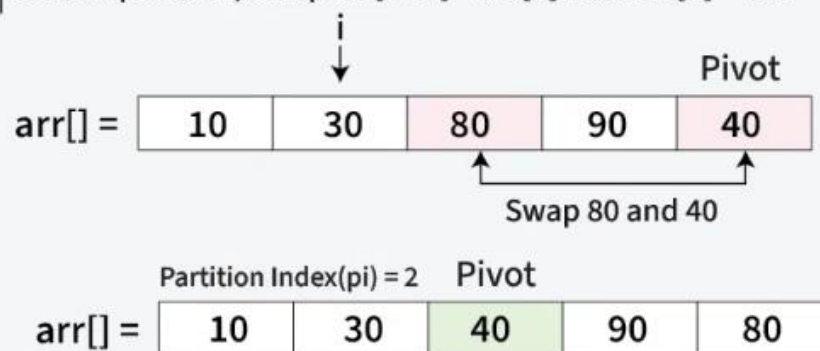
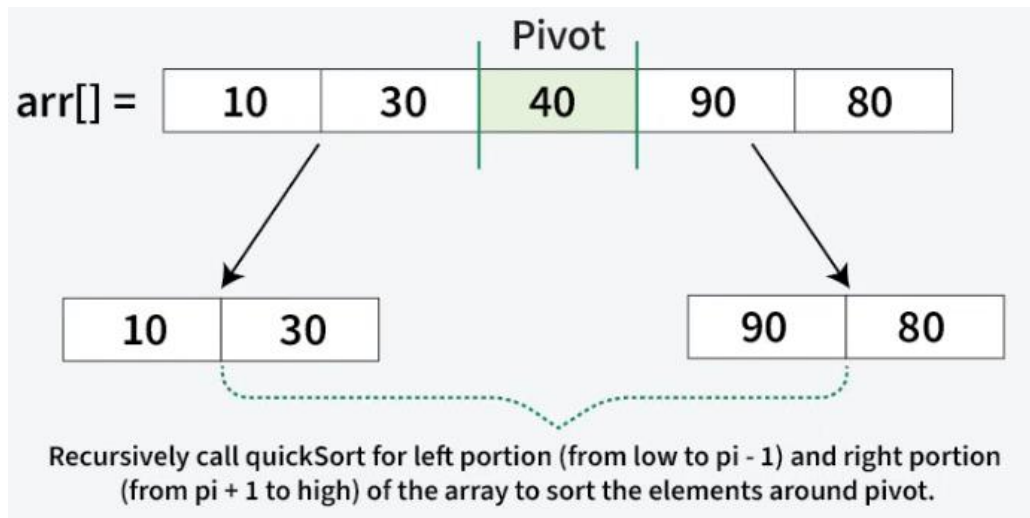


Illustration of QuickSort Algorithm

In the previous step, we looked at how the **partitioning** process rearranges the array based on the chosen **pivot**. Next, we apply the same method recursively to the smaller sub-arrays on the **left** and **right** of the pivot. Each time, we select new pivots and partition the arrays again. This process continues until only one element is left, which is always sorted. Once every element is in its correct position, the entire array is sorted.

Below image illustrates, how the recursive method calls for the smaller sub-arrays on the **left** and **right** of the **pivot**:



```
import java.util.Arrays;

class GfG {
    // partition function
    static int partition(int[] arr, int low, int high) {
        // choose the pivot
        int pivot = arr[high];

        // index of smaller element and indicates
        // the right position of pivot found so far
        int i = low - 1;

        // traverse arr[low..high] and move all smaller
        // elements to the left side. Elements from low to
        // i are smaller after every iteration
        for (int j = low; j <= high - 1; j++) {
            if (arr[j] < pivot) {
                i++;
                swap(arr, i, j);
            }
        }
    }
}
```

```

        // Move pivot after smaller elements and
        // return its position
        swap(arr, i + 1, high);
        return i + 1;
    }

    // swap function
    static void swap(int[] arr, int i, int j) {
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }

    // the QuickSort function implementation
    static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // pi is the partition return index of pivot
            int pi = partition(arr, low, high);

            // recursion calls for smaller elements
            // and greater or equals elements
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }

    public static void main(String[] args) {
        int[] arr = {10, 7, 8, 9, 1, 5};
        int n = arr.length;
        quickSort(arr, 0, n - 1);
        for (int val : arr) {
            System.out.print(val + " ");
        }
    }
}

```

}

Output

```
1 5 7 8 9 10
```

Complexity Analysis of Quick Sort

Time Complexity:

- **Best Case:** ($\Omega(n \log n)$), Occurs when the pivot element divides the array into two equal halves.
- **Average Case** ($\theta(n \log n)$), On average, the pivot divides the array into two parts, but not necessarily equal.
- **Worst Case:** ($O(n^2)$), Occurs when the smallest or largest element is always chosen as the pivot (e.g., sorted arrays).

Auxiliary Space:

- **Worst-case scenario: $O(n)$** due to unbalanced partitioning leading to a skewed recursion tree requiring a call stack of size $O(n)$.
- **Best-case scenario: $O(\log n)$** as a result of balanced partitioning leading to a balanced recursion tree with a call stack of size $O(\log n)$.

Advantages of Quick Sort

- 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.
- It is Cache Friendly as we work on the same array to sort and do not copy data to any auxiliary array.
- Fastest general purpose algorithm for large data when stability is not required.
- It is tail recursive and hence all the tail call optimization can be done.

Disadvantages of Quick Sort

- It has a worst-case time complexity of $O(n^2)$, which occurs when the pivot is chosen poorly.
- It is not a good choice for small data sets.
- It is not a stable sort, meaning that if two elements have the same key, their relative order will not be preserved in the sorted output in case of quick sort, because here we are swapping elements according to the pivot's position (without considering their original positions).

Applications of Quick Sort

- Sorting large datasets efficiently in memory.

- Used in library sort functions (like C++ `std::sort` and Java `Arrays.sort` for primitives).
- Arranging records in databases for faster searching.
- Preprocessing step in algorithms requiring sorted input (e.g., binary search, two-pointer techniques).
- Finding the kth smallest/largest element using Quickselect (a variant of quicksort).
- Sorting arrays of objects based on multiple keys (custom comparators).
- Data compression algorithms (like Huffman coding preprocessing).
- Graphics and computational geometry (e.g., convex hull algorithms).

Heap sort:

Heap Sort is a comparison-based sorting algorithm based on the **Binary Heap** data structure.

- It is an optimized version of selection sort.
- The algorithm repeatedly finds the maximum (or minimum) element and swaps it with the last (or first) element.
- Using a binary heap allows efficient access to the max (or min) element in $O(\log n)$ time instead of $O(n)$.
- The process is repeated for the remaining elements until the array is sorted.
- Overall, Heap Sort achieves a time complexity of $O(n \log n)$.

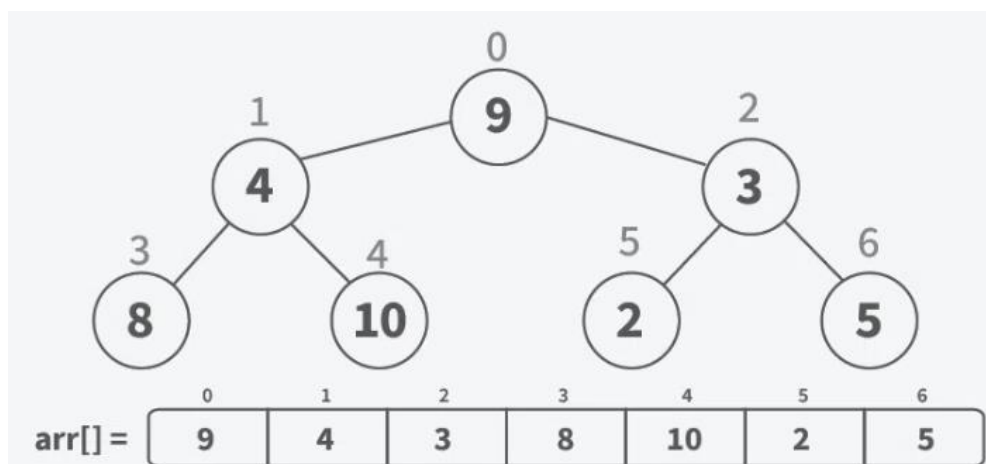
Heap Sort Algorithm :

First convert the array into a [max heap](#) using heapify, Please note that this happens in-place. The array elements are re-arranged to follow heap properties. Then one by one delete the root node of the Max-heap and replace it with the last node and heapify. Repeat this process while size of heap is greater than 1.

Detailed Working of Heap Sort

Step 1: Treat the Array as a Complete Binary Tree

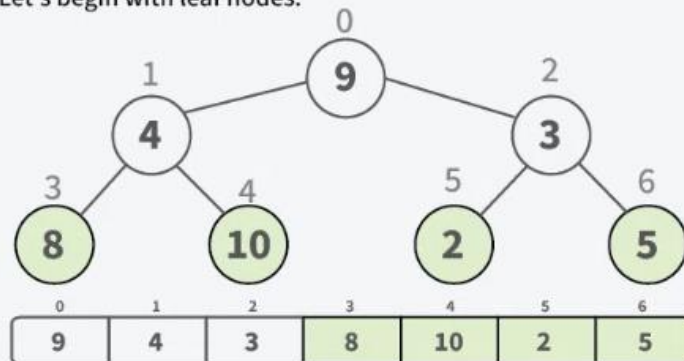
We first need to visualize the array as a complete binary tree. For an array of size n , the root is at index 0, the left child of an element at index i is at $2i + 1$, and the right child is at $2i + 2$.



Step 2: Build a Max Heap

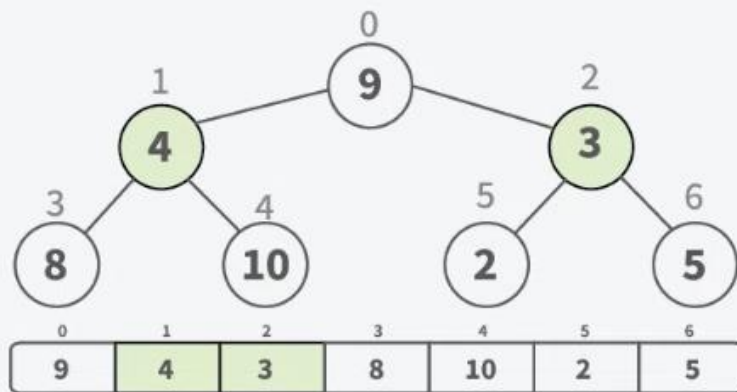
01
Step

Compare each node with its children, ensuring parent nodes are larger. This causes smaller nodes to bubble down and larger nodes to rise to the top. Let's begin with leaf nodes.



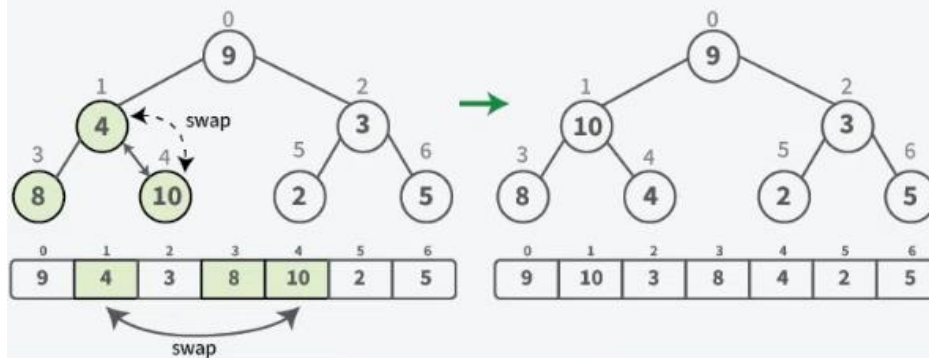
02
Step

Let's look in the next upper level (node 4 and 3)



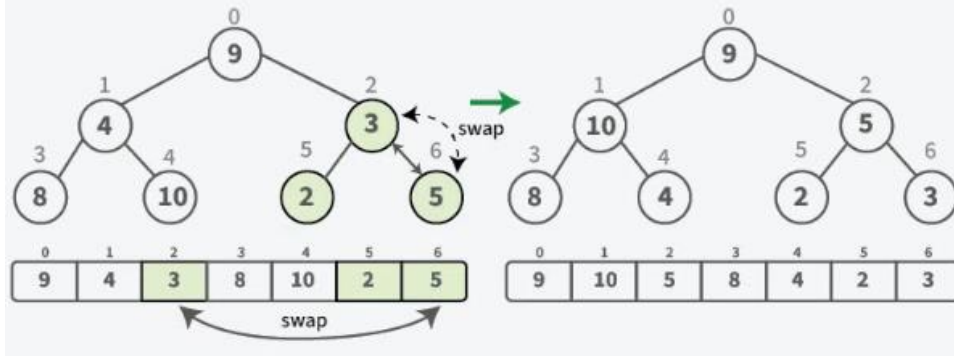
03
Step

Node 4 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.

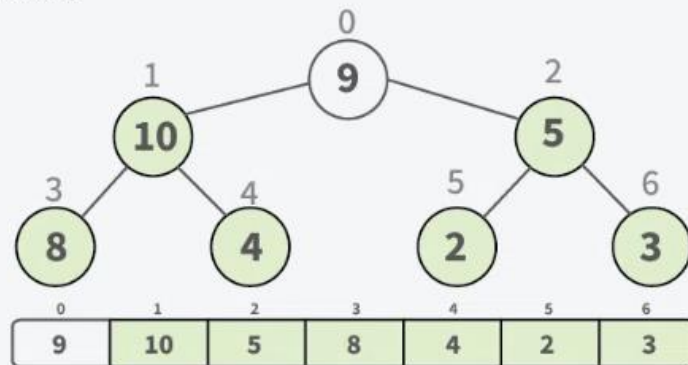


Heapify Binary Tree

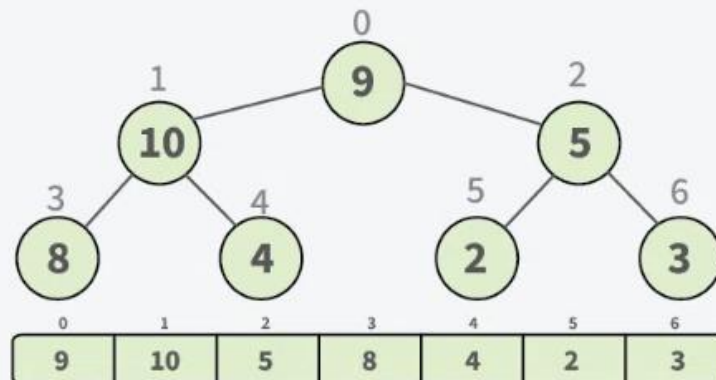
04 Step | Check for the next node (3) in current level. Since, it has a larger child, so swap it to ensure the parent has a larger value.



05 Step | After the completion of current level, we have now two smaller valid max heap

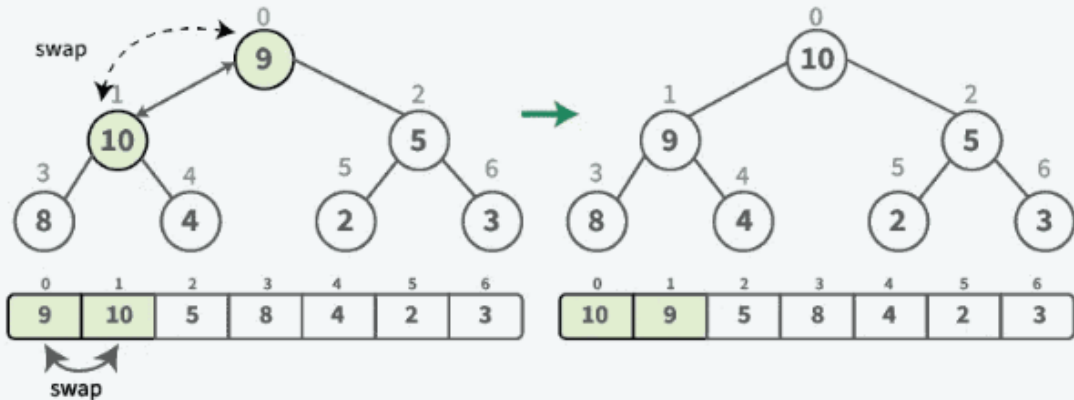


06 Step | Let's move to the next upper level. Here we've node 9 at the root.



07
Step

Node 9 is smaller than its child node (10), so swap it with the larger child to maintain the property that the parent should be larger than its children.

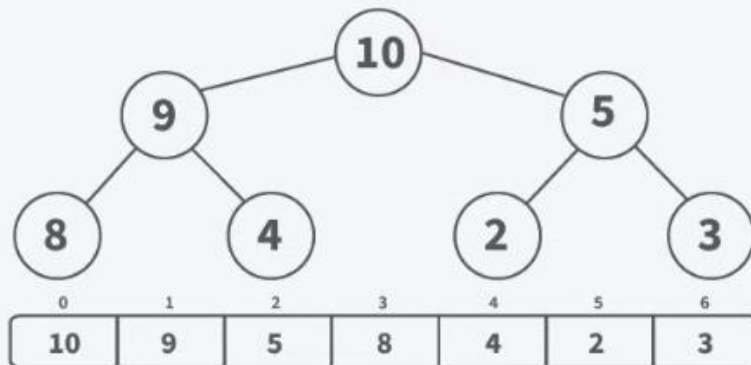


Heapify Binary Tree

Step 3: Sort the array by placing largest element at end of unsorted array.

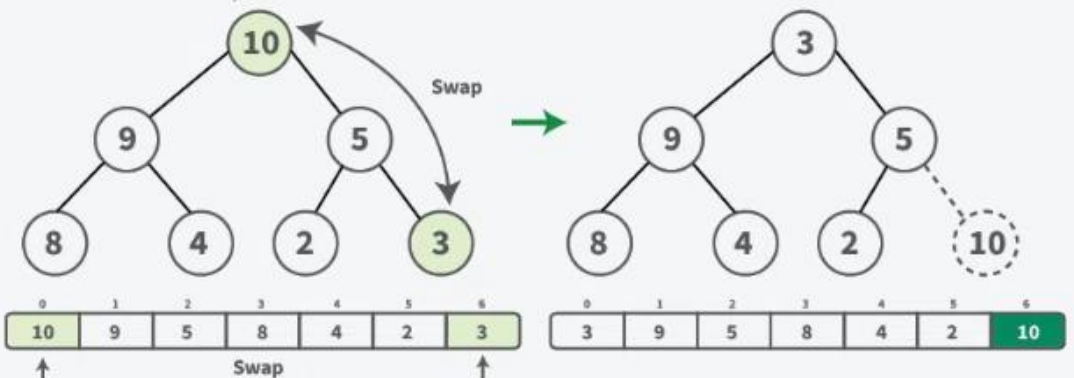
01
Step

Let's assume we have transformed the given array to follow the max heap property. Here's how our array would look in max heap form.



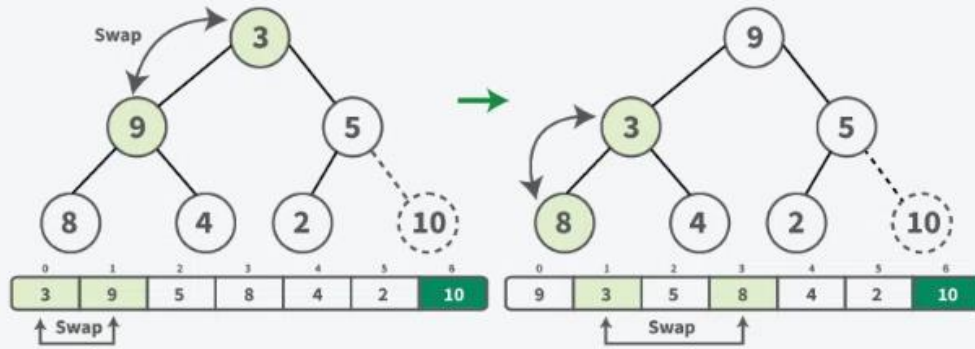
02
Step

Swap the maximum element (10) with the last element (3) in the unsorted array. Decrease the size of the heap by one (ignore the last element, as it is now sorted).



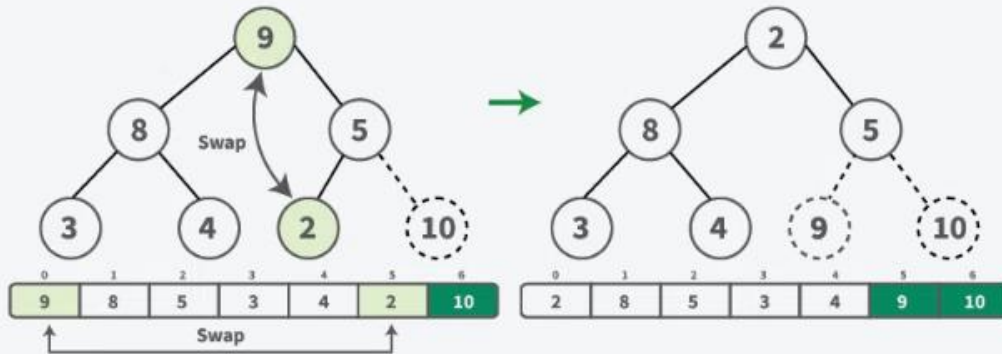
03
Step

Now, root violate the max-heap property, So, heapify it.
Swap node 3 with its largest child (9).
Still node 3 have larger children, so swap it with largest one (node 8).



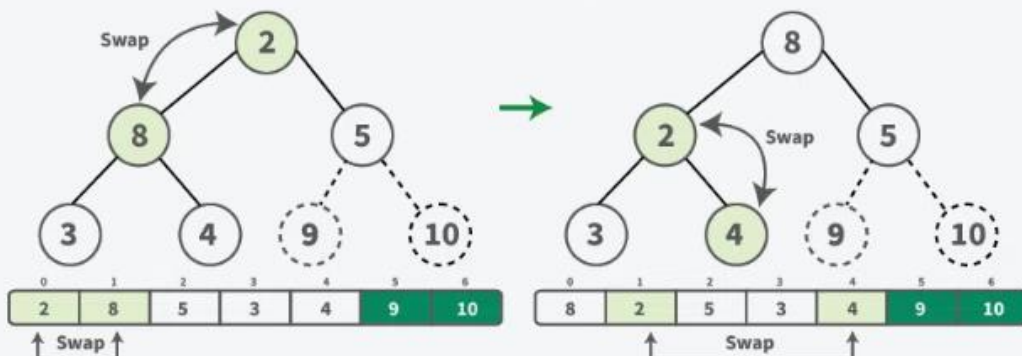
04
Step

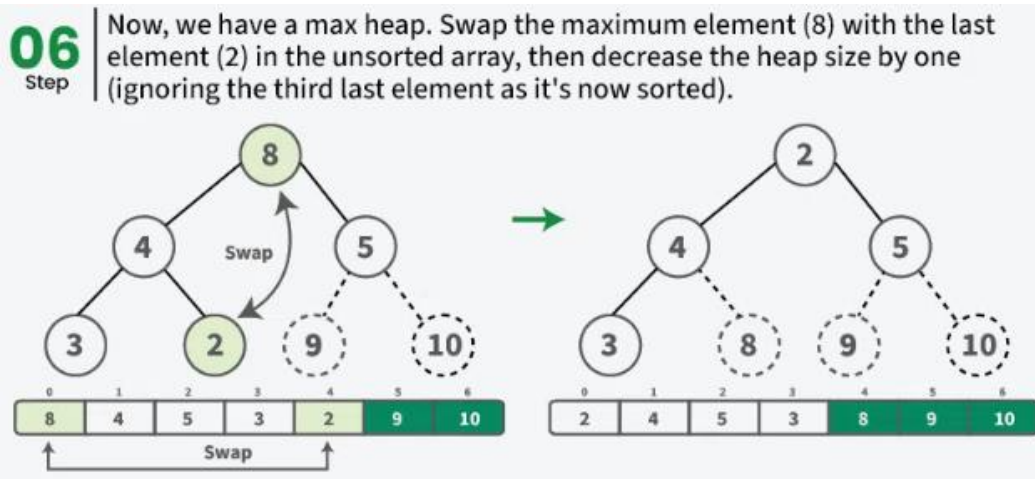
Now, we have a max heap. Swap the maximum element (9) with the last element (2) in the unsorted array, then decrease the heap size by one (ignoring the second last element as it's now sorted).



05
Step

Now, root violate the max-heap property, So, heapify it.
Swap node 2 with its largest child (8).
Still node 2 have larger children, so swap it with largest one (node 4).





In the illustration above, we have shown some steps to sort the array. We need to keep repeating these steps until there's only one element left in the heap.

```
import java.util.Arrays;
```

```
public class GFG {
```

```
    // To heapify a subtree rooted with node i
```

```
    static void heapify(int[] arr, int n, int i) {
```

```
        // Initialize largest as root
```

```
        int largest = i;
```

```
        // left index = 2*i + 1
```

```
        int l = 2 * i + 1;
```

```
        // right index = 2*i + 2
```

```
        int r = 2 * i + 2;
```

```
        // If left child is larger than root
```

```
        if (l < n && arr[l] > arr[largest])
```

```
            largest = l;
```

```
        // If right child is larger than largest so far
```

```
        if (r < n && arr[r] > arr[largest])
```

```
            largest = r;
```

```
        // If largest is not root
```

```
        if (largest != i) {
```

```
            int temp = arr[i];
```

```
            arr[i] = arr[largest];
```

```
            arr[largest] = temp;
```

```

        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

// Main function to do heap sort
static void heapSort(int[] arr) {
    int n = arr.length;
    // Build heap (rearrange vector)
    for (int i = n / 2 - 1; i >= 0; i--)
        heapify(arr, n, i);
    // One by one extract an element from heap
    for (int i = n - 1; i > 0; i--) {
        // Move current root to end
        int temp = arr[0];
        arr[0] = arr[i];
        arr[i] = temp;
        // Call max heapify on the reduced heap
        heapify(arr, i, 0);
    }
}

public static void main(String[] args) {
    int[] arr = { 9, 4, 3, 8, 10, 2, 5 };
    heapSort(arr);
    for (int i = 0; i < arr.length; ++i)
        System.out.print(arr[i] + " ");
}
}

```

Output

```

Sorted array is
2 3 4 5 8 9 10

```

Time Complexity: $O(n \log n)$

Auxiliary Space: $O(\log n)$, due to the recursive call stack. However, auxiliary space can be $O(1)$ for iterative implementation.

Important points about Heap Sort

- An in-place algorithm.
- Its typical implementation is not stable but can be made stable (See [this](#))
- Typically 2-3 times slower than well-implemented QuickSort. The reason for slowness is a lack of locality of reference.

Advantages of Heap Sort

- **Efficient Time Complexity:** Heap Sort has a guaranteed time complexity of $O(n \log n)$ in all cases, making it suitable for large datasets. The $\log n$ factor comes from the height of the binary heap, ensuring consistent performance.
- **Minimal Memory Usage:** Heap Sort can work in-place with minimal extra memory. Using an iterative `heapify()` avoids additional stack space, so apart from storing the array itself, no extra memory is required.
- **Simplicity:** Heap Sort is relatively easy to understand and implement compared to other efficient sorting algorithms, as it relies on the straightforward binary heap structure without advanced concepts like recursion (if iterative `heapify` is used).

Disadvantages of Heap Sort

- **Costly:** Heap sort is costly as the constants are higher compared to merge sort even if the time complexity is $O(n \log n)$ for both.
- **Unstable:** Heap sort is unstable. It might rearrange the relative order.
- **Inefficient:** Heap Sort is not very efficient because of the high constants in the time complexity.

Radix Sort :

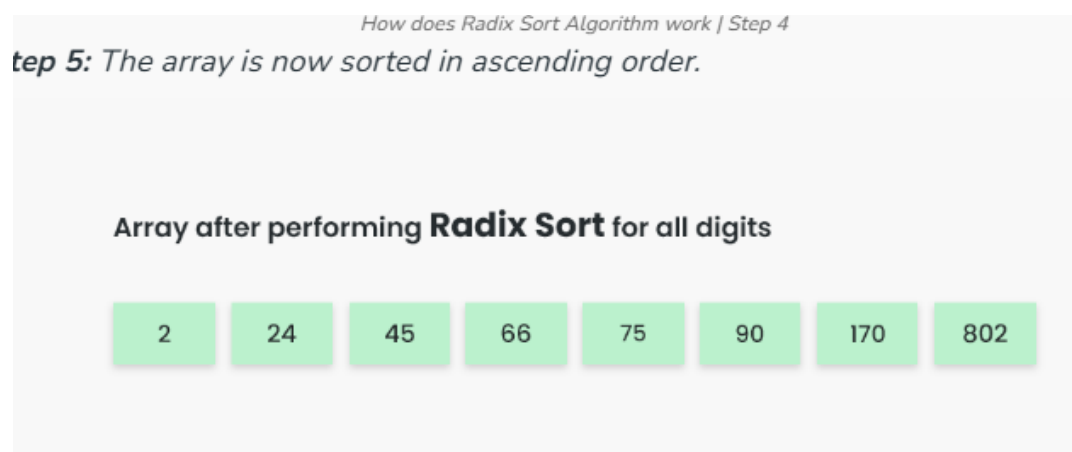
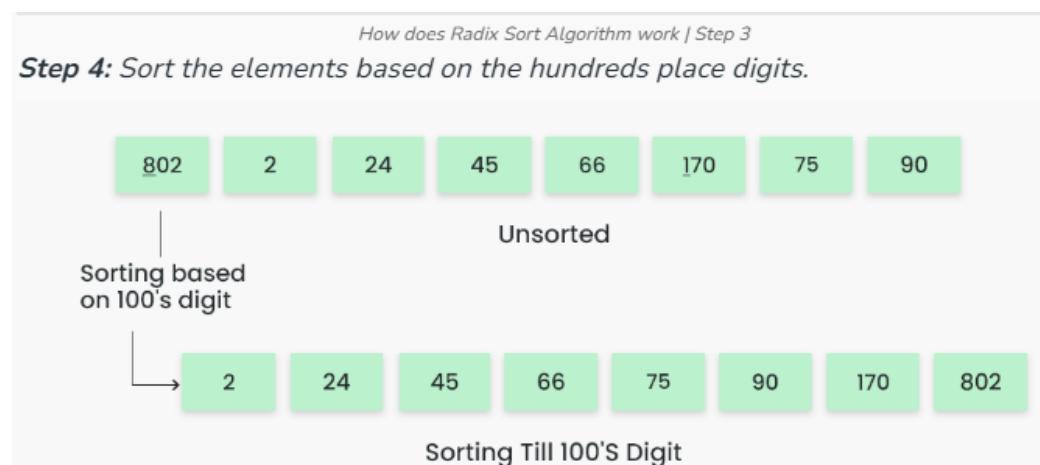
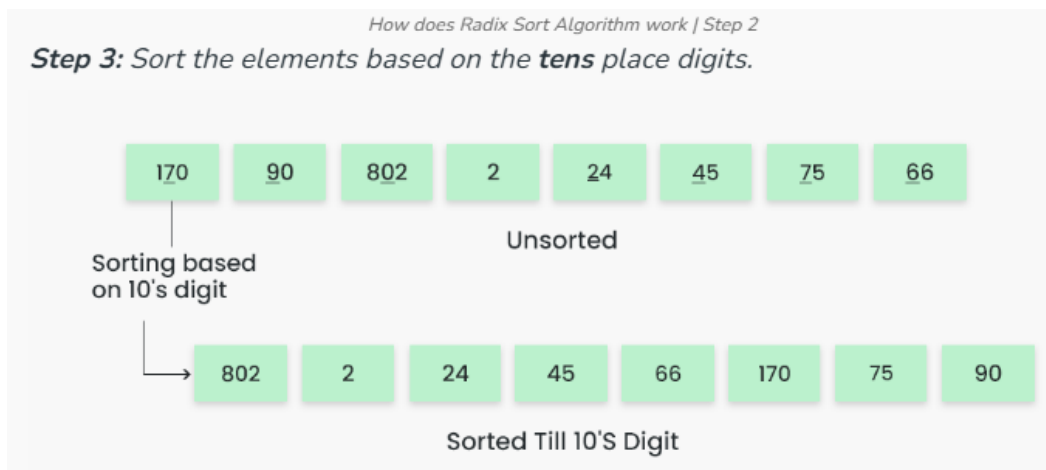
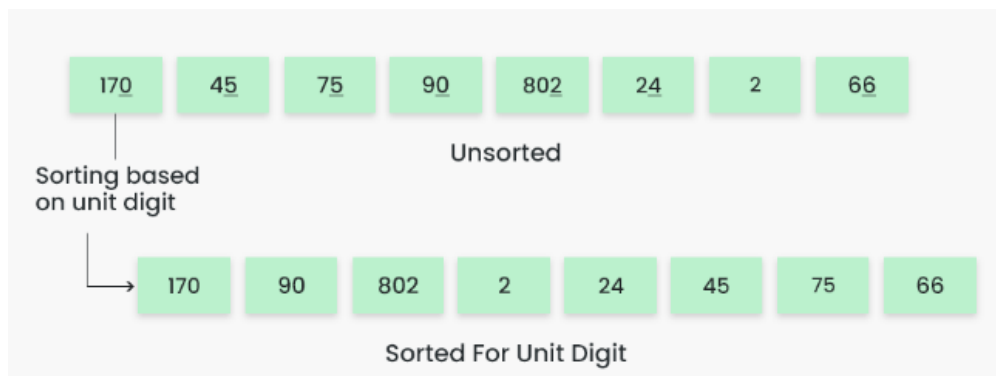
Radix Sort is a linear sorting algorithm (for fixed length digit counts) that sorts elements by processing them digit by digit. It is an efficient sorting algorithm for integers or strings with fixed-size keys.

- It repeatedly distributes the elements into buckets based on each digit's value. This is different from other algorithms like Merge Sort or Quick Sort where we compare elements directly.
- By repeatedly sorting the elements by their significant digits, from the least significant to the most significant, it achieves the final sorted order.
- We use a stable algorithm like Counting Sort to sort the individual digits so that the overall algorithm remains stable.

To perform radix sort on the array [170, 45, 75, 90, 802, 24, 2, 66], we follow these steps:

Step 1: Find the largest element, which is 802. It has three digits, so we will iterate three times.

Step 2: Sort the elements based on the **unit** place digits ($X=0$).



```

// Radix sort Java implementation
import java.io.*;
import java.util.*;
class Radix {
    // A utility function to get maximum value in arr[]
    static int getMax(int arr[], int n)
    {
        int mx = arr[0];
        for (int i = 1; i < n; i++)
            if (arr[i] > mx)
                mx = arr[i];
        return mx;
    }
    // A function to do counting sort of arr[] according to
    // the digit represented by exp.
    static void countSort(int arr[], int n, int exp)
    {
        int output[] = new int[n]; // output array
        int i;
        int count[] = new int[10];
        Arrays.fill(count, 0);
        // Store count of occurrences in count[]
        for (i = 0; i < n; i++)
            count[(arr[i] / exp) % 10]++;
        // Change count[i] so that count[i] now contains
        // actual position of this digit in output[]
        for (i = 1; i < 10; i++)
            count[i] += count[i - 1];
        // Build the output array
        for (i = n - 1; i >= 0; i--) {
            output[count[(arr[i] / exp) % 10] - 1] = arr[i];
            count[(arr[i] / exp) % 10]--;
        }

        // Copy the output array to arr[], so that arr[] now
        // contains sorted numbers according to current
        // digit
        for (i = 0; i < n; i++)
            arr[i] = output[i];
    }

    // The main function to that sorts arr[] of
    // size n using Radix Sort
    static void radixsort(int arr[], int n)

```

```

{
    // Find the maximum number to know number of digits
    int m = getMax(arr, n);
    // Do counting sort for every digit. Note that
    // instead of passing digit number, exp is passed.
    // exp is 10^i where i is current digit number
    for (int exp = 1; m / exp > 0; exp *= 10)
        countSort(arr, n, exp);
}
// A utility function to print an array
static void print(int arr[], int n)
{
    for (int i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
}

// Main driver method
public static void main(String[] args)
{
    int arr[] = { 170, 45, 75, 90, 802, 24, 2, 66 };
    int n = arr.length;
    // Function Call
    radixsort(arr, n);
    print(arr, n);
}
}

```

Output

```
2 24 45 66 75 90 170 802
```

Complexity Analysis of Radix Sort:

Time Complexity:

- Radix sort is a non-comparative integer sorting algorithm that sorts data with integer keys by grouping the keys by the individual digits which share the same significant position and value. It has a time complexity of $O(d * (n + b))$, where d is the number of digits, n is the number of elements, and b is the base of the number system being used.
- In practical implementations, radix sort is often faster than other comparison-based sorting algorithms, such as quicksort or merge sort, for large datasets, especially when the keys have many digits. However, its time complexity grows linearly with the number of digits, and so it is not as efficient for small datasets.

Auxiliary Space:

- Radix sort also has a space complexity of $O(n + b)$, where n is the number of elements and b is the base of the number system. This space complexity comes from the need to

create buckets for each digit value and to copy the elements back to the original array after each digit has been sorted.

Comparison of sorting algorithms in terms of Complexity:

Here we will discuss important properties of different sorting techniques including their complexity, stability and memory constraints.

Efficiency of an algorithm depends on two parameters:

1. Time Complexity
2. Space Complexity

Time Complexity: Time Complexity is defined as the number of times a particular instruction set is executed rather than the total time is taken. It is because the total time taken also depends on some external factors like the compiler used, processor's speed, etc.

Space Complexity: Space Complexity is the total memory space required by the program for its execution.

Both are calculated as the function of input size(n).

One important thing here is that in spite of these parameters the efficiency of an algorithm also depends upon the **nature** and **size of the input**.

Following is a quick revision sheet that you may refer at last minute

Algorithm	Time Complexity		
	Best	Average	Worst
Selection Sort	$\Omega(n^2)$	$\theta(n^2)$	$O(n^2)$
Bubble Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\theta(n^2)$	$O(n^2)$
Heap Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Quick Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n^2)$
Merge Sort	$\Omega(n \log(n))$	$\theta(n \log(n))$	$O(n \log(n))$
Bucket Sort	$\Omega(n+k)$	$\theta(n+k)$	$O(n^2)$
Radix Sort	$\Omega(nk)$	$\theta(nk)$	$O(nk)$

Time complexity Analysis –

We have discussed the best, average and worst case complexity of different sorting techniques with possible scenarios.

Comparison based sorting –

In comparison based sorting, elements of an array are compared with each other to find the sorted array. Insertion sort –

Average and worst case time complexity: n^2

Best case time complexity: n when array is already sorted. Worst case: when the array is reverse sorted.

Merge sort –

Best, average and worst case time complexity: $n \log n$ which is independent of distribution of data. Heap sort –

Best, average and worst case time complexity: $n \log n$ which is independent of distribution of data. Quick sort –

It is a divide and conquer approach with
recurrence relation: $T(n) = T(k) + T(n-k-1)$
+ cn

Worst case: when the array is sorted or reverse sorted, the partition algorithm divides the array in two subarrays with 0 and $n-1$ elements. Therefore,

$$T(n) = T(0) + T(n-1) + cn$$

Solving this we get, $T(n) = O(n^2)$

Best case and Average case: On an average, the partition algorithm divides the array in two subarrays with equal size. Therefore,

$$T(n) = 2T(n/2) + cn$$

Solving this we get,

$$T(n) = O(n \log n)$$

Non-comparison
based sorting –

In non-comparison based sorting, elements of array are not compared with each other to find the sorted array.

Sorting with disks:

We will first illustrate merge sort using disks and then SORTING analyse it as an external sorting method.

Example

The file F containing 600 records is to be sorted. The main memory is capable of sorting of 1000 records at a time. The input file F is stored on one disk and we have in addition another scratch disk. The block length of the input file is 500 records.

We see that the file could be treated as 6 sets of 1000 records each. Each set is sorted and stored on the scratch disk as a 'run'. These 6 runs will then be merged as follows: Allocate 3 blocks of memory each capable of holding 500 records. Two of these buffers B1 and B2 will be treated as input buffers and the third B3 as the output buffer. We have now the following.

1. 6 runs R1, R2, R3, R4, R5, R6 on the scratch disk. 3 buffers B1, B2 and B3.
Read 500 records from R1 into B1. Read 500 records from R2 into B2. Merge B1 and B2 and write into B3.
When B3 is full - write it out to the disk

as run R11. Similarly merge R3 and R4 to get run R12.

Merge R5 and R6 to get run R13.

Thus, from 6 runs of size 1000 each, we have now 3 runs of size 2000 each. The steps are repeated for steps R11 and R12 to get a run of size 4000.

This run is merged with R13 to get a single sorted run of size 6000. Pictorially, this can be represented as: Input file F with 6000 records

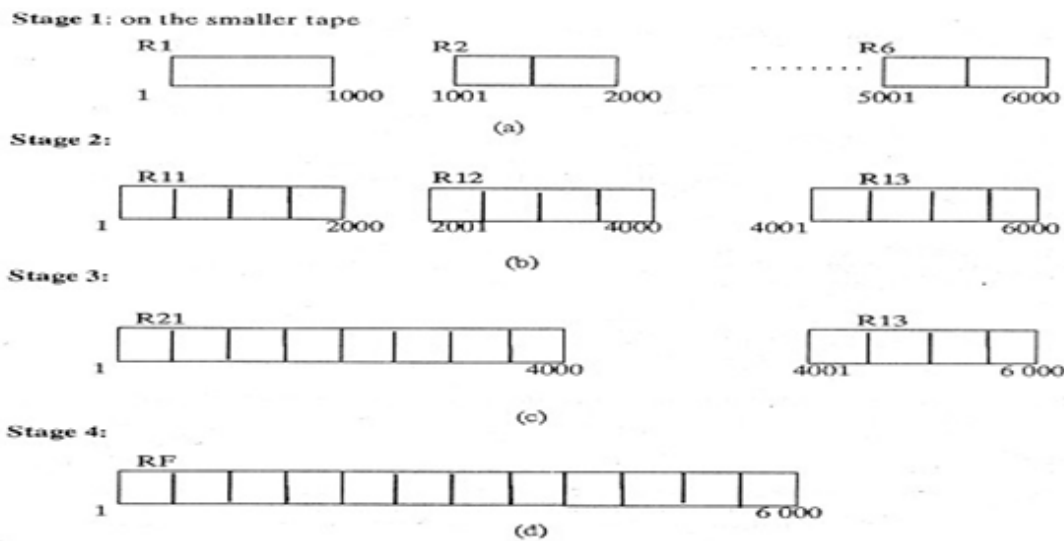


Figure 2: Merge Sort

The divisions in each run indicate the number of blocks.

Analysis

T1 -

See

k time T2

-

Lat

ency time

T3 - Transmission time for 1 block of

5000 records T - T1 + T2 + T3

T4 - Time to internally sort 1000 records

nTM - Time to merge n records from input buffers to the output buffer.

In stage 1 we read $6000/500 = 12$ blocks internally sort $6000/1000 = 6$ sets of 1000 records write $6000/500 = 12$ blocks

Therefore, time taken in stage 1 = $24T + 6T4$ In stage 2 we read 12 blocks write 12 blocks

Merge 5 x 2000 = 6000 records
 Time taken in stage 2 = $24T + 6000T_m$ In stage 3 we merge only 2 runs Therefore we read 8 blocks write 8 blocks
 merge 2 x 2000 = 4000 records
 Time taken in stage 3 = $16T + 4000T_m$ In stage 4 we read 12 blocks write 12 blocks
 merge 4000 + 2000 = 4000 records
 Therefore time taken in stage 4 = $24T + 6000T_m$

Total time taken is	=	$24T + 6T_4 + 24T + 6000T_m + 16T + 4000T_m + 24T + 6000T_m$
	=	$88T + 6T_4 + 1000T_m$

It is seen that the largest influencing factor is T_m , which depends on the number of passes made over the data or the number of times runs must be combined.

We have assumed a uniform seek and latency time for all blocks for simplicity of analysis. This may not be true in real life situations.

Time could also be reduced by exploiting the parallel features available, i.e. input/output and CPU processing carried out at the same time.

We will now focus on method to optimise the effects of the above factors, i.e. to say we will be carefully concerned with buffering and block size, assuming the internal sorting algorithms and the seek and latency time factors are the best possible.

K-Way Merging

In the above example, we used 2 way merging, i.e. combinations of two runs at a time. The number of passes over the data can be reduced by combining more runs at a time, hence the K-way merge where $K > 2$. In the game example, suppose we had used a 3 way merge then at stage 2 we would have had 2 runs of size 3000 each

at stage 3 we would have had a single sorted run of size 6000 This would have affected our analysis as follows:

$$\begin{aligned}
 \text{Stage 1} &= 24T + 6T_4 \\
 \text{Stage 2} &= 24T + 6000T_m \\
 \text{Stage 3} &= 24T + 6000T_m \\
 \text{Total} &= 74T + 6T_4 + 12000T_m
 \end{aligned}$$

There is obviously an enormous dip in the contribution of T_m to the total time.

This advantage is accompanied by certain side effects. The time to merge K runs, would obviously be more than the time to merge 2 runs. Here the value of T_m itself could be out of K in each step of the merge is needed. One such method is the use of a selection tree.

A selection tree is a binary tree in which each node represents the smaller of its children. It therefore follows that the root will be the smallest node. The way it makes is simple, Initially, the selection tree is built from the 1st item of each of the K runs. The one which gets to the root is selected as the smallest. Then the next item in the run from which the root was

selected enters the tree and the tree gets restructured to get a new root and so on till the K runs are merged.

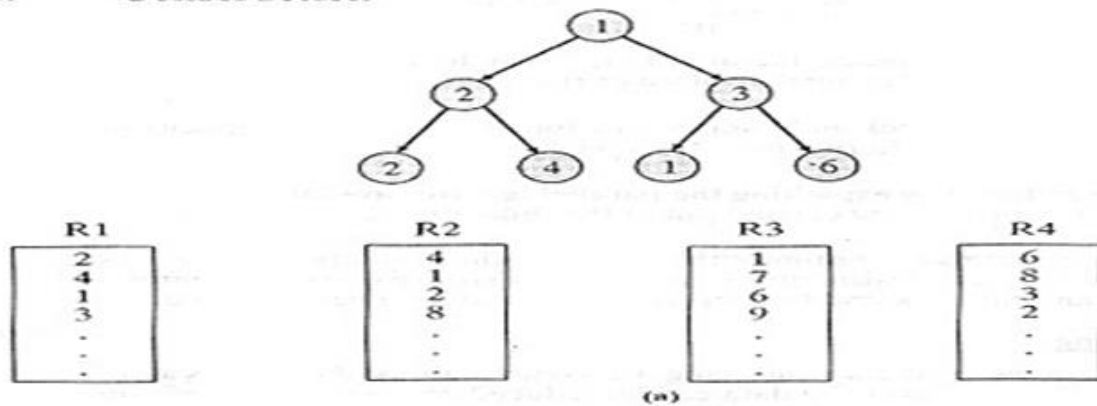
Example

Consider a selection tree for the following 4 runs:

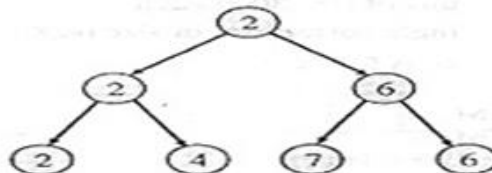
Consider a selection tree for the following 4 runs:

R1	2	4	1	3	
R2	4	1	2	8	
R3	1	7	6	9	
R4	6	8	3	2	

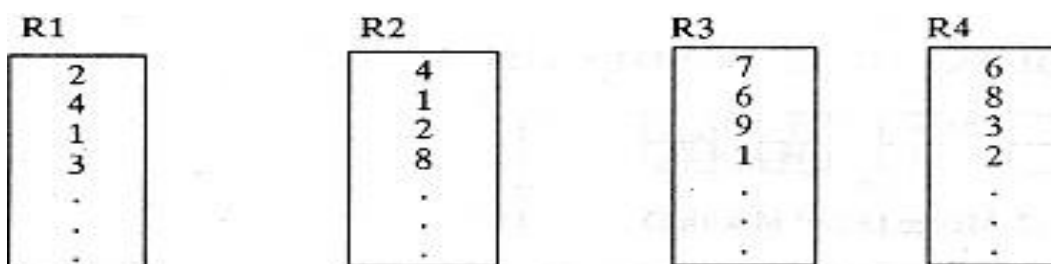
1. Construction



Node 1 was selected as the minimum. This is removed from the tree. Node belonged to R3, therefore the next item from R3 . 7 enters the tree.



The new root is 2. This came from R1 in the next step, the next item from R1 will enter the tree and so on. In going into a higher order merge, we save on input/output time. But with the increase in the order of the merge the number of buffers required also increases at the minimum to merge K runs we need $K + 1$ buffers. Now internal memory is a fixed entity, therefore if the number of buffers increases, there size must decrease. This implies a smaller block size on the disk and hence larger number of input/outputs. So beyond an optional value of K, we find that the advantage of reduced number of passes



(b)

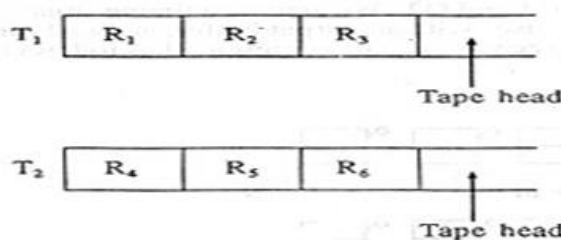
Sorting With Tapes:

Sorting with tapes is essentially similar to the merge sort used for sorting with disks. The differences arise due to the sequential access restriction of tapes. This makes the selection time prior to data transmission an important factor, unlike seek time and latency time. Thus in sorting with tapes we will be more concerned with arrangement of blocks and runs on the tape so as to reduce the selection or access time.

As in 6.3.4 we will use an example to determine the factors involved.

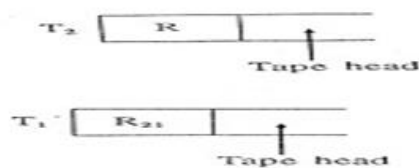
Example: A file of 6000 records is to be sorted. It is stored on a tape and the block length is 500. The main memory can sort upto a 1000 records at a time. We have in addition 4 search tapes T1 -T4 The steps in merging can be summarized as follows

Step 1:

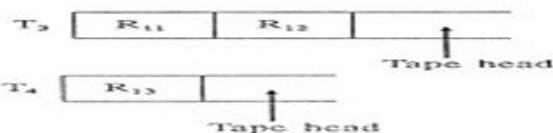
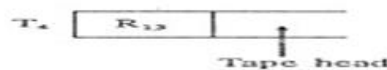


Step 2:

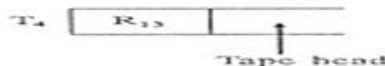
Rewind T₁ & T₂



Step 3: Rewind T₁ - T₄



Step 4: Rewind T₁ & T₄



Analysis

- t_{is} = time taken to internally sort 750 records.
- t_{rw} = time to read or write. One block of 250 records. Onto tape starting from present position of tape read/write head.
- t_{rew} = time to rewind tape over a length corresponding to one block.
- nt_m = time to merge n records from input buffers to output buffers using a 2-way merge.
- A = delay caused by having to wait for T_4 to be mounted in case we are ready to use T_n before it is mounted.
- Total time = $6t_{is} + 132t_{rw} + 12000t_m + 51t_{rew} + A$

The above computing time analysis assumes that no operations are carried out in parallel. The analysis could be carried further or in the case of disks to show the dependence of sort time on the number of passes made on the data.

Balanced Merge Sorts:

We see that the computing time in the above example depends as, in the case of disk sorting, essentially on the number of passes being made over the data. Use of a higher order merge results in a decrease in the number of passes being made without significantly changing the internal merge time. Hence we would like to use as high an order merge as possible. In disk sorting, the order of merge was limited essentially by the amount of main memory available for input/output buffers. A k -way merge required the use of $2k + 2$ buffers. Another more severe restriction on the merge order in the case of tapes is the number of tapes available. In order to avoid excessive seek time, it is necessary that runs being merged be on different tapes. Thus a k -way merge requires at least k -tapes for use as input tapes during the merge.

In addition, another tape is required for the output generated during the merge. Hence at least $k + 1$ tapes must be available for a k -way tape merge. Using $k + 1$ tapes for a k -way merge requires an additional pass over the output tape to redistribute the runs onto k -tapes for the next level of merge. This redistribution pass can be avoided by using $2k$ tapes. During the k -way merge, k of these tapes are used as input tapes and the remaining k as output tapes. At the next merge level, the role of input-output tapes is interchanged. These two approaches are now examined. Algorithms x1 and x2 perform a k -way merge with the $k + 1$ tapes, strategy while algorithm x3 performs a k -way merge using the $2k$ tapes strategy

```
Procedure x1;
[sort a file of records from a
given input tape using a  $k$ -way
merge given tapes  $t_1, \dots, t_{k+1}$  are
available for the sort.] label 200
begin
Create runs from the input tape distributing
them evenly over tapes  $t_1, \dots, t_k$  Rewind  $t_1 \dots t_k$ 
and also at the input tape;
If there is only one run goto
200; [the sorted file is on
 $t_1$ ].replace input tape by  $t_{k+1}$ ;
while true do
[repeatedly merge onto  $t_{k+1}$  and redistribute.
back onto  $t_1, t_k$ ]. begin
merge runs from  $t_1$  to
 $t_{k+1}$  onto  $t_{k+1}$ ; rewind
 $t_1, \dots, t_{k+1}$ ;

If number of runs on
 $t_{k+1} = 1$  then goto
200; [output on  $t_{k+1}$ ]
evenly distribute from  $t_{k+1}$ 
onto  $t_1, \dots, t_k$ ; rewind  $t_1, \dots, t_{k+1}$ ;
end;
200 end; [of x1]
```

Analysis: To simplify the analysis we assume that the number of runs generated (m) is a power of k . One pass over the entire file includes both reading and writing. The number of

passes in the while loop is $\log k^m$ merge passes and $\log k^{m-1}$ redistribution. passes . . the total number of passes is $21\log k^m$. If the time to rewind the entire input tape is t_{rew} , then the non-overlapping rewind time is approximately $21\log k^{m.trew}$

A cleverer way to tackle the problem is to rotate the output tape, i.e. tapes are used as output tapes in the cyclic order, $K+1, 1, 2, \dots, k$. This makes the redistribution from the output tape make less than a full pass over the file. Algorithm x2 describes the process.

Procedure x2

[Same definitions as for x1]

label 200 begin

Create runs from the input file distributing them evenly over tapes t_1, t_k ;

Rewind $t_1, \dots, t_k$ and also the input tape; If there is only one run then goto 200; [sorted file on t_1] replace input tape by t_{k+1} ;

$i = k + 1$; i is index of the output tape]

while true do begin

merge runs from the k tapes t_j ; $1 \leq j \leq k + 1$ and i onto t_i ;

rewind t_1, t_{k+1} ; If number of runs on $t_i = 1$ then goto 200 [output on t_1]

evenly distribute $(k-1)/k$ of the runs on t_i onto

tape t_j , $k = j \leq k + 1$ and $j \neq i$ and $j \neq i \bmod (k + 1) + 1$; rewind tapes t_j , $1 \leq j \leq k + 1$ and $j \neq i$; $i = i \bmod (k + 1) + 1$; end;

200 end; [end of x2]

Analysis: The only difference between algorithms x_1 and x_2 is in the redistributing time. m is the number of runs generated in line 5. Redistributing is done $\log k^{m-1}$ times, but each such pass reads and writes only $(k-1)/k$ of the file. Hence the effective number of passes made over the data is $(2-1/k) \log k^m + 1/k$. for two- way merging on 3 tapes algorithm x^2 will make $3/2 \log k^m + 1/2$ passes while x_1 will make $21\log k^m$ passes. If t_{rew} is the rewind time then the non-overlapping rewind time for x^2 is utmost $(1 + 1/k)(\log k^m) t_{rew} +$

$(1-1/k) t_{rew}$. Instead of distributing runs as shown we could write the first m/k runs on one tape, begin rewind, write the next m/k runs on the second tape, begin rewind etc. In this case we can begin filling input buffers for the next merge level while some of the tapes are still rewinding.

In case a k -way merge is using $2k$ tapes, no redistribution is needed and so the number of passes made is only $\log k^{m+1}$. This implies that if $2k + 1$ tapes are available than a $2k$ way merge will make $(2-1/2k)\log 2k^m$

$+ 1/(2k)$ passes while a k -way merge utilizing $2k$ tapes will make only $\log k^{m+1}$ passes. The table compares the number of passes being made in the two methods for some value of k .

k	$2k$ -way	k -way
1.	$3/2 \log 2m + 1/2$	-

- | | | |
|----|--------------------------------------|----------------|
| 2. | $\frac{7}{8} \log_2 m + \frac{1}{4}$ | $\log_2 M + 1$ |
| 3. | $1.124 \log_3 m + \frac{1}{6}$ | $\log_3 m + 1$ |
| 4. | $1.25 \log_4 m + \frac{1}{8}$ | $\log_4 m + 1$ |

As is evident from the table, for $k \geq 2$ a k -way merge using $2k$ tapes is better than a $2k$ -way merge using $2k + 1$ tapes.

Algorithm x3 makes a k -way merge sort using $2k$ tapes. Procedure x3;

[sort a file of records from a given input tape using a k -way merge on $2k$ tapes] begin

Create runs from the input file distributing them evenly over tapes $t_1 \dots t_k$; rewind $t_1 \dots t_k$; rewind the input tape;

replace the input tape by tape t_{2k} ; $i := 0$; while total number of runs $t_{i+1} \dots t_{i+k} \neq 1$ do begin $j := 1 - i$;

perform a k -way merge from $t_{i+1}, \dots, t_{i+k}$ evenly distributing output runs onto $t_{j+1}, \dots, t_{j+k}$; rewind $t_1 \dots t_{2k}$;

$i := j$; [switch input and output tapes] end;

[sorted file is on t_{i+1}] end; [end of x3]

Analysis: To simplify the analysis, assume that m is a power of k . In addition to the initial run creation pass, the algorithm makes $\log_k m$ merge passes. Let t_{rew} be the time to rewind the entire input file. If m is a power of k then the time to rewind tapes $t_1, \dots, t_{2k}$ in the while loop takes t_{rew}/k for each but the last loop iteration. The last rewind takes time t_{rew} . The total rewind time is therefore bound by $(2 + (\log_k m - 1)/k)t_{rew}$.

It should be noted that all the above algorithms use the buffering strategy developed in the k -way merge. The proper choice of buffer lengths and the merge order (restricted by the number of tapes available) would result in an almost complete overlap of internal processing with input/output time. At the end of each level of merge, processing will have to wait till the tapes rewind. This wait can be minimized using the run distribution strategy developed in algorithm x2.

Polyphase merging:

The problem with balanced multiway merging is that it requires either an excessive number of tape units or excessive copying. Polyphase merging is a method to eliminate virtually all the copying by changing the way in which the small sorted blocks are merged together. The basic idea is to distribute the sorted blocks produced by replacement selection somewhat unevenly among the available tape units (leaving one empty) and then to apply a 'merge = until-empty' strategy at which point one of the input tapes and the output tape switch roles.

For example suppose that we have just 3 tapes, and we start with the initial configuration of sorted blocks on the tapes as shown at the top of the figure. Tape 3 is initially empty the output tape for the first merges.

Tape	1	B P S T U-J O - B H O -E F N S - H J O.
	2	FHY - BNQ - FM.
	3	.
Tape	1	EFNS.HJO.

```

      2 .
      3 B F H P S T U Y-B J N O Q. B F H M O.
Tape  1 .
      2 B E F F H N P S S T U Y- B H J J N O O Q.
      3 B F H M O.

```

Now after three two way merges from tapes 1 and 2 to tape 3, the second tape becomes empty. Then after two two-way merges from tapes 1 and 3 onto tape 2, the first tape becomes empty. The sort is completed in

two more steps. First a 2-way merge from tapes 2 and 3 onto tape 1 leaves one file on tape 1, one file on tape 2. Then a two-way merge from tapes 1 and 2 leaves the entire sorted file on tape 3.

This merge - until-empty strategy can be extended to work for an arbitrary number of tapes. The figure indicates how 6 tapes might be used to sort with 497 initial runs. If we start as shown with tape 2 the output tape, tape 1 having 61 initial runs etc, then after running a five-way "merge-until-empty", we have tape 1 empty, tape 2 with 61 runs etc. as shown in the second column of the figure. At this point we can rewind tape 1 and make it the output tape and rewind tape 2 and make it an input tape. Continuing in this way we arrive at the entire file sorted on tape 1 as shown by the last column. The merge is broken up into many phases which don't involve all the data, but no direct copying is involved.

The main difficulty in implementing polyphase merge is to determine how to distribute the initial runs. The table can be built by working backwards: take the largest number in each column, make it zero and add it to each of the other numbers to get the previous column. This technique works for any number of tapes (at least 3): the numbers which arise are "generalized Fibonacci numbers". Of course the number of initial runs may not be known in advance, and it probably won't be exactly a generalized Fibonacci number. Thus a number of "dummy" runs must be added to make the number of initial runs exactly what is needed for the table.

Tape 1	61	0	31	15	7	3	1	0	1
Tape 2	0	61	30	14	6	2	0	1	0
Tape 3	120	59	28	12	4	0	2	1	0
Tape 4	116	59	24	8	0	4	2	1	0
Tape 5	108	47	16	0	8	4	2	1	0
Tape 6	92	31	0	16	8	4	2	1	0

Run distribution for sin-tape polyphase merge.

Remarks: A factor we have not considered is the time taken to rewind the tape. Before the merge for the next phase can begin, the tape must be rewound and the computer is essentially idle. It is possible to modify the above method so that virtually all rewind time is overlapped with internal processing and read/write on other tapes. However, the savings over the multiway balanced merge are quite limited. Even polyphase merging is better than balanced merging only for small P, and that not substantially. For P8, balanced merging is likely to run faster than polyphase and for smaller P, the effect of polyphase merging is to save two tapes.

UNIT-V

Searching And Indexing:

Searching

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

- Linear Search
- Binary Search

Linear Search:

Linear search is the simplest search algorithm and often called sequential search. In this type of searching, we simply traverse the list completely and match each element of the list with the item whose location is to be found. If the match found then location of the item is returned otherwise the algorithm return NULL.

Linear search is mostly used to search an unordered list in which the items are not sorted.

The algorithm of linear search is given as follows

algorithm linear_search(a, n,
val)

step 1: [initialize] set pos = -

1 step 2: [initialize] set i = 1

step 3: repeat step 4 while

i<=n step 4: if a[i] = val

set pos = i

print posgo to step 6 [end of
if]

set i = i + 1 [end of loop]

step 5: if pos = -1

print " value is not presentin
the array " [end of if]

step 6: exit

Complexity of algorithm

Complexity	Best Case	Average Case	Worst Case
Time	O(1)	O(n)	O(n)
Space			O(1)

```
import java.util.Scanner;
```

```
public class Leniear_Search {
```

```
    public static void main(String[] args) {
```

```
        int[] arr = {10, 23, 15, 8, 4, 3, 25, 30, 34, 2, 19};
```

```
        int item,flag=0;
```

```
        Scanner sc = new
```

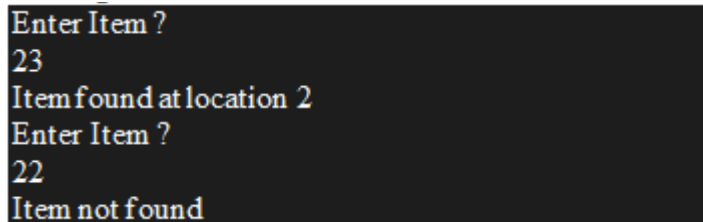
```
        Scanner(System.in);
```

```

System.out.println("Enter Item ?");
item = sc.nextInt();
for(int i = 0; i<10; i++)
{
    if(arr[i]==item)
    {
        flag = i+1;
        break;
    }
    else
        flag = 0;
}
if(flag != 0)
{
    System.out.println("Item found at location" + flag);
}
else
    System.out.println("Item not found"); }}

```

Output:



```

Enter Item ?
23
Item found at location 2
Enter Item ?
22
Item not found

```

Binary Search:

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Binary search algorithm is given below.

BINARY_SEARCH(A, lower_bound, upper_bound, VAL) Step 1: [initialize] set beg = lower_bound END = upper_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG

<=END Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL SET POS = MID

PRINT POS

Go to Step 6

ELSE IF A[MID] > VAL SET END = MID - 1 ELSE

SET BEG = MID + 1 [END OF IF]

[END OF LOOP]

Step 5: IF POS = -1

PRINT "VALUE IS NOT PRESENT IN THE
ARRAY" [END OF IF]

Step 6: EXIT

Complexity

SN	Performance	Complexity
1	Worst case	$O(\log n)$
2	Best case	$O(1)$
3	Average Case	$O(\log n)$
4	Worst case space complexity	$O(1)$

Example

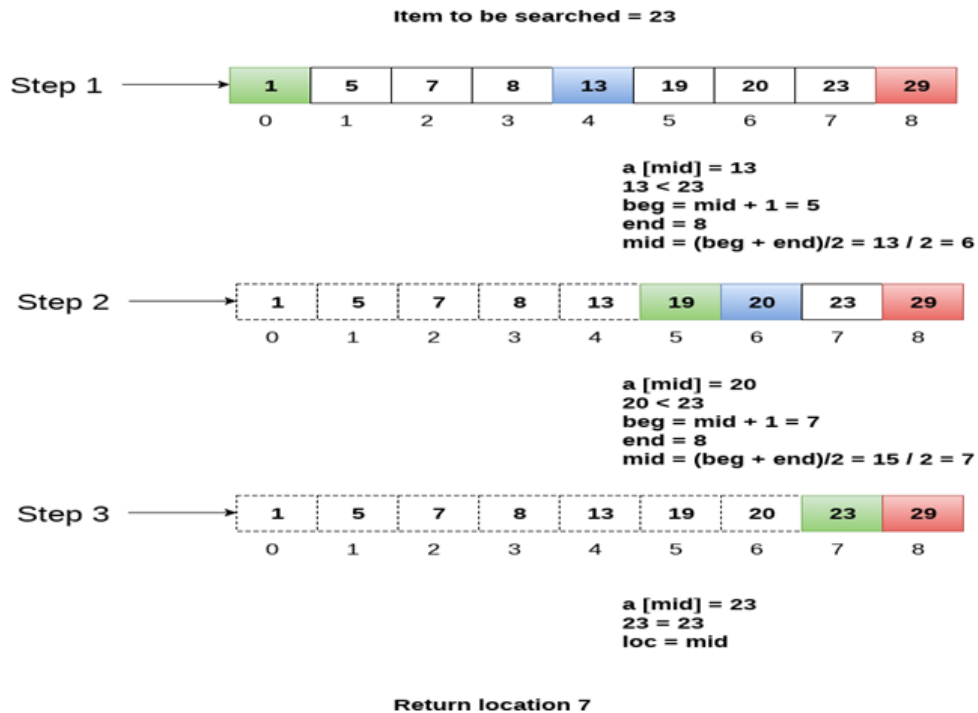
Let us consider an array $arr = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$. Find the location of the item 23 in the array.

In 1st step :

- . $BEG = 0$
- . $END = 8$
- . $MID = 4$
- . $a[mid] = a[4] = 13 < 23$, therefore in Second step:
- . $Beg = mid + 1 = 5$
- . $End = 8$
- . $mid = 13/2 = 6$
- . $a[mid] = a[6] = 20 < 23$,
therefore;

in third step:

- . $beg = mid + 1 = 7$
- . $End = 8$
- . $mid = 15/2 = 7$
- . $a[mid] = a[7]$
- . $a[7] = 23 = \text{item}$;
- . therefore, set location = mid;
- . The location of the item will be 7.



```

import java.util.*;

public class BinarySearch {

    public static void main(String[] args) {

        int[] arr = {16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
        int item, location = -1;
        System.out.println("Enter the item which you want
        to search"); Scanner sc = new Scanner(System.in);
        item = sc.nextInt();
        location = binarySearch(arr,0,9,item);
        if(location != -1)
            System.out.println("the location of the item is "+location);

        else
            System.out.println("Item not found");
        }

    public static int binarySearch(int[] a, int beg, int end, int item)
    {

        int mid;
        if(end >= beg)
        {

            mid = (beg + end)/2;
            if(a[mid] == item)
            {

                return mid+1;
            }
        }
    }
}

```

```

    }
    else if(a[mid] < item)
    {
        return binarySearch(a,mid+1,end,item);
    }

    else
    {
        return binarySearch(a,beg,mid-1,item);
    } }

return -1; }

```

Output:

```

Enter the item which you want to search
45
the location of the item is 5

```

Hash Table:

Hash table is one of the most important data structures that uses a special function known as a hash function that maps a given value with a key to access the elements faster.

A Hash table is a data structure that stores some information, and the information has basically two main components, i.e., key and value. The hash table can be implemented with the help of an associative array. The efficiency of mapping depends upon the efficiency of the hash function used for mapping.

For example, suppose the key value is John and the value is the phone number, so when we pass the key value in the hash function shown as below:

Hash(key)= index;

When we pass the key in the hash function, then it gives the index. Hash(john) = 3;

The above example adds the john at the index 3.

Drawback of Hash function

A Hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key of two different values.

Hashing:

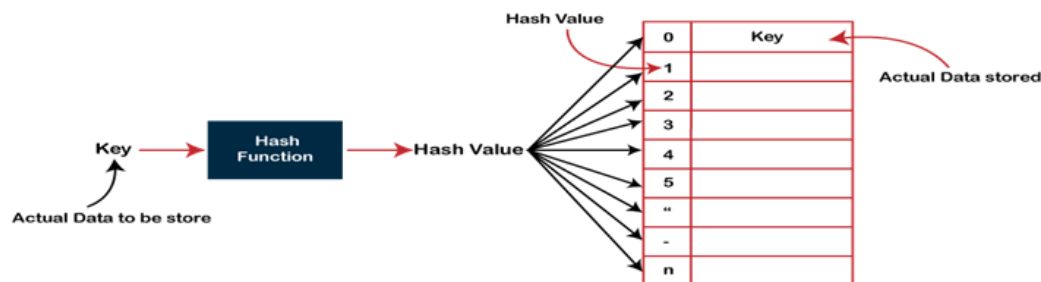
Hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is $O(1)$. Till now, we read the two techniques for searching, i.e., linear search and binary search.

The worst time complexity in linear search is $O(n)$, and $O(\log n)$ in binary search. In both the searching techniques, the searching depends upon the number of elements but we want the technique that takes a constant time. So, hashing technique came that provides a constant time.

In Hashing technique, the hash table and hash function are used. Using the hash function, we can calculate the address at which the value can be stored.

The main idea behind the hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as:

$$\text{Index} = \text{hash}(\text{key})$$



There are three ways of calculating the hash function:

- **Division method**
- **Folding method**
- **Mid square method**

In the division method, the hash function can be defined as:

$$h(k_i) = k_i \% m;$$

where **m** is the size of the hash table.

For example, if the key value is 6 and the size of the hash table is 10. When we apply the hash function to key 6 then the index would be:

$$h(6) = 6\%10 = 6$$

The index is 6 at which the value is stored.

Collision:

When the two different values have the same value, then the problem occurs between the two values, known as a collision. In the above example, the value is stored at index 6. If the key value is 26, then the index would be:

$$h(26) = 26\%10 = 6$$

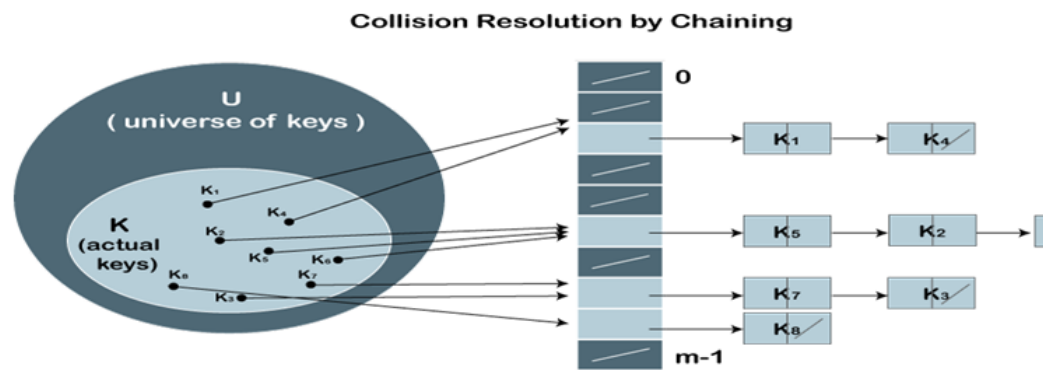
Therefore, two values are stored at the same index, i.e., 6, and this leads to the collision problem. To resolve these collisions, we have some techniques known as collision techniques.

The following are the collision techniques:

- **Open Hashing:** It is also known as closed addressing.
- **Closed Hashing:** It is also known as open addressing.

Open Hashing

In Open Hashing, one of the methods used to resolve the collision is known as a chaining method.



Let's first understand the chaining to resolve the collision. Suppose we have a list of key values

A = 3, 2, 9, 6, 11, 13, 7, 12 where $m = 10$, and $h(k) = 2k+3$

In this case, we cannot directly use $h(k) = k_i/m$ as $h(k) = 2k+3$

The index of key value 3 is:

$$\text{index} = h(3) = (2(3)+3)\%10 = 9$$

The value 3 would be stored at the index 9.

The index of key value 2 is:

$$\text{index} = h(2) = (2(2)+3)\%10 = 7$$

The value 2 would be stored at the index 7.

The index of key value 9 is:

$$\text{index} = h(9) = (2(9)+3)\%10 = 1$$

The value 9 would be stored at the index 1.

The index of key value 6 is:

$$\text{index} = h(6) = (2(6)+3)\%10 = 5$$

The value 6 would be stored at the index 5.

The index of key value 11 is:

$$\text{index} = h(11) = (2(11)+3)\%10 = 5$$

The value 11 would be stored at the index 5. Now, we have two values (6, 11) stored at the same index, i.e., 5. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 11 to this list. After the creation of the new list, the newly created list will be linked to the list having value 6.

The index of key value 13 is:

$$\text{index} = h(13) = (2(13)+3)\%10 = 9$$

The value 13 would be stored at index 9. Now, we have two values (3, 13) stored at the same index, i.e., 9. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 13 to this list. After the creation of the new list, the newly created list will be linked to the list having value 3.

The index of key value 7 is:

$$\text{index} = h(7) = (2(7)+3)\%10 = 7$$

The value 7 would be stored at index 7. Now, we have two values (2, 7) stored at the same index, i.e., 7. This leads to the collision problem, so we will use the chaining method to avoid the collision. We will create one more list and add the value 7 to this list. After the creation of the new list, the newly created list will be linked to the list having value 2.

The index of key value 12 is:

$$\text{index} = h(12) = (2(12)+3)\%10 = 7$$

According to the above calculation, the value 12 must be stored at index 7, but the value 2 exists at index 7. So, we will create a new list and add 12 to the list. The newly created list will be linked to the list having a value 7.

The calculated index value associated with each key value is shown in the below table:

key	Location(u)
3	$((2*3)+3)\%10 = 9$
2	$((2*2)+3)\%10 = 7$
9	$((2*9)+3)\%10 = 1$
6	$((2*6)+3)\%10 = 5$
11	$((2*11)+3)\%10 = 5$

13	$((2*13)+3)\%10 = 9$
7	$((2*7)+3)\%10 = 7$
12	$((2*12)+3)\%10 = 7$

Closed Hashing

In Closed hashing, three techniques are used to resolve the collision:

1. Linear probing
2. Quadratic probing
3. Double Hashing technique

Linear Probing:

Linear probing is one of the forms of open addressing. As we know that each cell in the hash table contains a key-value pair, so when the collision occurs by mapping a new key to the cell already occupied by another key, then linear probing technique searches for the closest free locations and adds a new key to that empty cell. In this case, searching is performed sequentially, starting from the position where the collision occurs till the empty cell is not found.

Let's understand the linear probing through an example.

Consider the above example for the linear probing:

A = 3, 2, 9, 6, 11, 13, 7, 12 where m = 10, and $h(k) = 2k+3$

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5 respectively. The calculated index value of 11 is 5 which is already occupied by another key value, i.e., 6. When linear probing is applied, the nearest empty cell to the index 5 is 6; therefore, the value 11 will be added at the index 6.

The next key value is 13. The index value associated with this key value is 9 when hash function is applied. The cell is already filled at index 9. When linear probing is applied, the nearest empty cell to the index 9 is 0; therefore, the value 13 will be added at the index 0.

The next key value is 7. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 8; therefore, the value 7 will be added at the index 8.

The next key value is 12. The index value associated with the key value is 7 when hash function is applied. The cell is already filled at index 7. When linear probing is applied, the nearest empty cell to the index 7 is 2; therefore, the value 12 will be added at the index 2.

Quadratic Probing:

In case of linear probing, searching is performed linearly. In contrast, quadratic probing is an open addressing technique that uses quadratic polynomial for searching until a empty slot is found.

It can also be defined as that it allows the insertion k_i at first free location from $(u+i^2)\%m$ where $i=0$ to $m-1$.

Let's understand the quadratic probing through an example. Consider the same example which we discussed in the linear probing. **A = 3, 2, 9, 6, 11, 13, 7, 12** where **m = 10**, and **h(k) = 2k+3**

The key values 3, 2, 9, 6 are stored at the indexes 9, 7, 1, 5, respectively. We do not need to apply the quadratic probing technique on these key values as there is no occurrence of the collision.

The index value of 11 is 5, but this location is already occupied by the 6. So, we apply the quadratic probing technique.

When i = 0

$$\text{Index} = (5+0^2)\%10 = 5$$

When i=1

Since location 6 is empty, so the value 11 will be added at the index 6.

The next element is 13. When the hash function is applied on 13, then the index value comes out to be 9, which we already discussed in the chaining method. At index 9, the cell is occupied by

0	13
1	9
2	
3	12
4	
5	6
6	11
7	2
8	7
9	3

another value, i.e., 3. So, we will apply the quadratic probing technique to calculate the free location.

$$\text{Index} = (5+1^2)\%10 = 6$$

When i=0

$$\text{Index} = (9+0^2)\%10 = 9$$

When i=1

$$\text{Index} = (9+1^2)\%10 = 0$$

Since location 0 is empty, so the value 13 will be added at the index 0.

The next element is 7. When the hash function is applied on 7, then the index value comes out to be 7, which we already discussed in the chaining method. At index 7, the cell is occupied by another value, i.e., 7. So, we will apply the quadratic probing technique to calculate the free location.

When i=0

$$\text{Index} = (7+0^2)\%10 = 7$$

When i=1

$$\text{Index} = (7+1^2)\%10 = 8$$

Since location 8 is empty, so the value 7 will be added at the index 8.

The next element is 12. When the hash function is applied on 12, then the index value comes out to be 7. When we observe the hash table then we will get to know that the cell at index 7 is already occupied by the value 2. So, we apply the Quadratic probing technique on 12 to determine the free location.

When i=0

$$\text{Index} = (7+0^2)\%10 = 7$$

When i=1

$$\text{Index} = (7+1^2)\%10 = 8$$

When i=2

$$\text{Index} = (7+2^2)\%10 = 1$$

When i=3

$$\text{Index} = (7+3^2)\%10 = 6$$

When i=4

$$\text{Index} = (7+4^2)\%10 = 3$$

Since the location 3 is empty, so the value 12 would be stored at the index 3. The final hash table would be:

Therefore, the order of the elements is 13, 9, _, 12, _, 6, 11, 2, 7, 3.

Double Hashing:

Double hashing is an open addressing technique which is used to avoid the collisions. When the collision occurs then this technique uses the secondary hash of the key. It uses one hash value as an index to move forward until the empty location is found.

In double hashing, two hash functions are used. Suppose $h_1(k)$ is one of the hash functions used to calculate the locations whereas $h_2(k)$ is another hash function. It can be defined as "insert k_i at first free place from $(u+v*i)\%m$ where $i=(0 \text{ to } m-1)$ ". In this case, u is the location computed

using the hash function and v is equal to $(h_2(k)\%m)$. Consider the same example that we use in quadratic probing.

A = 3, 2, 9, 6, 11, 13, 7, 12 where

m = 10, and $h_1(k) = 2k+3$

$h_2(k) = 3k+1$

key	Location (u)	v	probe
3	$((2*3)+3)\%10 = 9$	-	1
2	$((2*2)+3)\%10 = 7$	-	1
9	$((2*9)+3)\%10 = 1$	-	1
6	$((2*6)+3)\%10 = 5$	-	1
11	$((2*11)+3)\%10 = 5$	$(3(11)+1)\%10 = 4$	3
13	$((2*13)+3)\%10 = 9$	$(3(13)+1)\%10 = 0$	
7	$((2*7)+3)\%10 = 7$	$(3(7)+1)\%10 = 2$	
12	$((2*12)+3)\%10 = 7$	$(3(12)+1)\%10 = 7$	2

As we know that no collision would occur while inserting the keys (3, 2, 9, 6), so we will not apply double hashing on these key values.

On inserting the key 11 in a hash table, collision will occur because the calculated index value of 11 is 5 which is already occupied by some another value. Therefore, we will apply the double hashing technique on key 11. When the key value is 11, the value of v is 4.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

$$\text{Index} = (5+4*0)\%10 = 5$$

When i=1

$$\text{Index} = (5+4*1)\%10 = 9$$

When i=2

$$\text{Index} = (5+4*2)\%10 = 3$$

Since the location 3 is empty in a hash table; therefore, the key 11 is added at the index 3.

The next element is 13. The calculated index value of 13 is 9 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 0.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

$$\text{Index} = (9+0*0)\%10 = 9$$

We will get 9 value in all the iterations from 0 to m-1 as the value of v is zero. Therefore, we cannot insert 13 into a hash table.

The next element is 7. The calculated index value of 7 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 2.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

$$\text{Index} = (7 + 2*0)\%10 = 7$$

When i=1

$$\text{Index} = (7+2*1)\%10 = 9$$

When i=2

$$\text{Index} = (7+2*2)\%10 = 1$$

When i=3

$$\text{Index} = (7+2*3)\%10 = 3$$

When i=4

$$\text{Index} = (7+2*4)\%10 = 5$$

When i=5

$$\text{Index} = (7+2*5)\%10 = 7$$

When i=6

$$\text{Index} = (7+2*6)\%10 = 9$$

When i=7

$$\text{Index} = (7+2*7)\%10 = 1$$

When i=8

$$\text{Index} = (7+2*8)\%10 = 3$$

When i=9

$$\text{Index} = (7+2*9)\%10 = 5$$

Since we checked all the cases of i (from 0 to 9), but we do not find suitable place to insert 7. Therefore, key 7 cannot be inserted in a hash table.

The next element is 12. The calculated index value of 12 is 7 which is already occupied by some another key value. So, we will use double hashing technique to find the free location. The value of v is 7.

Now, substituting the values of u and v in $(u+v*i)\%m$

When i=0

$$\text{Index} = (7+7*0)\%10 = 7$$

When i=1

$$\text{Index} = (7+7*1)\%10 = 4$$

Since the location 4 is empty; therefore, the key 12 is inserted at the index 4. The final hash table would be:

0	
1	9
2	
3	11
4	12
5	6
6	
7	2
8	
9	3

The order of the elements is _, 9, _, 11, 12, 6, _, 2, _, 3.

Cylinder Surface Indexing:

Cylinder surface indexing is a very simple technique used only for the primary key index of a sequentially ordered file.

The index file will contain two fields—cylinder index and several surface indices.

There are multiple cylinders, and each cylinder has multiple surfaces. If the file needs m cylinders for storage then the cylinder index will contain m entries

When a record with a particular key value has to be searched, then the following steps are

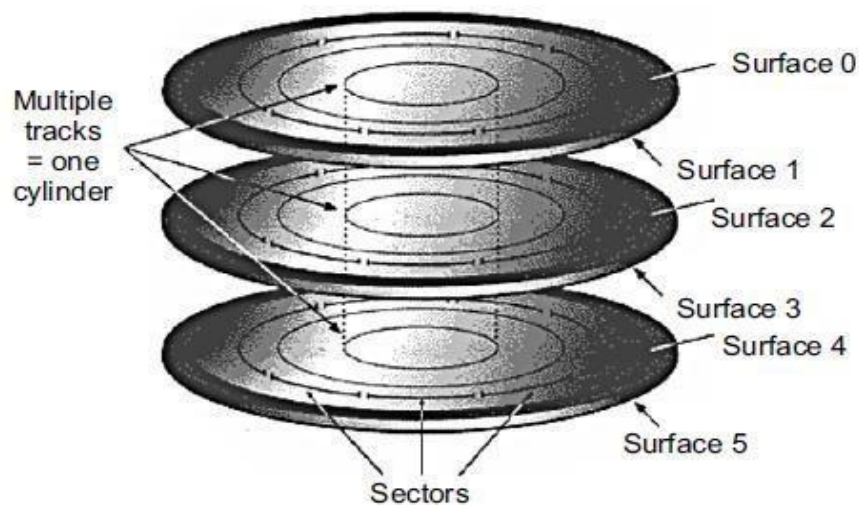


Figure Physical and logical organization of disk

performed:

- First the cylinder index of the file is read into memory.
- Second, the cylinder index is searched to determine which cylinder holds the desired record. For this, either the binary search technique can be used or the cylinder index can be made to store an array of pointers to the starting of individual key values. In either case the search will take $O(\log m)$ time.
- After the cylinder index is searched, appropriate cylinder is determined.
- Depending on the cylinder, the surface index corresponding to the cylinder is then retrieved from the disk.

- Since the number of surfaces on a disk is very small, linear search can be used to determine surface index of the record.
- Once the cylinder and the surface are determined, the corresponding track is read and searched for the record with the desired key.

Hence, the total number of disk accesses is three—first, for accessing the cylinder index, second for accessing the surface index, and third for getting the track address.

Hashed Indices:

Hashing is used map to the same address, then collision occurs and schemes to resolve these collisions are applied to generate a new address to compute the address of a record by using a hash function on the search key value.

The hashed values

Choosing a **good hash function** is critical to the success of this technique. By a good hash function, it mean two things.

1. First, a good hash function, irrespective of the number of search keys, gives an average-case lookup that is a small constant.
2. Second, the function distributes records uniformly and randomly among the buckets, where a bucket is defined as a unit of one or more records

The **worst hash function** is one that maps all the keys to the same bucket.

The drawback of using hashed indices includes:

- Though the number of buckets is fixed, the number of files may grow with time.
- If the number of buckets is too large, storage space is wasted.
- If the number of buckets is too small, there may be too many collisions.

The following operations are performed in a hashed file organization.

Insertion

To insert a record that has k_i as its search value, use the hash function $h(k_i)$ to compute the address of the bucket for that record.

If the bucket is free, store the record else use chaining to store the record.

Search

To search a record having the key value k_i , use $h(k_i)$ to compute the address of the bucket where the record is stored.

The bucket may contain one or several records, so check for every record in the bucket to retrieve the desired record with the given key value.

Deletion

To delete a record with key value k_i , use $h(k_i)$ to compute the address of the bucket where the record is stored. The bucket may contain one or several records so check for every record in the bucket, and then delete the record.

B-Tree (Balanced Tree) Indices:

It is impractical to maintain the entire database in the memory, hence B-trees are used to index the data in order to provide fast access.

B-trees are used for its data retrieval speed, ease of maintenance, and simplicity.

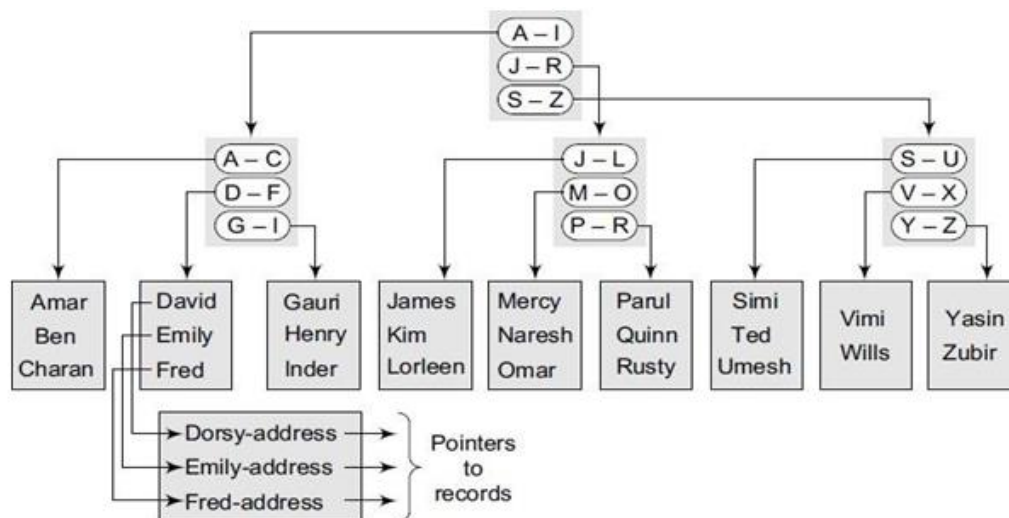


Figure B-tree index

It forms a tree structure with the root at the top. The index consists of a B-tree (balanced tree) structure based on the values of the indexed column.

In this example, the indexed column is name and the B-tree is created using all the existing names that are the values of the indexed column.

The upper blocks of the tree contain index data pointing to the next lower block, thus forming a hierarchical structure. The lowest level blocks, also known as leaf blocks, contain pointers to the data rows stored in the table.

The B-tree structure has the following advantages:

- Since the leaf nodes of a B-tree are at the same depth, retrieval of any record from anywhere in the index takes approximately the same time.
- B-trees improve the performance of a wide range of queries that either search a value having an exact match or for a value within specified range.
- B-trees provide fast and efficient algorithms to insert, update, and delete records that maintain the key order.
- B-trees perform well for small as well as large tables. Their performance does not degrade as the size of a table grows.

B-trees optimize costly disk access.

B+ Tree:

B+ Tree is an extension of B Tree which allows efficient insertion, deletion and search operations.

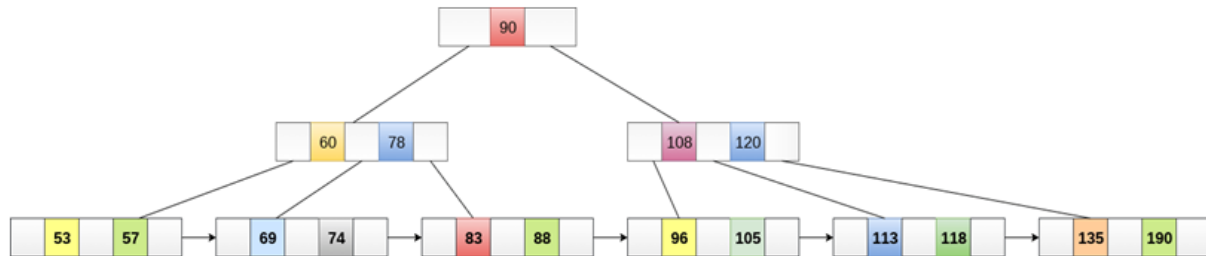
In B Tree, Keys and records both can be stored in the internal as well as leaf nodes. Whereas, in B+ tree, records (data) can only be stored on the leaf nodes while internal nodes can only store the key values.

The leaf nodes of a B+ tree are linked together in the form of a singly linked lists to make the search queries more efficient.

B+ Tree are used to store the large amount of data which can not be stored in the main memory. Due to the fact that, size of main memory is always limited, the internal nodes

(keys to access records) of the B+ tree are stored in the main memory whereas, leaf nodes are stored in the secondary memory.

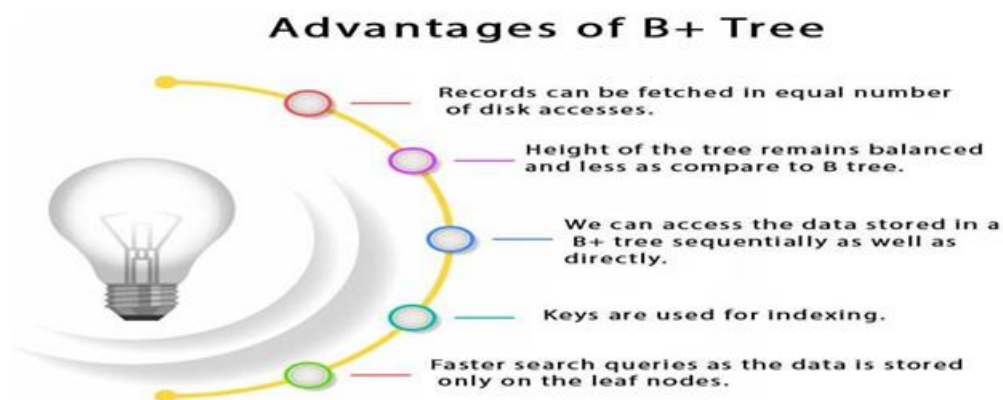
The internal nodes of B+ tree are often called index nodes. A B+ tree of order 3 is shown in the following figure.



Advantages of B+ Tree

Advantages of B+ Tree

1. Records can be fetched in equal number of disk accesses.
2. Height of the tree remains balanced and less as compare to B tree.
3. We can access the data stored in a B+ tree sequentially as well as directly.
4. Keys are used for indexing.
5. Faster search queries as the data is stored only on the leaf nodes.



B Tree VS B+ Tree

SN	B Tree	B+ Tree
1	Search keys can not be repeatedly stored.	Redundant search keys can be present.
2	Data can be stored in leaf nodes as well as internal nodes	Data can only be stored on the leaf nodes.
3	Searching for some data is a slower process since data can be found on internal nodes as well as on the leaf nodes.	Searching is comparatively faster as data can only be found on the leaf nodes.
4	Deletion of internal nodes are so complicated and time consuming.	Deletion will never be a complexed process since element will always be deleted from the leaf nodes.
5	Leaf nodes can not be linked together.	Leaf nodes are linked together to make the search operations more efficient.

Insertion in B+ Tree

Step 1: Insert the new node as a leaf node

Step 2: If the leaf doesn't have required space, split the node and copy the middle node to the next index node.

Step 3: If the index node doesn't have required space, split the node and copy the middle element to the next index page.

Example :

Insert the value 195 into the B+ tree of order 5 shown in the following figure.



195 will be inserted in the right sub-tree of 120 after 190. Insert it at the desired position.



The node contains greater than the maximum number of elements i.e. 4, therefore split it and place the median node up to the parent.



Now, the index node contains 6 children and 5 keys which violates the B+ tree properties, therefore we need to split it, shown as follows.



Deletion in B+ Tree

Step 1: Delete the key and data from the leaves.

Step 2: if the leaf node contains less than minimum number of elements, merge down the node with its sibling and delete the key in between them.

Step 3: if the index node contains less than minimum number of elements, merge the node with the sibling and move down the key in between them.

Example

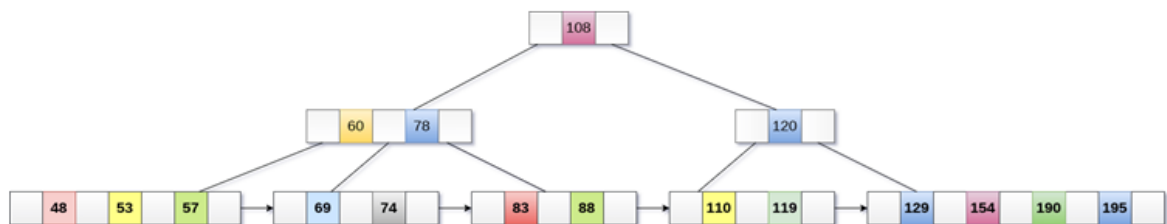
Delete the key 200 from the B+ Tree shown in the following figure.



200 is present in the right sub-tree of 190, after 195. delete it.



Merge the two nodes by using 195, 190, 154 and 129.



Now, element 120 is the single element present in the node which is violating the B+ Tree properties. Therefore, we need to merge it by using 60, 78, 108 and 120.

Now, the height of B+ tree will be decreased by 1.

