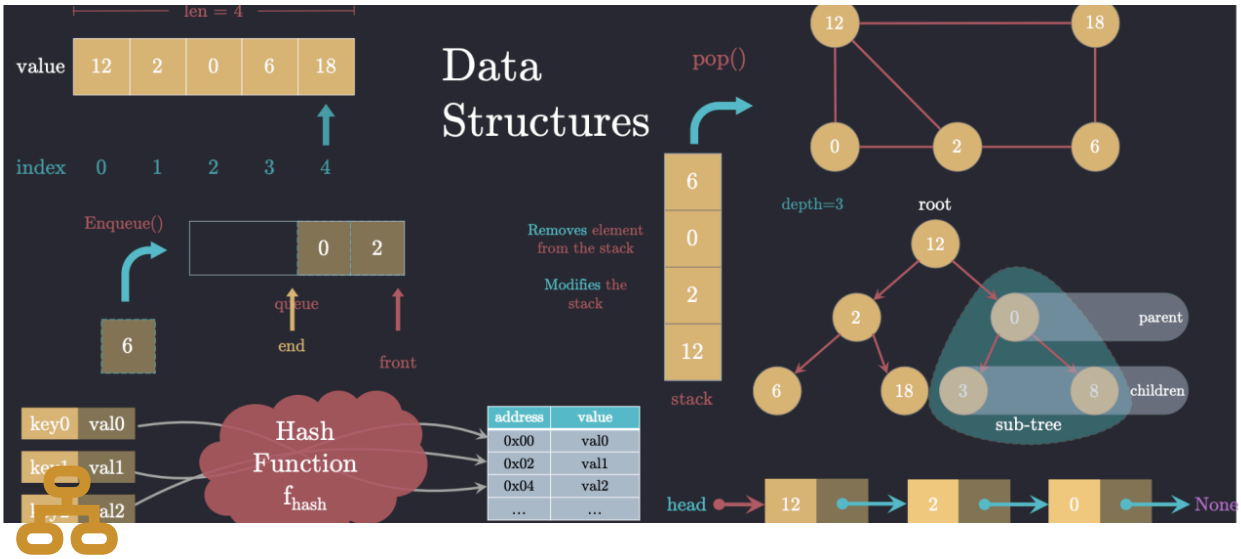


Data Structure

▼ Introduction

▼ what is data structure?

- **Data structures** are the fundamental building blocks of computer programming. They define how data is organized, stored, and manipulated within a program. Understanding data structures is very important for developing efficient and effective algorithms. In this summary, we will explore the most commonly used data structures, including **arrays, linked lists, stacks, queues, trees, and graphs.**
- A **data structure** is a storage that is used to store and organize data. It is a way of arranging data on a computer so that it can be accessed and updated efficiently.

A data structure is not only used for organizing the data. It is also used for processing, retrieving, and storing data. There are different basic and advanced types of data structures that are used in almost every program or

software system that has been developed. So we must have good knowledge about data structures.

▼ Classification of data structure

1. Primitive Data Structures

These are basic data structures that directly operate upon the machine instructions. They include:

- **Integer**
- **Float**
- **Character**
- **Pointer**

2. Non-Primitive Data Structures

These are more complex structures derived from primitive data structures. They are divided into two categories:

a. Linear Data Structures

Elements are arranged in a sequential manner, and each element is connected to its previous and next element. Examples include:

- **Arrays:** A collection of elements identified by index or key.
- **Linked Lists:** A sequence of elements where each element points to the next.
  - **Singly Linked List**
  - **Doubly Linked List**
  - **Circular Linked List**
- **Stacks:** Follows `“Last In First Out (LIFO)”` principle.
- **Queues:** Follows `“First In First Out (FIFO)”` principle.

    - **Simple Queue**
    - **Circular Queue**
    - **Priority Queue**
    - **Dequeue (Double-ended Queue)**

b. Non-Linear Data Structures

Elements are not arranged sequentially, and they might have hierarchical relationships. Examples include:

- **Trees:** A hierarchical structure with a root node and children nodes.
    - **Binary Tree**
    - **Binary Search Tree (BST)**
    - **AVL Tree**
    - **B-tree**
    - **Heap**
    - **Trie**
- **Graphs:** A set of nodes connected by edges.
    - **Directed Graph**
    - **Undirected Graph**
    - **Weighted Graph**
    - **Unweighted Graph**

3. Abstract Data Types (ADT)

These are models for data structures defined by their behavior (operations) rather than their implementation. Examples include:

- **List ADT**
- **Stack ADT**

- **Queue ADT**
- **Map ADT**
- **Set ADT**

4. Storage and Access Data Structures

These are classified based on how they store data and allow access:

- **Linear Storage Structures:** Store data in a linear fashion.
- **Non-Linear Storage Structures:** Store data in a non-linear fashion.

5. Persistent and Ephemeral Data Structures

These are classified based on their ability to preserve previous versions of themselves:

- **Persistent Data Structures:** Allow access to any version of the data structure.
- **Ephemeral Data Structures:** Do not preserve previous versions after modifications.

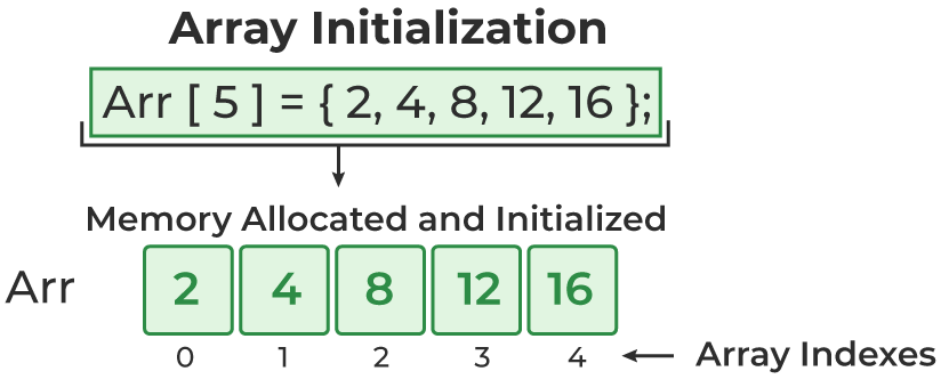
⇒ Each of these data structures has its advantages, disadvantages, and suitable use cases depending on the requirements of the application or algorithm being implemented.

▼ Most Popular Data Structures

1. Array:

An array is a collection of data items stored at contiguous memory locations. The idea is to store multiple items of the same type together. This makes it easier to calculate

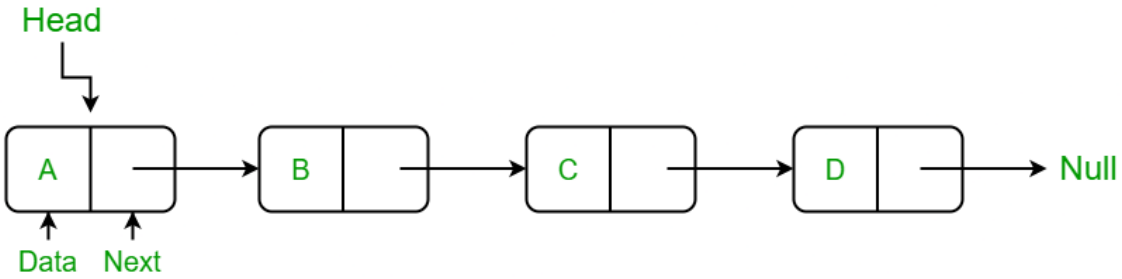
the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).



💡 *Array Data Structure*

2. Linked Lists:

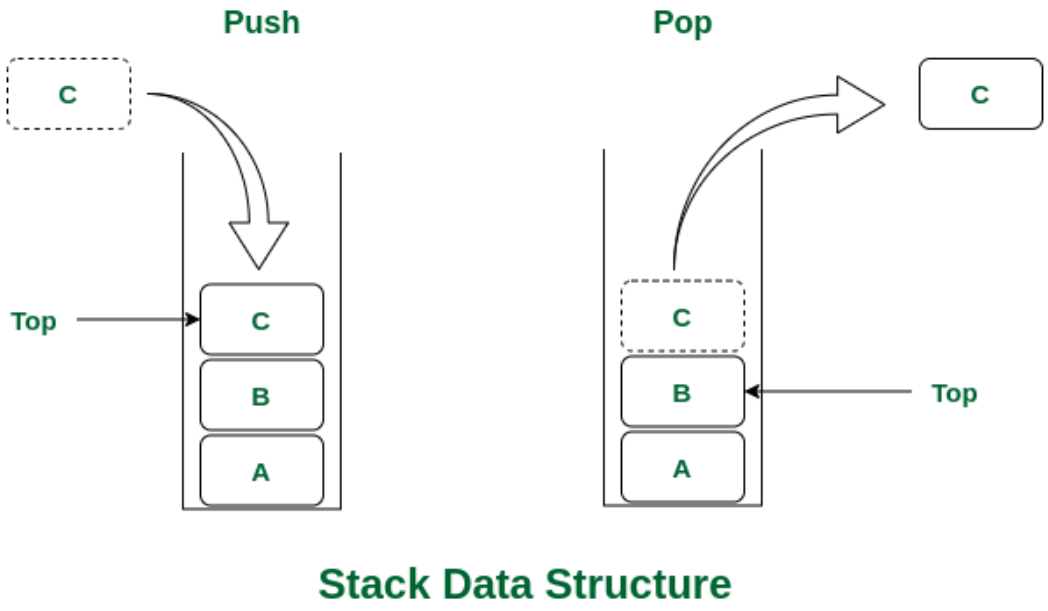
Like arrays, Linked List is a linear data structure. Unlike arrays, linked list elements are not stored at a contiguous location; the elements are linked using pointers.



💡 *Linked Data Structure*

3. Stack:

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



Stack Data Structure

💡 **Stack Operations:**

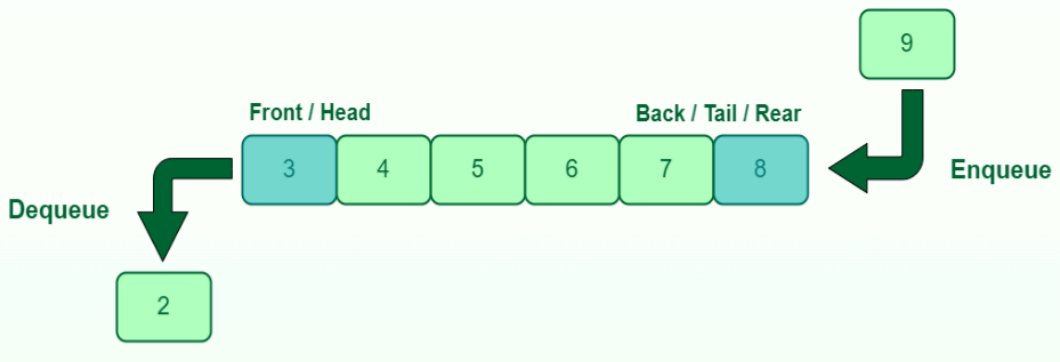
- `push()`: When this operation is performed, an element is inserted into the stack.
- `pop()`: When this operation is performed, an element is removed from the top of the stack and is returned.

- `top()`: This operation will return the last inserted element that is at the top without removing it.

- `size()`: This operation will return the size of the stack i.e. the total number of elements present in the stack.
- `isEmpty()`: This operation indicates whether the stack is empty or not.

4. Queue:

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



Queue Data Structure

> 💡 *Queue Data Structure*

Queue Operations:

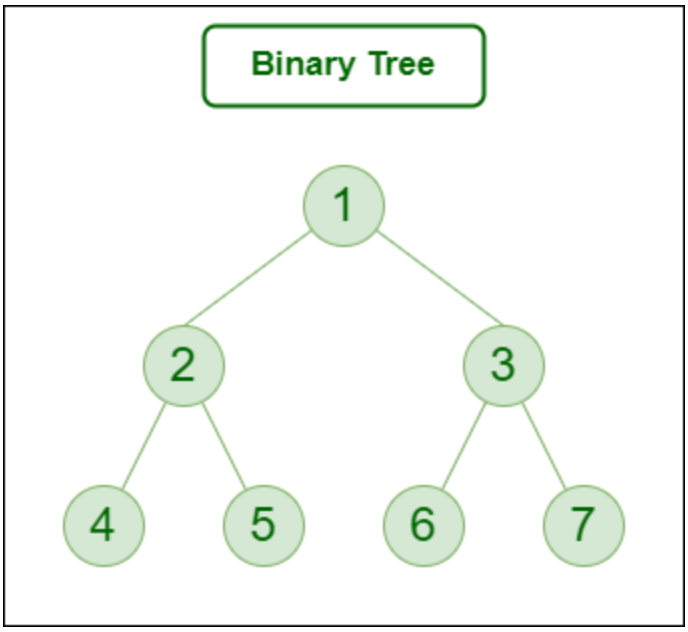
- `Enqueue()`: Adds (or stores) an element to the end of the queue..
- `Dequeue()`: Removal of elements from the queue.
- `Peek()` **or** `front()`: Acquires the data element available at the front node of the queue without deleting it.
- `rear()`: This operation returns the element at the rear end without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

5. Binary Tree:

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

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

> 💡 **1. Data**
> **2. Pointer to left child**
> **3. Pointer to the right child**



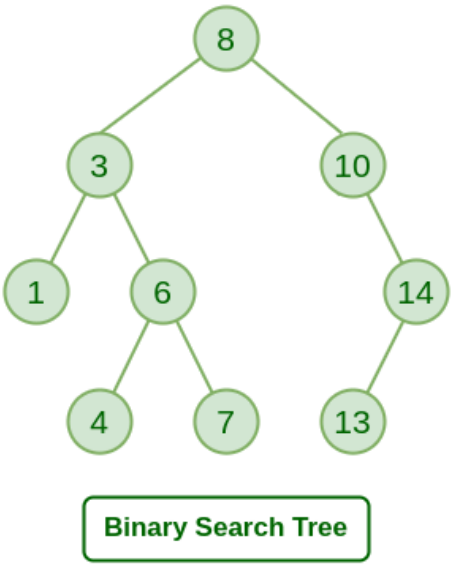
💡 *Binary Tree Data Structure*

6. Binary Search Tree:

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

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

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

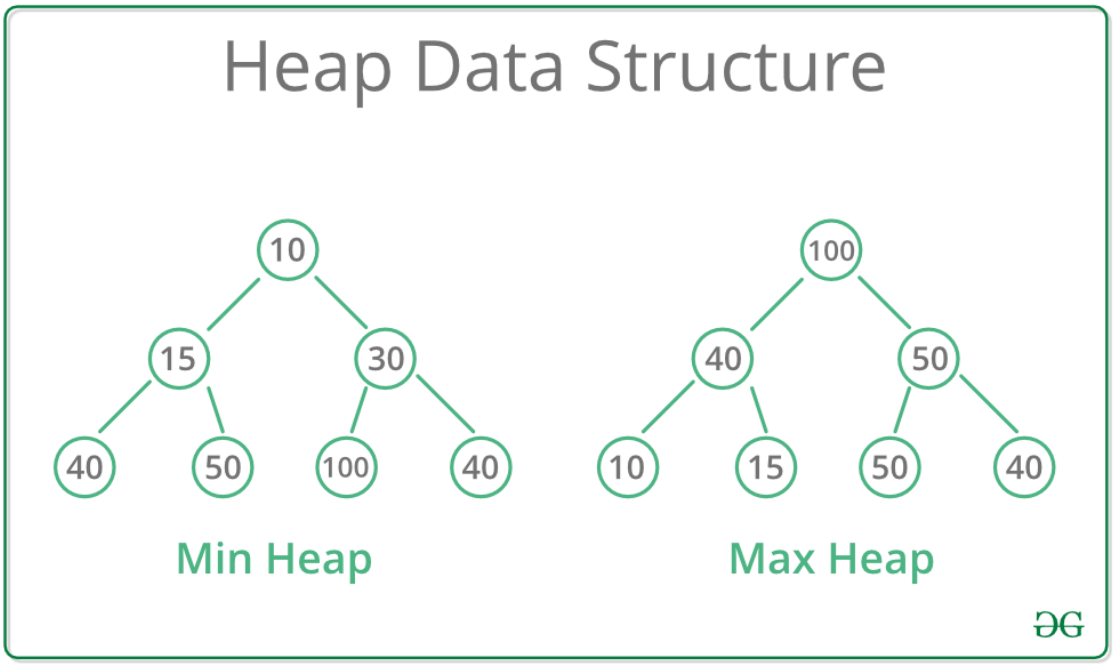


💡 *Binary Search Tree Data Structure*

7. Heap:

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

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

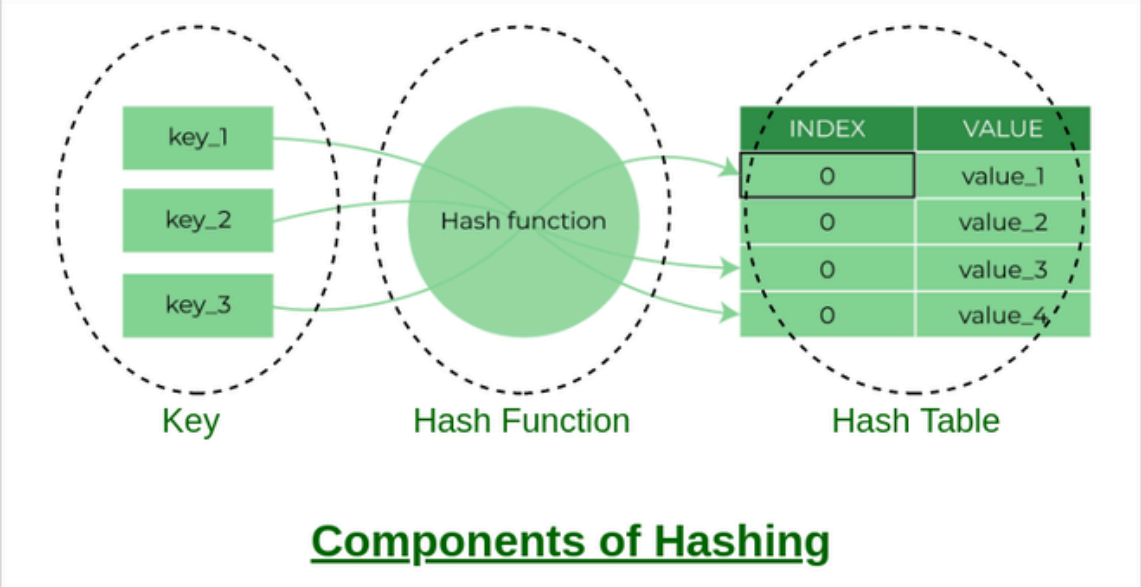


Max and Min Heap

8. Hashing Data Structure:

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

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

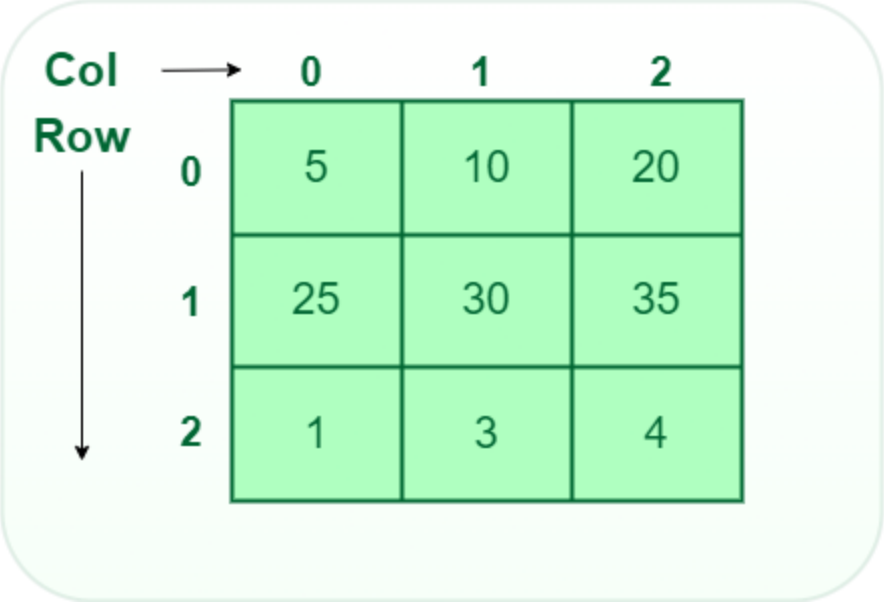


💡 *Hash Data Structure*

9. Matrix:

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

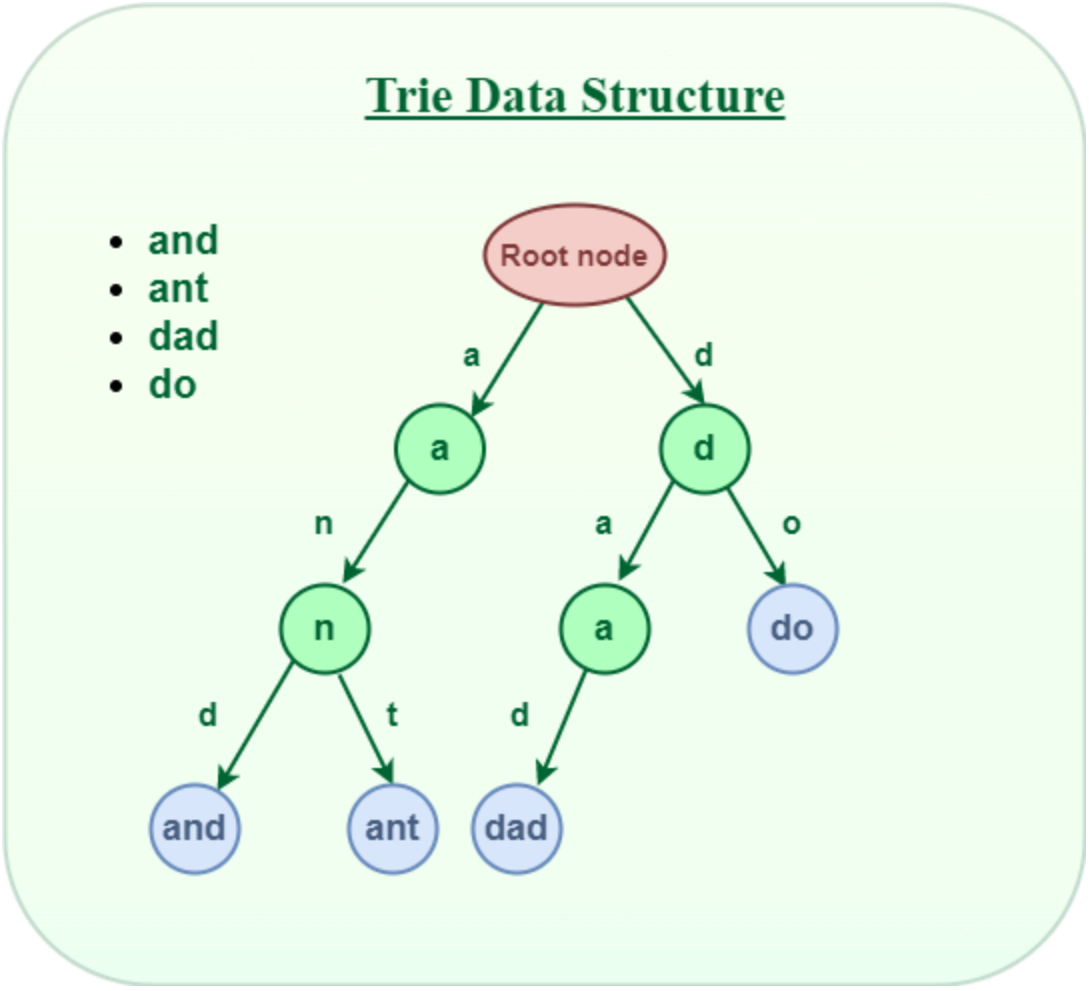
A matrix with 9 elements is shown below.



💡 *Matrix*

10. Trie:

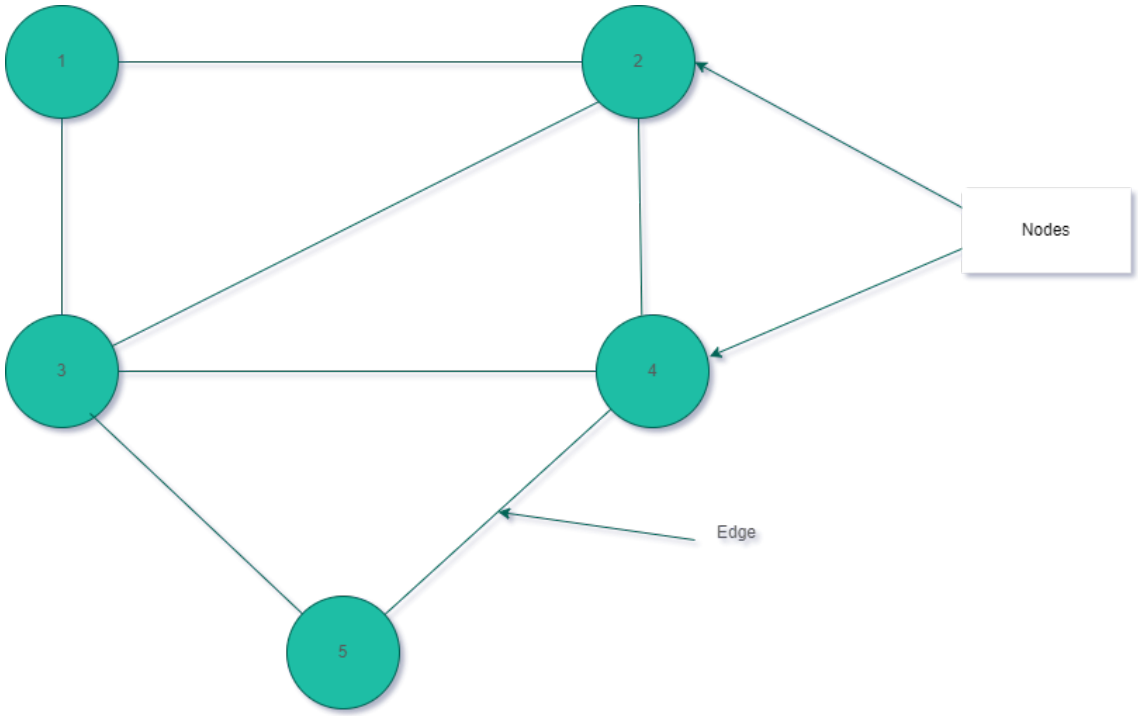
Trie is an efficient information retrieval data structure. Using Trie, search complexities can be brought to an optimal limit (key length). If we store keys in the binary search tree, a well-balanced BST will need time proportional to M * log N, where M is maximum string length and N is the number of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



💡 *Trie Data Structure*

11. Graph:

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



💡 Graph Data Structure

▼ Small article about data structure

What is Data Structure?

The data structure name indicates itself that organizing the data in memory. There are many ways of organizing the data in the memory as we have already seen one of the data structures, i.e., array in C language. Array is a collection of memory elements in which data is stored sequentially, i.e., one after another. In other words, we can say that array stores the elements in a continuous manner. This organization of data is done with the help of an array of data structures. There are also other ways to organize the data in memory. Let's see the different types of data structures.

Backward Skip 10sPlay VideoForward Skip 10s

The data structure is not any programming language like C, C++, java, etc. It is a set of algorithms that we can use in any programming language to structure the data in the memory.

To structure the data in memory, 'n' number of algorithms were proposed, and all these algorithms are known as Abstract data types. These abstract data types are the set of rules.

Types of Data Structures

There are two types of data structures:

- Primitive data structure
- Non-primitive data structure

Primitive Data structure

The primitive data structures are primitive data types. The int, char, float, double, and pointer are the primitive data structures that can hold a single value.

Non-Primitive Data structure

The non-primitive data structure is divided into two types:

- Linear data structure
- Non-linear data structure

Linear Data Structure

The arrangement of data in a sequential manner is known as a linear data structure. The data structures used for this purpose are Arrays, Linked list, Stacks, and Queues. In these data structures, one element is connected to only one another element in a linear form.

When one element is connected to the 'n' number of elements known as a non-linear data structure. The best example is trees and graphs. In this case, the elements are arranged in a random manner.

We will discuss the above data structures in brief in the coming topics. Now, we will see the common operations that we can perform on these data structures.

Data structures can also be classified as:

- **Static data structure:** It is a type of data structure where the size is allocated at the compile time. Therefore, the maximum size is fixed.
- **Dynamic data structure:** It is a type of data structure where the size is allocated at the run time. Therefore, the maximum size is flexible.

Major Operations

The major or the common operations that can be performed on the data structures are:

- **Searching:** We can search for any element in a data structure.
- **Sorting:** We can sort the elements of a data structure either in an ascending or descending order.
- **Insertion:** We can also insert the new element in a data structure.
- **Updation:** We can also update the element, i.e., we can replace the element with another element.
- **Deletion:** We can also perform the delete operation to remove the element from the data structure.

Which Data Structure?

A data structure is a way of organizing the data so that it can be used efficiently. Here, we have used the word efficiently, which in terms of both the space and time. For example, a stack is an ADT (Abstract data type) which uses either arrays or linked list data structure for the

implementation. Therefore, we conclude that we require some data structure to implement a particular ADT.

An ADT tells **what** is to be done and data structure tells **how** it is to be done. In other words, we can say that ADT gives us the blueprint while data structure provides the implementation part. Now the question arises: how can one get to know which data structure to be used for a particular ADT?.

As the different data structures can be implemented in a particular ADT, but the different implementations are compared for time and space. For example, the Stack ADT can be implemented by both Arrays and linked list. Suppose the array is providing time efficiency while the linked list is providing space efficiency, so the one which is the best suited for the current user's requirements will be selected.

Advantages of Data structures

The following are the advantages of a data structure:

- **Efficiency:** If the choice of a data structure for implementing a particular ADT is proper, it makes the program very efficient in terms of time and space.
- **Reusability:** The data structure provides reusability means that multiple client programs can use the data structure.
- **Abstraction:** The data structure specified by an ADT also provides the level of abstraction. The client cannot see the internal working of the data structure, so it does not have to worry about the implementation part. The client can only see the interface.

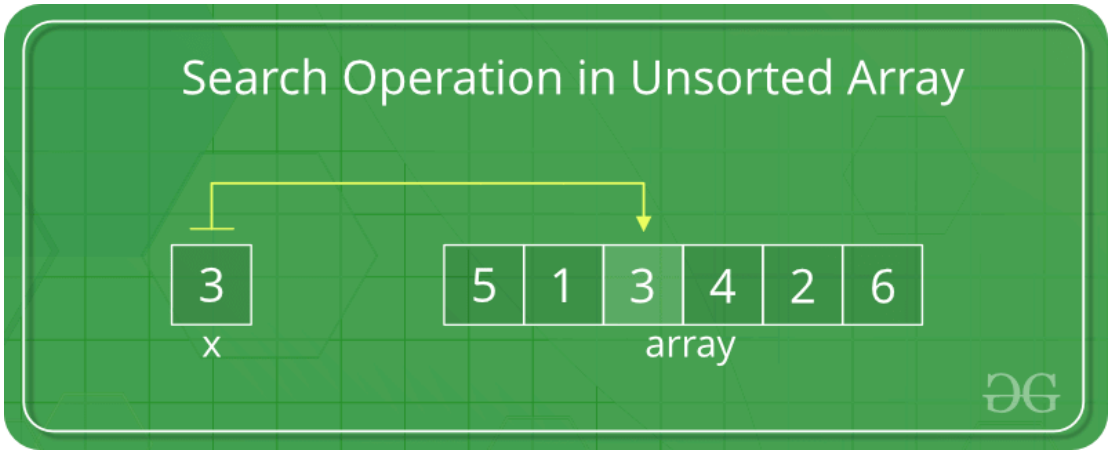
▼ Linear Data Structures

▼ Array Data Structure

▼ Search, Insert, and Delete in an Unsorted Array | Array Operations

Search Operation:

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



Coding implementation of the search operation:

```
/*
   Java program to implement linear search in unsorted array
*/

class Main{
   /*
      Function to implement search operation
   */
   public 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 isn't found
        return -1;
    }
    // Driver's code
    public static void main(String args[]){
    int arr[] = { 12, 13,14,6,75} ;
    int n = arr.length;

    int position = findElement(arr,n,75); // find the position of the elemen
    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)
```

Time Complexity:

O(N)

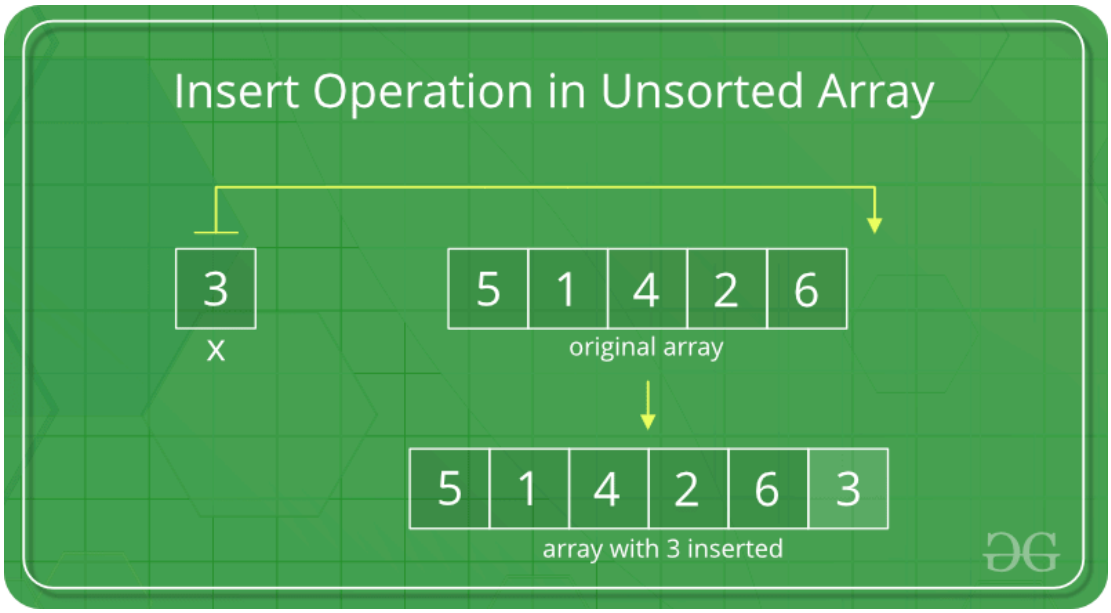
Auxiliary Space:

O(1)

Insert operation:

1. Insert at the end:

In an unsorted array, the insert operation is faster as compared to a sorted array because we don’t have to care about the position at which the element is to be placed.



Coding implementation of inserting an element at the end:

```
/*
   Java program to implement insert operation in an unsorted array.
*/

class Main {
   // Function to insert a given key in
   // the array. This function returns n+1
   // if insertion is successful, else n.
   static int insertSorted(int arr[], int n, int key,
                           int capacity)
   {

      // Cannot insert more elements if n
      // is already more than or equal to
      // capacity
      if (n >= capacity)
         return n;

      arr[n] = key;
```

```
        return (n + 1);
    }

    // Driver Code
    public static void main(String[] args)
    {
        int[] arr = new int[20];
        arr[0] = 12;
        arr[1] = 16;
        arr[2] = 20;
        arr[3] = 40;
        arr[4] = 50;
        arr[5] = 70;
        int capacity = 20;
        int n = 6;
        int i, key = 26;

        System.out.print("Before Insertion: ");
        for (i = 0; i < n; i++)
            System.out.print(arr[i] + " ");

        // Inserting key
        n = insertSorted(arr, n, key, capacity);

        System.out.print("\n After Insertion: ");
        for (i = 0; i < n; i++)
            System.out.print(arr[i] + " ");
    }
}
```

Time Complexity:

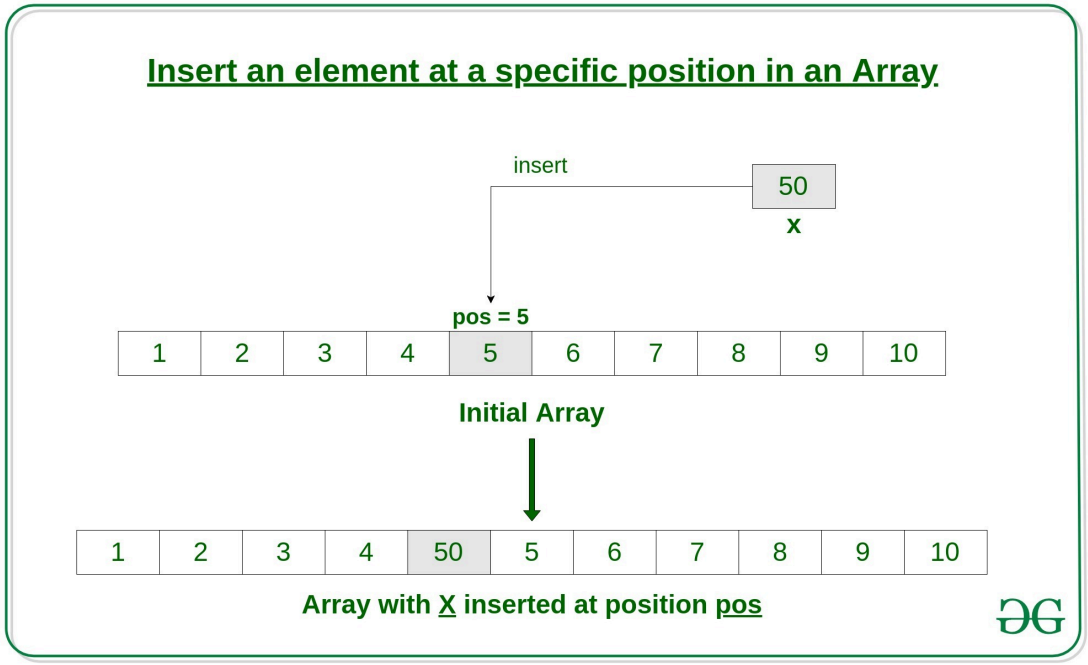
O(1)

Auxiliary Space:

O(1)

2. Insert at any position

Insert operation in an array at any position can be performed by shifting elements to the right, which are on the right side of the required position



Coding implementation of inserting an element at any position:

```
// Java Program to Insert an element
// at a specific position in an Array
class Main {
    static void insertElement(int[] arr, int numberOfElements, int inserted
                              int position)
    {
        // shift elements to the right
        // which are on the right side of position
        for (int i = numberOfElements - 1; i >= position; i--)
            arr[i + 1] = arr[i];
        arr[position] = insertedNumber;
    }
```

```
    public static void main(String[] args)
    {
        int[] arr = new int[15];
        arr[0] = 2;
        arr[1] = 4;
        arr[2] = 1;
        arr[3] = 8;
        arr[4] = 5;
        int numberOfElements = 5;
        int insertedNumber = 10, position = 2/* Index */;

        System.out.print("Before Insertion: ");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");

        // Inserting key at specific position
        insertElement(arr, numberOfElements, insertedNumber, position);
        numberOfElements += 1;

        System.out.print("\n\nAfter Insertion: ");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");
    }
}
```

Time complexity:

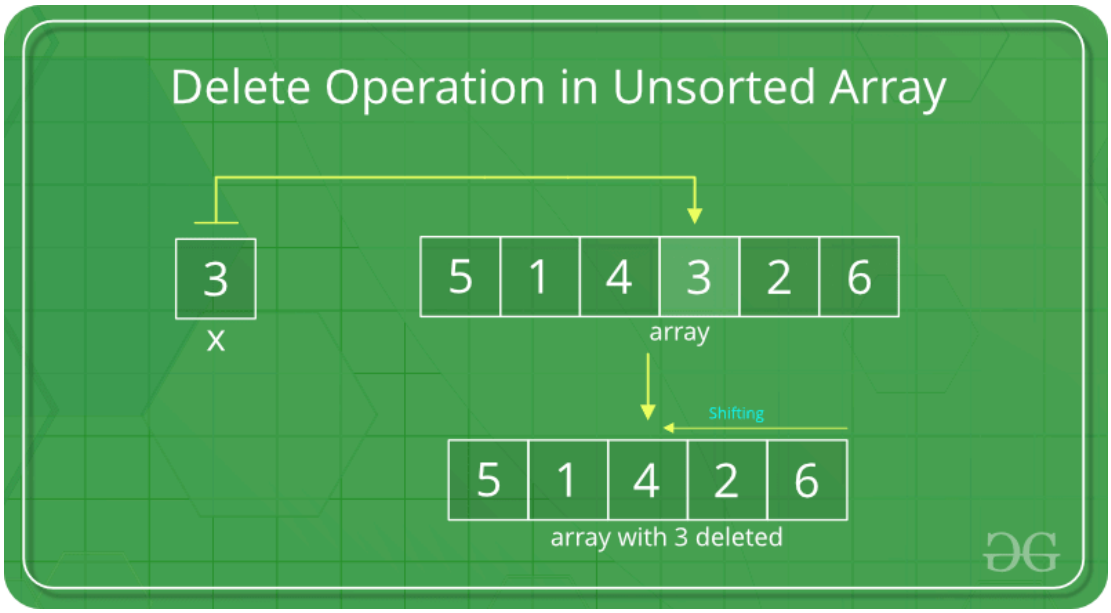
O(N)

Auxiliary Space:

O(1)

Delete Operation:

In the delete operation, the element to be deleted is searched using the **linear search**, and then the delete operation is performed followed by shifting the elements.



code implementation for delete operation:

```
// Java program to implement delete
// operation in an unsorted array

class Main {
    // function to search a key to
    // be deleted
    static int findElement(int arr[], int n, int key)
    {
        int i;
        for (i = 0; i < n; i++)
            if (arr[i] == key)
                return i;

        return -1;
    }

    // Function to delete an element
    static int deleteElement(int arr[], int n, int key)
    {
        // Find position of element to be
```

```
    // deleted
    int pos = findElement(arr, n, key);

    if (pos == -1) {
        System.out.println("Element not found");
        return n;
    }

    // Deleting element
    int i;
    for (i = pos; i < n - 1; i++)
        arr[i] = arr[i + 1];

    return n - 1;
}

// Driver's Code
public static void main(String args[])
{
    int i;
    int arr[] = { 10, 50, 30, 40, 20 };

    int n = arr.length;
    int key = 30;

    System.out.println("Array before deletion");
    for (i = 0; i < n; i++)
        System.out.print(arr[i] + " ");

     // Function call
    n = deleteElement(arr, n, key);

    System.out.println("\n\nArray after deletion");
    for (i = 0; i < n; i++)
        System.out.print(arr[i] + " ");
```

```
  }
}
```

Output

```
Array before deletion
10 50 30 40 20

Array after deletion
10 50 40 20
```

Time Complexity:

O(N)

Auxiliary Space:

O(1)

▼ Search, Insert, and Delete in an Sorted Array | Array Operations

How to Search in a Sorted Array?

In a sorted array, the search operation can be performed by using **binary search** **because of complexity**.

Below is the implementation of the above approach:

```
class Main {
   // function to implement
   // binary search
   public static int binarySearch(int[] arr, int low, int high, int key){
      if(high < low)
         return -1;

      int mid = (low + high) / 2;
```

```
    if(key == arr[mid])
        return mid;
    if(key > arr[mid])
        return binarySearch(arr,mid + 1, high,key);
    return  binarySearch(arr,low,mid - 1,key);
}
// another implementation using loops(iterative statements)
public static int runBinarySearchIteratively( int[] sortedArray, int key,
    int index = Integer.MAX_VALUE;

    while (low <= high) {
        int mid = low  + ((high - low) / 2);
        if (sortedArray[mid] < key) {
            low = mid + 1;
        } else if (sortedArray[mid] > key) {
            high = mid - 1;
        } else if (sortedArray[mid] == key) {
            index = mid;
            break;
        }
    }
    return index;
}
public static void main(String[] args)
{
    int[] arr = { 5, 6, 7, 8, 9, 10 };
    int n, key;
    n = arr.length - 1;
    key = 10;

    // Function call
    System.out.println("Index: "+ binarySearch(arr, 0, n, key));
    System.out.println("Index: "+ runBinarySearchIteratively(arr,key,0,
}
```

```
}
```

Output

```
Index: 5
Index: 5
```

Time Complexity:

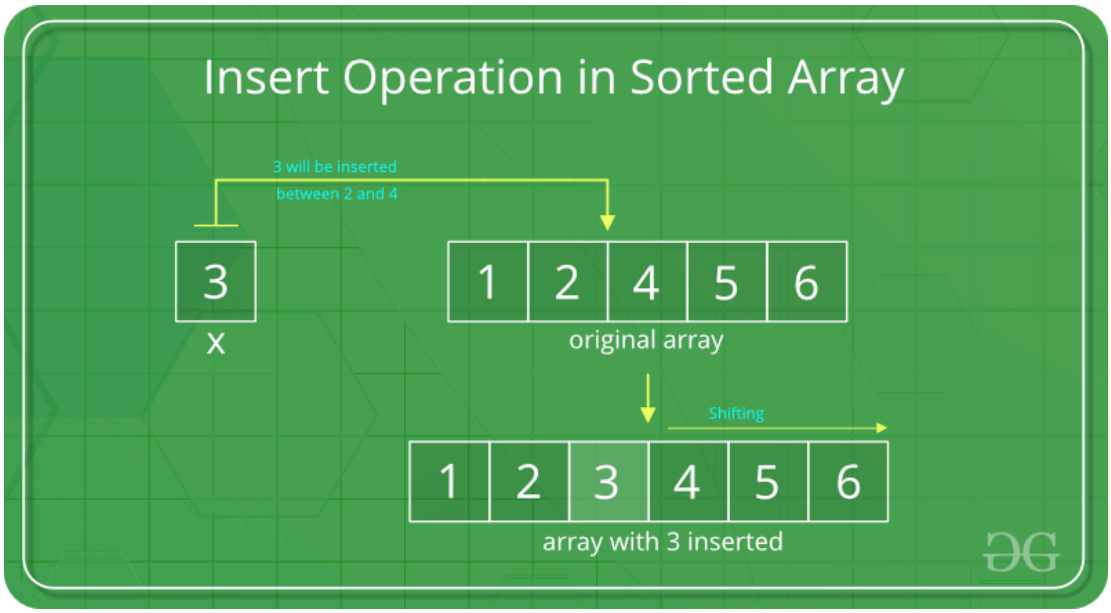
O(log(n)) Using Binary Search

Auxiliary Space:

O(log(n)) due to recursive calls, otherwise iterative version uses Auxiliary Space of O(1).

How to Insert in a Sorted Array?

In a sorted array, a search operation is performed for the possible position of the given element by using **Binary search,** and then an insert operation is performed followed by shifting the elements. And in an unsorted array, the insert operation is faster as compared to the sorted array because we don’t have to care about the position at which the element is placed.



Below is the implementation of the above approach:

```
class Main {
    public static int insertSorted(int[] arr, int numberOfElements, int key,
        if(numberOfElements > capacity)
            return numberOfElements;
        int i;
        for (i = numberOfElements - 1; (i >= 0 && arr[i] > key); i--)
            arr[i + 1] = arr[i];

        arr[i + 1] = key;
        return numberOfElements + 1;
    }

    public static void main(String[] args)
    {
        int[] arr = new int[20];
        arr[0] = 12;
        arr[1] = 16;
        arr[2] = 20;
        arr[3] = 40;
```

```
        arr[4] = 50;
        arr[5] = 70;
        int capacity = arr.length;
        int numberOfElements = 6;
        int key = 26;

        System.out.print("\nBefore Insertion: ");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");

        // Function call
        numberOfElements = insertSorted(arr, numberOfElements, key, ca

        System.out.print("\nAfter Insertion: ");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");
    }

}
```

Output

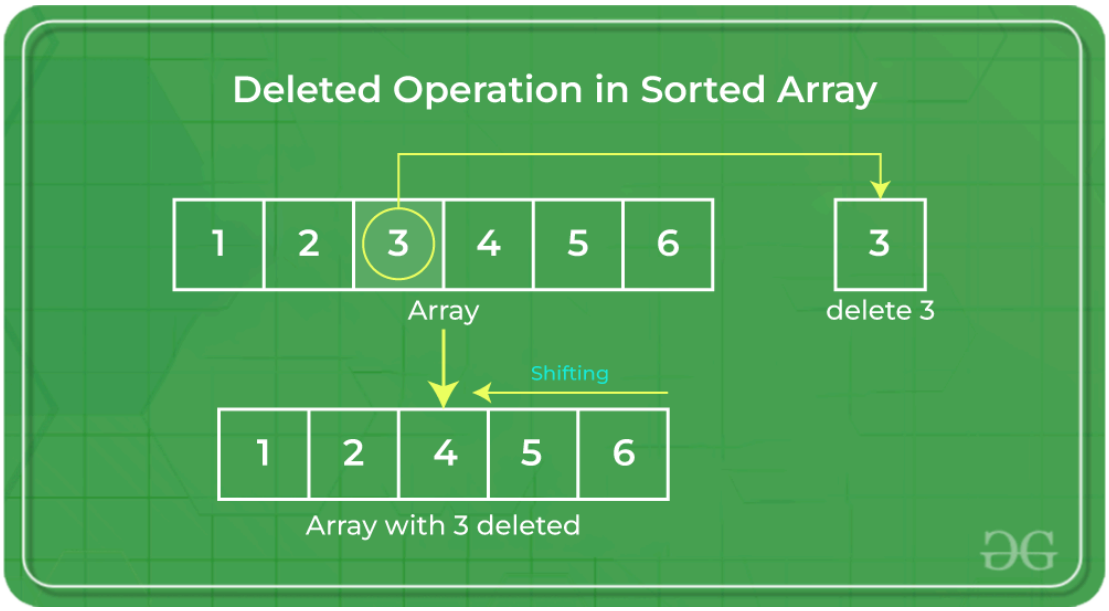
```
Before Insertion: 12 16 20 40 50 70
After Insertion: 12 16 20 26 40 50 70
```

Time Complexity: O(N) [In the worst case all elements may have to be moved]

Auxiliary Space: O(1)

How to Delete in a Sorted Array?

In the delete operation, the element to be deleted is searched using binary search, and then the delete operation is performed followed by shifting the elements.



Below is the implementation of the above approach:

```
class Main {
    public static int binarySearch(int[] arr, int low, int high, int key){
        if(high < low)
            return -1;
        int mid = (low + high) / 2;
        if(key == arr[mid])
            return mid;
        else if(key > arr[mid])
            return binarySearch(arr,mid + 1, high, key);
        return binarySearch(arr,low, mid - 1,key);
    }
    public static int deleteElement(int[] arr, int numberOfElements,int key
        int position  = binarySearch(arr,0,numberOfElements - 1,key);
        if(position == -1) {
            System.out.println("Element Not Found!");
            return numberOfElements;
        }
        // Delete the element
        int i;
```

```
        for( i = position;i < numberOfElements - 1;i++){
            arr[i] = arr[i + 1];
        }
        return numberOfElements - 1;
    }
    //driver Code
    public static void main(String[] args) {
        int[] arr = { 10, 20, 30, 40, 50 };
        int numberOfElements = arr.length;
        int key = 20;
        System.out.print("Array before deletion:\n");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");

        // Function call
        numberOfElements = deleteElement(arr, numberOfElements, key);

        System.out.print("\n\nArray after deletion:\n");
        for (int i = 0; i < numberOfElements; i++)
            System.out.print(arr[i] + " ");
    }

}
```

Output

```
Array before deletion
10 20 30 40 50

Array after deletion
10 20 40 50
```

Time Complexity: O(N). In the worst case all elements may have to be moved

Auxiliary Space: O(log N). An implicit stack will be used

▼ ArrayList Data Structure

Key Concepts of the `ArrayList` Class

1. **Dynamic Resizing**:
    - Internally, `ArrayList` uses an array to store elements. When the array becomes full, `ArrayList` creates a new array with a larger capacity and copies the elements from the old array to the new one.
    - The default initial capacity of an `ArrayList` is usually 10, but this can be specified by the user.
2. **Index-Based Access**:
    - `ArrayList` allows random access to its elements. This means you can retrieve or update elements quickly using an index, similar to how you would with a regular array.
3. **Automatic Memory Management**:
    - `ArrayList` manages the underlying array's size and ensures there is enough room to add new elements. If there is not enough space, the `ArrayList` grows automatically.
    - When an element is removed, `ArrayList` may shrink if a large amount of extra space is left unused.
4. **Type Safety and Generics**:
    - `ArrayList` can be defined to hold a specific type of object using **generics**, ensuring type safety at compile time. For example, `ArrayList<Integer>` creates a list that can only store `Integer` objects.
    - Generics prevent runtime errors by checking types at compile time.

Common Methods in the `ArrayList` Class

1. `add(E element)`: Adds the specified element to the end of the list.
2. `add(int index, E element)`: Inserts the specified element at the specified position in the list.
3. `get(int index)`: Returns the element at the specified position in the list.
4. `set(int index, E element)`: Replaces the element at the specified position with the specified element.
5. `remove(int index)`: Removes the element at the specified position in the list.
6. `remove(Object o)`: Removes the first occurrence of the specified element from the list.
7. `size()`: Returns the number of elements in the list.
8. `clear()`: Removes all elements from the list.
9. `isEmpty()`: Returns `true` if the list contains no elements.
10. `contains(Object o)`: Returns `true` if the list contains the specified element.
11. `toArray()`: Converts the `ArrayList` to an array.

Time Complexity Analysis

- **Access (`get` and `set` operations)**: `O(1)` because `ArrayList` allows random access to elements.
- **Add**: `O(1)` on average, but can be `O(n)` in the worst case when the array is resized.
- **Remove**: `O(n)` because elements may need to be shifted to fill the gap.

Code Implementation

```
import java.util.Arrays;

public class MyArrayList<E> {
    // Initial capacity of the ArrayList
    private static final int DEFAULT_CAPACITY = 10;
    private Object[] elements; // Array to store the elements
    private int size = 0; // Number of elements in the ArrayList

    // Constructor to initialize the ArrayList with the default capacity
    public MyArrayList() {
        elements = new Object[DEFAULT_CAPACITY];
    }

    // Constructor to initialize the ArrayList with a specified capacity
    public MyArrayList(int initialCapacity) {
        if (initialCapacity <= 0) {
            throw new IllegalArgumentException("Capacity must be greater tha
        }
        elements = new Object[initialCapacity];
    }

    // Method to add an element to the end of the ArrayList
    public void add(E element) {
        ensureCapacity();
        elements[size++] = element;
    }

    // Method to add an element at a specific index
    public void add(int index, E element) {
        if (index < 0 || index > size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size:
        }
        ensureCapacity();
        System.arraycopy(elements, index, elements, index + 1, size - index);
        elements[index] = element;
```

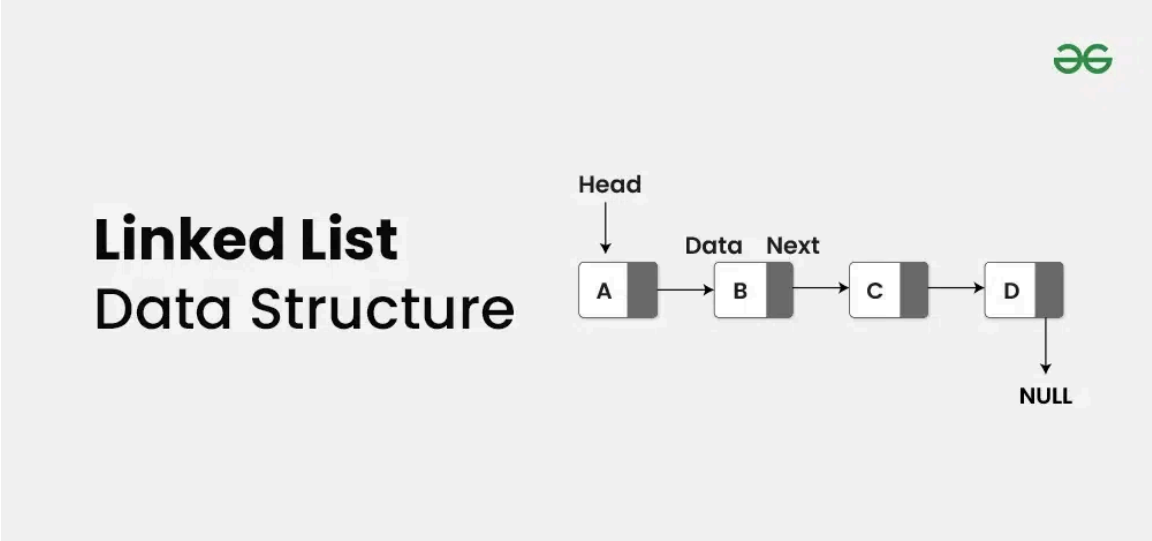
```
        size++;
    }

    // Method to get an element at a specific index
    @SuppressWarnings("unchecked")
    public E get(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size:
        }
        return (E) elements[index];
    }

    // Method to set an element at a specific index
    public void set(int index, E element) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size:
        }
        elements[index] = element;
    }

    // Method to remove an element at a specific index
    @SuppressWarnings("unchecked")
    public E remove(int index) {
        if (index < 0 || index >= size) {
            throw new IndexOutOfBoundsException("Index: " + index + ", Size:
        }
        E removedElement = (E) elements[index];
        int numMoved = size - index - 1;
        if (numMoved > 0) {
            System.arraycopy(elements, index + 1, elements, index, numMoved
        }
        elements[--size] = null; // Clear the last element to avoid memory leak
        return removedElement;
    }

    // Method to get the size of the ArrayList
```

```
    public int size() {
        return size;
    }

    // Method to check if the ArrayList is empty
    public boolean isEmpty() {
        return size == 0;
    }

    // Method to check if the ArrayList contains a specific element
    public boolean contains(E element) {
        return indexOf(element) >= 0;
    }

    // Method to get the index of a specific element
    public int indexOf(E element) {
        for (int i = 0; i < size; i++) {
            if (element.equals(elements[i])) {
                return i;
            }
        }
        return -1;
    }

    // Method to clear all elements from the ArrayList
    public void clear() {
        Arrays.fill(elements, 0, size, null);
        size = 0;
    }

    // Helper method to ensure there is enough capacity to add new elemen
    private void ensureCapacity() {
        if (size == elements.length) {
            int newCapacity = elements.length * 2;
            elements = Arrays.copyOf(elements, newCapacity);
        }
```

```
    }

    // Main method to demonstrate the functionality of MyArrayList
    public static void main(String[] args) {
        MyArrayList<Integer> myList = new MyArrayList<>();
        myList.add(10);
        myList.add(20);
        myList.add(30);
        System.out.println("Element at index 1: " + myList.get(1)); // Output: 2(
        myList.add(1, 15);
        System.out.println("Element at index 1 after insertion: " + myList.get(1
        myList.remove(2);
        System.out.println("Size after removal: " + myList.size()); // Output: 3
        System.out.println("Is the list empty? " + myList.isEmpty()); // Output:
    }
}
```

▼ Linked list Data Structure (Not Ended)

A **linked list** is a fundamental data structure in computer science. It consists of nodes where each node contains **data** and a **reference (link)** to the next node in the sequence. This allows for dynamic memory allocation and efficient **insertion** and **deletion** operations compared to arrays.



Linked List Data Structure

▼ What is a Linked List?

A **linked list** is a linear data structure that consists of a series of nodes connected by pointers. Each node contains **data** and a **reference** to the next node in the list. Unlike **arrays, linked lists** allow for efficient **insertion** or **removal** of elements from any position in the list, as the nodes are not stored contiguously in memory.

▼ Major differences between array and linked

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

Linked List:

- **Data Structure:** Non-contiguous
- **Memory Allocation:** Dynamic
- **Insertion/Deletion:** Efficient
- **Access:** Sequential

Array:

- **Data Structure:** Contiguous
- **Memory Allocation:** Static
- **Insertion/Deletion:** Inefficient
- **Access:** Random

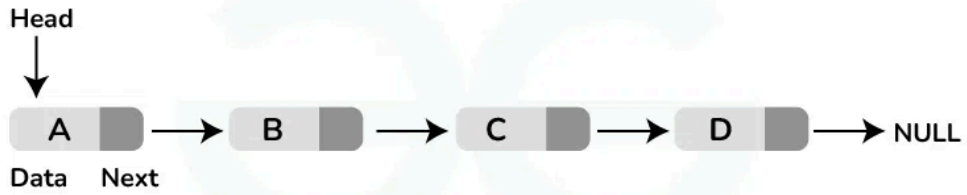
▼ Types of Linked List

▼ Singly Linked list

What is Singly Linked List?

> **A singly linked list is a linear data structure in which the elements are not stored in contiguous memory locations and**

> **each element is connected only to its next element using a pointer.**



Singly Linked List

Understanding Node Structure

In a singly linked list, each node consists of two parts: data and a pointer to the next node. The data part stores the actual information, while the pointer (or reference) part stores the address of the next node in the sequence. This structure allows nodes to be dynamically linked together, forming a chain-like sequence.

Node Structure of Singly Linked List

In most programming languages, a node in a singly linked list is typically defined using a class or a struct.

```
// Definition of a Node in a singly linked list
public class Node <T>{
    // Data part of the node
    T data;
```

```
    // Pointer to the next node in the list
    Node <T> next;

    // Constructor to initialize the node with data
    public Node(T 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.

Operations on Singly Linked List

- **Printing**
- **Searching**
- **Length**
- **Insertion:**
  - Insert at the beginning
  - Insert at the end
  - Insert at a specific position
- **Deletion:**
  - Delete from the beginning
  - Delete from the end
  - Delete a specific node

LinkedList class:

Next, create the `LinkedList` class that uses the `Node` class.

```
public class LinkedList <T>{
    private Node<T> head;
    public LinkedList(){
        this.head = null;
    }

    // add a new node at the beginning of the linked list
    public void addFirst(T data){
        Node<T> newNode = new Node<>(data);
        newNode.next = head;
        head = newNode;
    }

    // add a new node at the end of the linked list
    public void addLast(T data){
        Node<T> newNode = new Node<>(data);

        if(head == null)
            head = newNode;
        else {
            Node<T> current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }

    // insert a data at a specific position

    public void insertAt(int index,T data){
```

```
    if(index < 0)
        throw new IndexOutOfBoundsException();
    Node<T> newNode = new Node<>(data);
    if(index == 0){
        newNode.next = head;
        head = newNode;
        return;
    }

    Node<T> current = head;
    for(int i = 0; i < index - 1;i++){
        if(current == null) throw new IndexOutOfBoundsException();
        current = current.next;
    }
    newNode.next = current.next;
    current.next = newNode;
}
/*
* Removing Methods
* */

// remove the first element of the linked list
public void removeFirst(){
    if(head != null){
            head = head.next;
    }
}

// remove the last element of the linked list
public void removeLast(){
    if(head == null) return;

    if(head.next == null){
        head = null;
    }
    else{
```

```
        Node<T> current = head;
        while(current.next.next != null){
            current = current.next;
        }
        current.next = null;
    }
}

// delete a node with the given data
public void delete(T data){
    if(head == null) return;
    if(head.data.equals(data)){
        head = head.next;
        return;
    }

    Node<T> current = head;
    while(current.next != null && !current.next.data.equals(data))
        current = current.next;
    if(current.next != null){
        current.next = current.next.next;
    }
}

// delete a node at a specific index
public void deleteAt(int index){
    if(index < 0 || head == null) throw new IndexOutOfBoundsExce

    if(index == 0){
        head = head.next;
        return;
    }

    Node<T> current = head;
    for(int i = 0; i < index - 1; i++){
        if(current == null || current.next == null) throw new IndexOut
```

```
            current = current.next;
        }

        if(current.next != null){
            current.next = current.next.next;
        }
        else{
            throw new IndexOutOfBoundsException();
        }
    }

    // Search for a node with the given data
    public boolean search(T data) {
        Node<T> current = head;
        while (current != null) {
            if (current.data.equals(data)) {
                return true;
            }
            current = current.next;
        }
        return false;
    }

    // Get the length of the list
    public int length() {
        int count = 0;
        Node<T> current = head;
        while (current != null) {
            count++;
            current = current.next;
        }
        return count;
    }

    // print the list
```

```
    public void printList(){
        Node<T> current = head;
        while(current != null){
            System.out.print(current.data + " → ");
            current = current.next;
        }
        System.out.println("null");
    }
}
```

example of linked list:

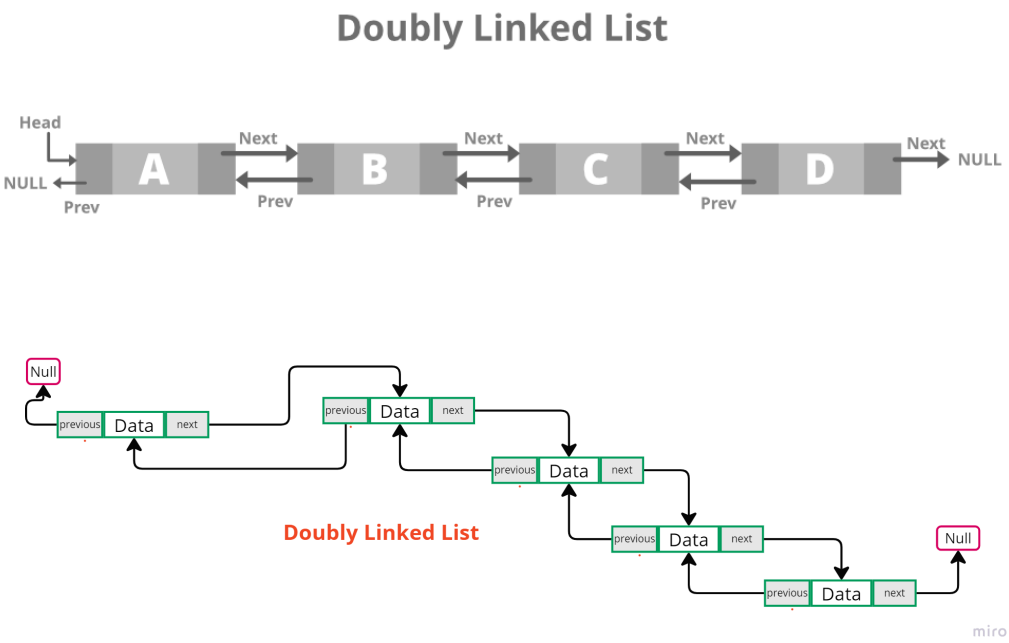
```
public class Main {
    public static void main(String[] args) {
        LinkedList<Integer> intList = new LinkedList<>();
        intList.addFirst(3);
        intList.addFirst(2);
        intList.addFirst(1);
        intList.printList(); // Output: 1 → 2 → 3 → null

        intList.addLast(4);
        intList.addLast(5);
        intList.printList(); // Output: 1 → 2 → 3 → 4 → 5 → null

        intList.removeFirst();
        intList.printList(); // Output: 2 → 3 → 4 → 5 → null

        intList.removeLast();
        intList.printList(); // Output: 2 → 3 → 4 → null

        intList.delete(3);
        intList.printList(); // Output: 2 → 4 → null

        intList.insertAt(1, 3);
        intList.printList(); // Output: 2 → 3 → 4 → null
```

```
        intList.deleteAt(1);
        intList.printList(); // Output: 2 → 4 → null

        LinkedList<String> strList = new LinkedList<>();
        strList.addFirst("d");
        strList.addFirst("e");
        strList.addFirst("m");
        strList.addFirst("a");
        strList.addFirst("h");
        strList.addFirst("o");
        strList.addFirst("m");
        strList.printList(); // Output: m → o → h → a → m → e → d → nul

    }
}
```

▼ Doubly-Linked list

> A doubly linked list or a two-way linked list is a more complex type of linked list that contains a pointer to the next as well as the previous node in sequence.

Therefore, it contains three parts of data, a pointer to the next node, and a pointer to the previous node. This would enable us to traverse the list in the backward direction as well. Below is the image for the same:



Structure of Doubly Linked List:

In a doubly linked list:

- Each node points to both its previous and next node, unlike a singly linked list which only points to the next node.
- The head node is the first node in the list, and it has a null reference in its previous field.
- The tail node is the last node in the list, and it has a null reference in its next field.

Operations of doubly linked list:

1. Add Nodes

- `addFront(T data)` :
  - Adds a new node with the given data at the front (head) of the list.
  - The new node becomes the new head of the list.

- `addEnd(T data)`:
  - Adds a new node with the given data at the end (tail) of the list.
  - The new node becomes the new tail of the list.

2. Insert Nodes

- `insertAt(int index, T data)`:
  - Inserts a new node with the given data at the specified index.
  - If the index is 0, it adds the node at the front.
  - Adjusts pointers to insert the node at the correct position.

3. Delete Nodes

- `deleteFront()` :
  - Deletes the node at the front (head) of the list.
  - The next node becomes the new head.
- `deleteEnd()`:
  - Deletes the node at the end (tail) of the list.
  - Adjusts pointers to remove the last node.
- `deleteAt(int index)`:
  - Deletes the node at the specified index.
  - Adjusts pointers to remove the node at the correct position.
- `delete(T data)`:
  - Deletes the first node found with the specified data.

- Adjusts pointers to remove the node and maintain the list.

4. Display Nodes

- `display()`:
  - Traverses the list and prints the data of each node.
  - Indicates the end of the list with "END".

5. Reverse List

- `reverse()`:
  - Reverses the order of the nodes in the list.
  - Adjusts pointers to reverse the direction of traversal.

Implementation in Java

Below is the Java implementation of a doubly linked list with methods for inserting, deleting, and displaying nodes:

```
// Node class
class Node<T>{
      T data;
      Node<T> next;
   Node<T> prev;
   public Node(T data){
      this.data = data;
      next = null;
      prev = null;
   }
}
// Doubly linked list class
public class DoublyLinkedList <T>{
```

```
    Node<T> head;

public void addFront(T data){
    Node<T> newNode = new Node<>(data);
    if(head != null){
        head.prev = newNode;
        newNode.next = head;
    }
    head = newNode;
}
public void addEnd(T data) {
    Node<T> newNode = new Node<>(data);
    if (head == null) {
        head = newNode;
        return;
    }
    Node<T> current = head;
    while (current.next != null)
        current = current.next;
    current.next = newNode;
    newNode.prev = current;

}

// Method to insert at a specific index
public void insertAt (int index,T data){
    Node<T> newNode = new Node<>(data);
    if(index < 0)
        throw new IndexOutOfBoundsException("Index out of bound
    Node<T> current = head;
    for(int i = 0; i < index - 1;i++){
        if(current == null)
            throw new IndexOutOfBoundsException("Index out of bou
        current = current.next;
    }
    newNode.next = current.next;
```

```
        newNode.prev = current;
        if(current.next != null){
            current.next.prev = newNode;
        }
        current.next = newNode;
    }

    // Method to delete a node from the front of the list
    public void deleteFront(){
        if(head == null) {
            System.out.println("The list is already empty");
            return;
        }
        head = head.next;
        if(head != null)
            head.prev = null;

    }

    // Method to delete a node from the end of the list
    public void deleteEnd(){
        if(head == null){
            System.out.println("The list is already empty");
            return;
        }
        if(head.next == null){
            head = null;
            return;
        }
        Node<T> current = head;
        while(current.next != null)
            current = current.next;
        current = null;
    }

    // Method to delete a node at a given index
```

```
public void deleteAt(int index){
    if(head == null){
        System.out.println("List is already empty");
        return;
    }
    if(index < 0)
        throw new IndexOutOfBoundsException();
    if(index == 0){
        deleteFront();
    }
    Node<T> current = head;
    for(int i = 0; i < index; i++){
        if(current == null)
            throw new IndexOutOfBoundsException();
        current = current.next;
    }
    if(current.prev != null) {
        current.prev.next = current.next;
    }
    if(current.next != null)
        current.next.prev = current.prev;
}

// Method to delete a node by value
public void delete(T data){

    if(head == null)
        throw new IndexOutOfBoundsException();

    Node<T> current = head;
    while(current != null && ! current.data.equals(data)){
        current = current.next;
    }

    if(current == null) {
        System.out.println("Data not found in the list");
```

```
            return;
        }
        if(current.prev != null)
            current.prev.next = current.next;
        else
            head = current.next;
        if(current.next != null)
            current.next.prev = current.prev;
    }

    // Method to display the nodes of the list
    public void display(){
        if(head == null) {
            System.out.println("List is empty");
            return;
        }
        Node<T> current = head;
        while(current != null){
            System.out.print(current.data + " → ");
            current = current.next;
        }
        System.out.println("END");
    }

    // Method to reverse the list
    public void reverse(){
        Node<T> current = head;
        Node<T> temp = current.next;

        current.next = null;
        current.prev = temp;
        while(temp != null){
            temp.prev = temp.next;
            temp.next = current;
            current = temp;
            temp = temp.prev;
```

```
        }
        head = current;
    }
}
```

Example Usage

Below is an example of how you might use these operations in the `DoublyLinkedList` class:

```
public class Main {
    public static void main(String[] args) {
        DoublyLinkedList<Integer> dll = new DoublyLinkedList<>();

        // Adding nodes
        dll.addFront(10); // List: 10 → END
        dll.addEnd(20);   // List: 10 → 20 → END
        dll.addFront(5);  // List: 5 → 10 → 20 → END

        // Display nodes
        dll.display(); // Output: 5 → 10 → 20 → END

        // Insert node at index 1
        dll.insertAt(1, 15); // List: 5 → 15 → 10 → 20 → END
        dll.display();       // Output: 5 → 15 → 10 → 20 → END

        // Delete node at index 2
        dll.deleteAt(2); // List: 5 → 15 → 20 → END
        dll.display();   // Output: 5 → 15 → 20 → END

        // Delete node with data 15
        dll.delete(15); // List: 5 → 20 → END
        dll.display();  // Output: 5 → 20 → END

        // Reverse the list
        dll.reverse(); // List: 20 → 5 → END
```

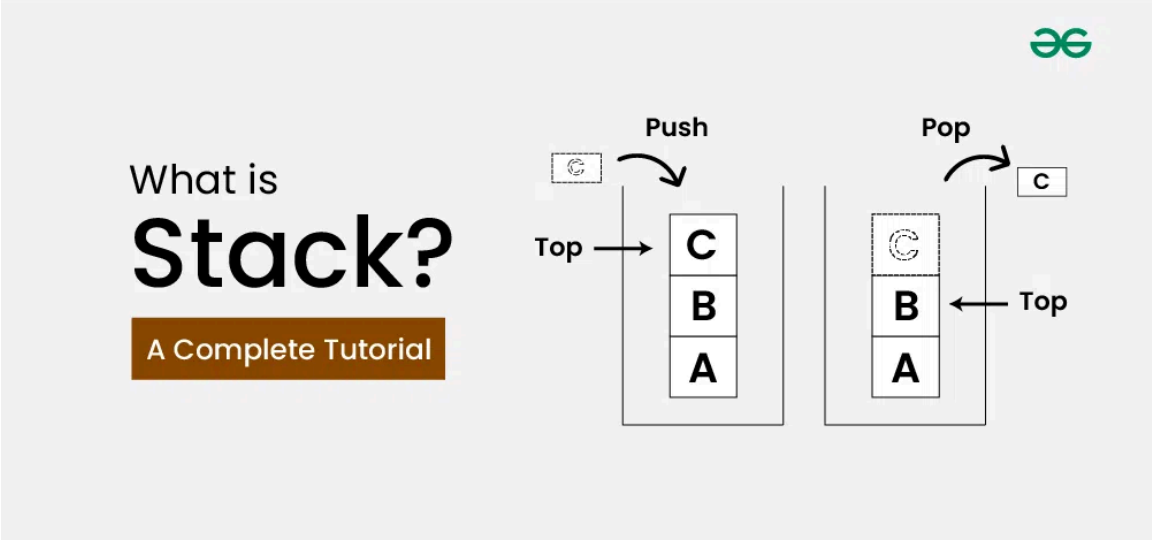
```
        dll.display(); // Output: 20 → 5 → END
    }
}
```

- **Circular Linked list**
- **Double-Circular linked list**
- **Header linked list**

Stack Data Structure

What is Stack Data Structure?

Stack Data Structure is a **linear data structure** _that follows **LIFO (Last In First Out) Principle** , so the last element inserted is the first to be popped out. In this article, I will cover all the basics of Stack, Operations on Stack, its implementation, advantages, disadvantages which will help you solve all the problems based on Stack.



To implement the stack, it is required to maintain the **pointer to the top of the stack** , which is the last

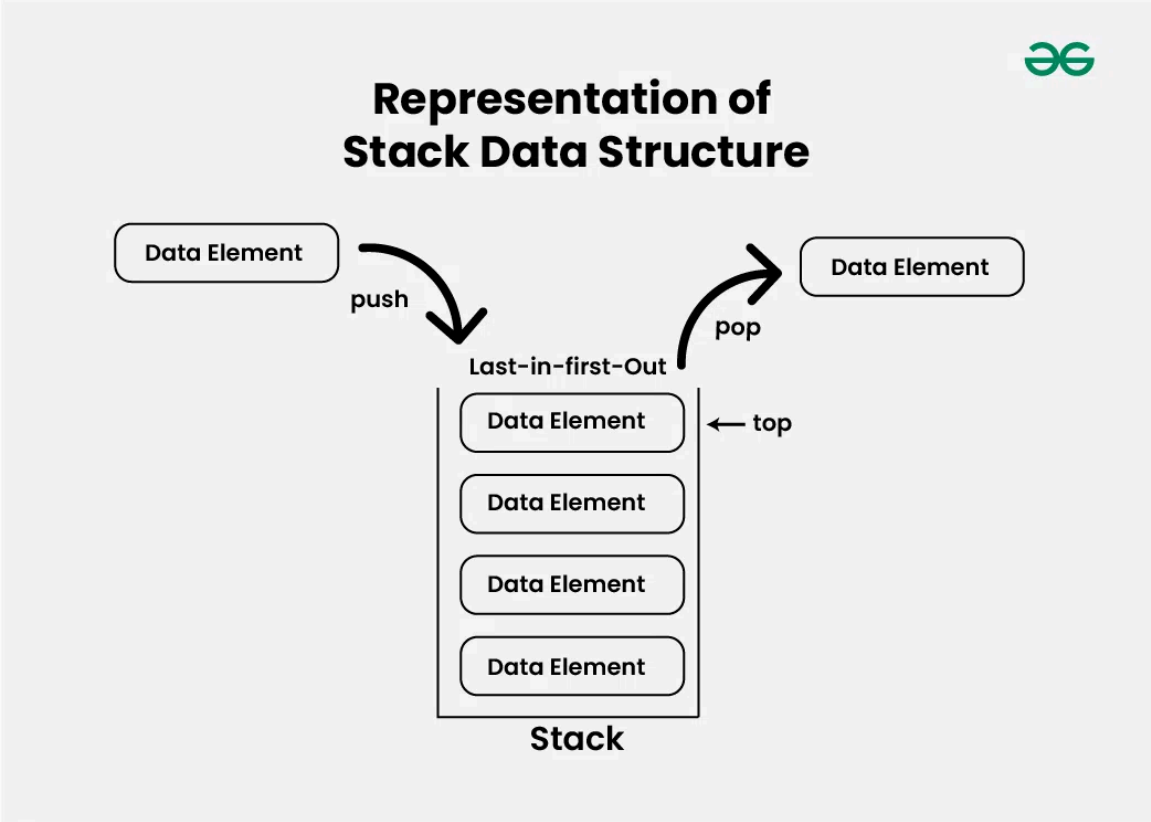
element to be inserted because **we can access the elements only on the top of the stack.**

LIFO(Last In First Out) Principle in Stack Data Structure:

This strategy states that the element that is inserted last will come out first. You can take a pile of plates kept on top of each other as a real-life example. The plate which we put last is on the top and since we remove the plate that is at the top, we can say that the plate that was put last comes out first.

Representation of Stack Data Structure:

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



Types of Stack Data Structure:

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

Basic Operations on Stack Data Structure:

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

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

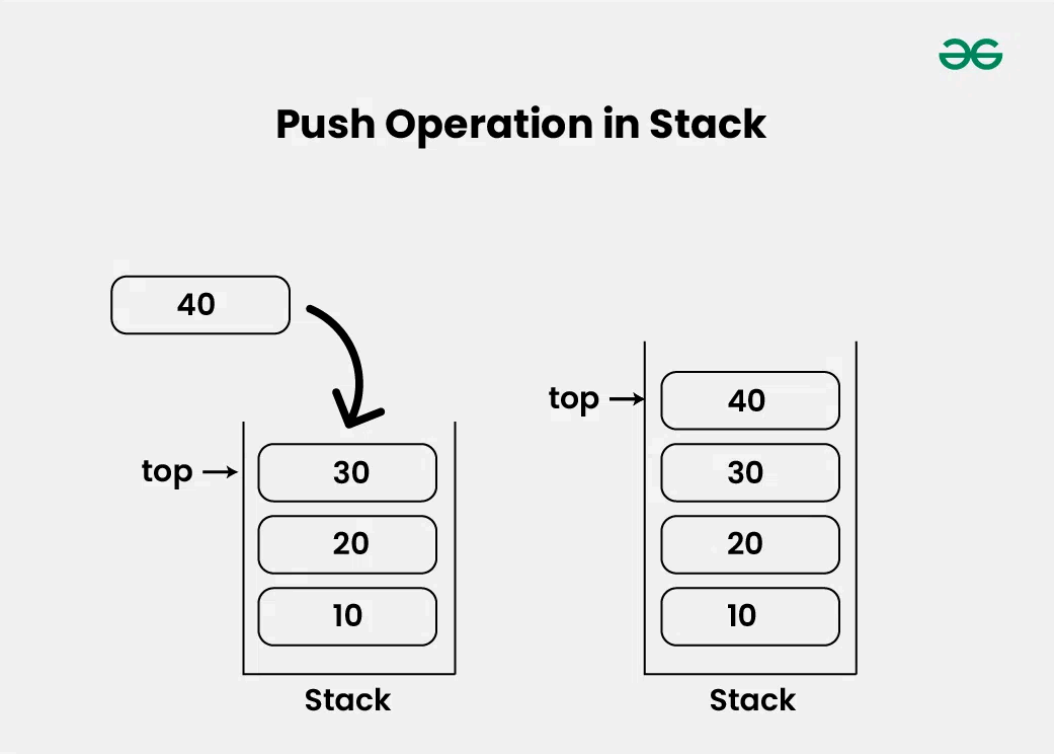
Push Operation in Stack Data Structure:

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

Algorithm for Push Operation:

- Before pushing the element to the stack, we check if the stack is **full** .

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



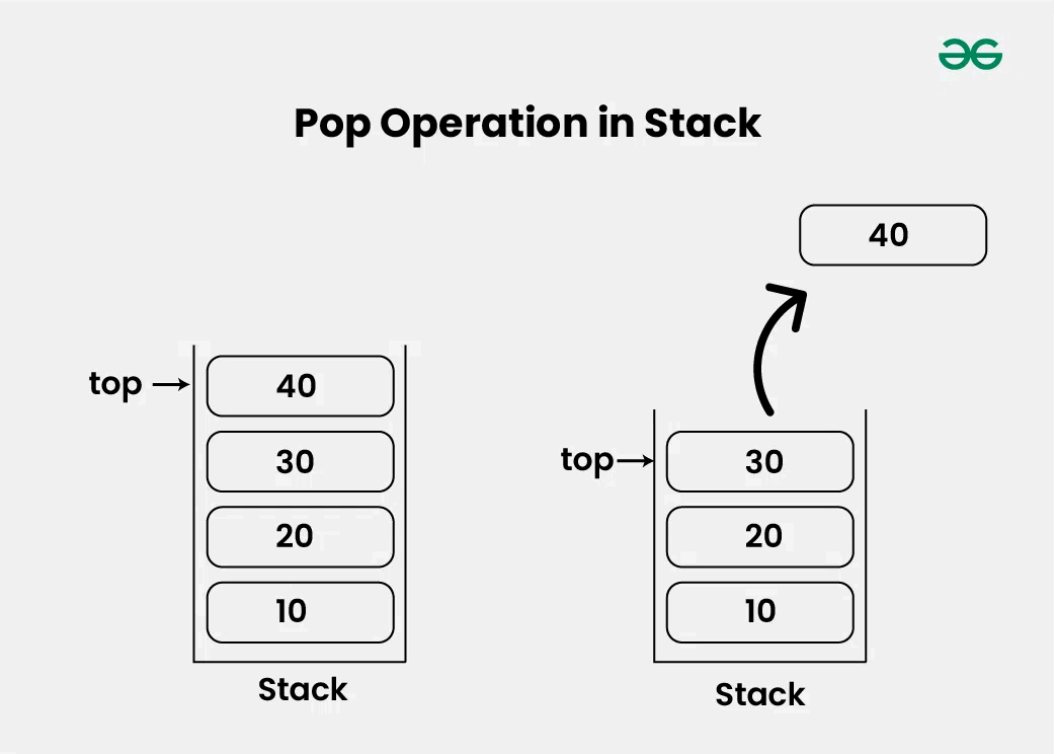
Pop Operation in Stack Data Structure:

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

Algorithm for Pop Operation:

- Before popping the element from the stack, we check if the stack is **empty** .
- If the stack is empty (`top == -1`), then **Stack Underflows** and we cannot remove any element from the stack.

- Otherwise, we store the value at top, decrement the value of top by 1 **(top = top – 1)** and return the stored top value.

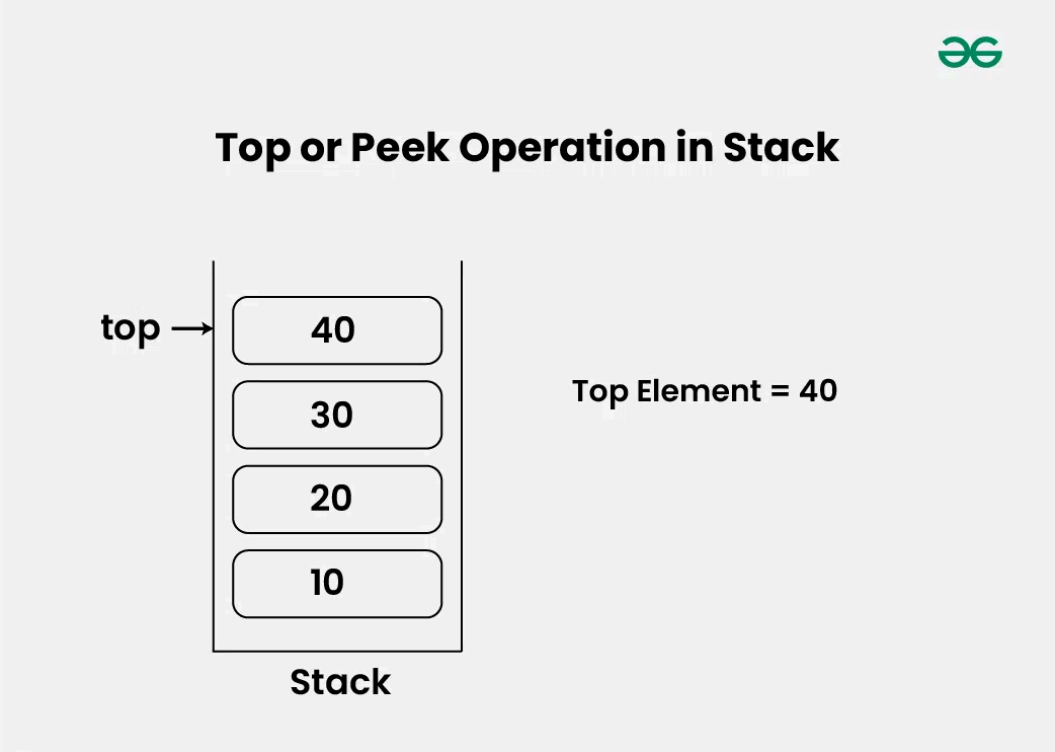


Top or Peek Operation in Stack Data Structure:

Returns the top element of the stack.

Algorithm for Top Operation:

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

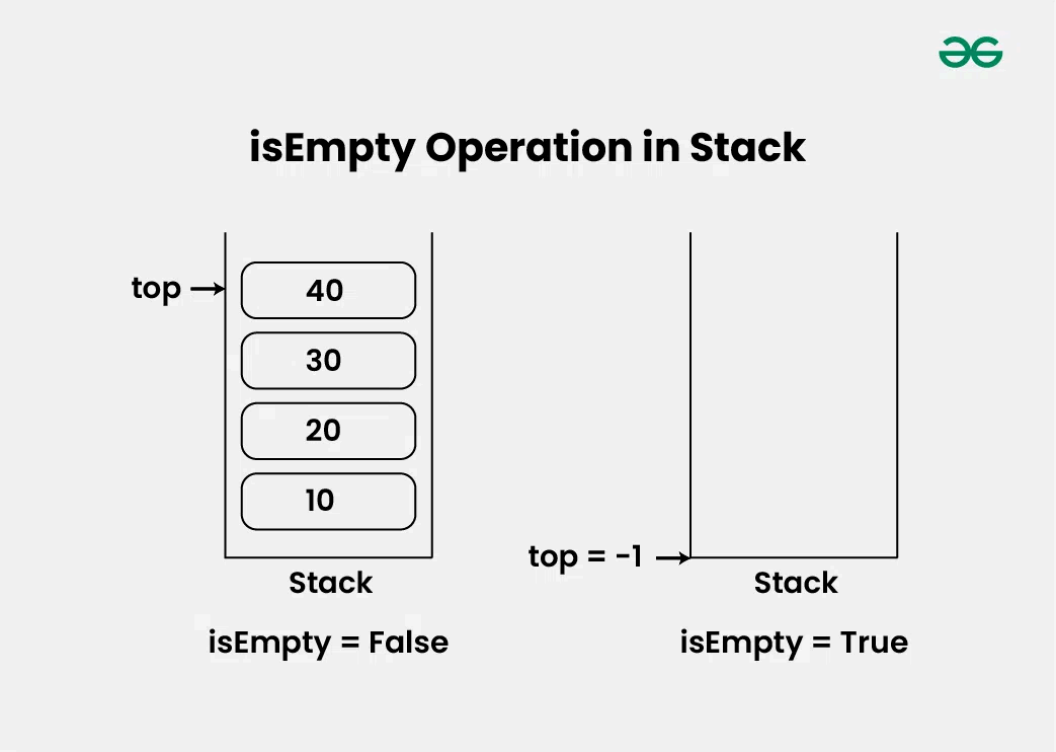


isEmpty Operation in Stack Data Structure:

Returns true if the stack is empty, else false.

Algorithm for isEmpty Operation:

- 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** .

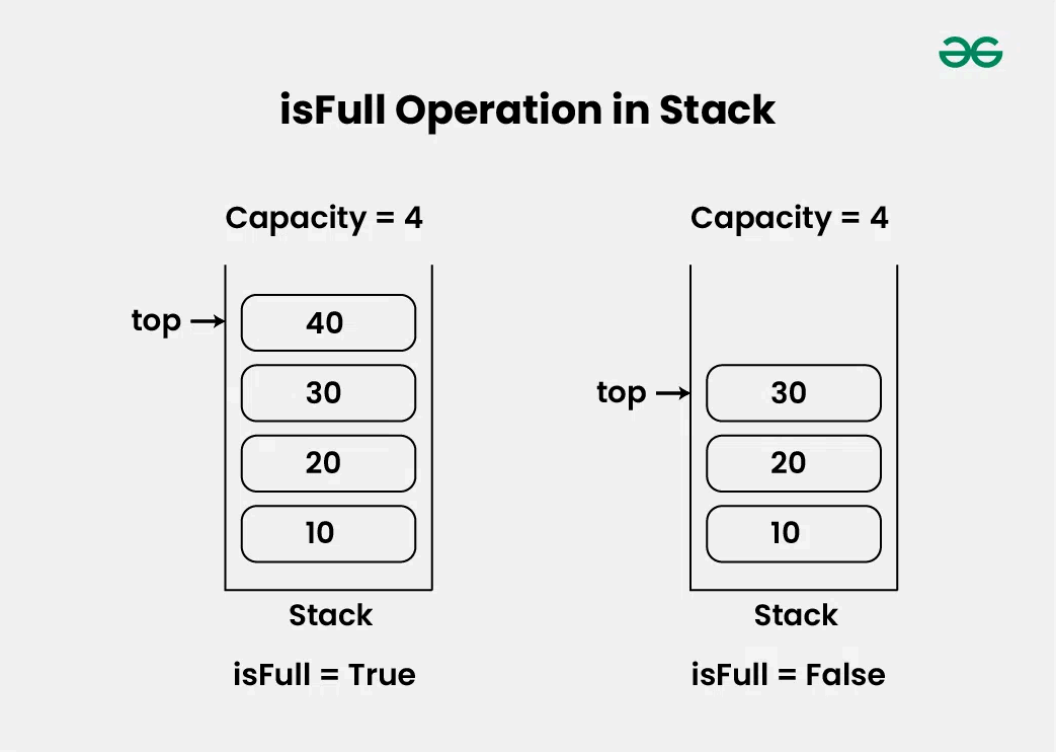


isFull Operation in Stack Data Structure:

Returns true if the stack is full, else false.

Algorithm for isFull Operation:

- 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**.



Implementation of Stack Data Structure:

The basic operations that can be performed on a stack include push, pop, and peek. There are two ways to implement a stack –

- Using Array
- Using Linked List

Stack Implementation Using Array

A stack is a data structure that follows the Last In First Out (LIFO) principle. In an array-based implementation, a fixed-size array is used to store the stack elements, and an index is maintained to track the top of the stack.

Array-Based Stack Methods

1. **push(T data)**: Add an element to the top of the stack.
2. **pop()**: Remove and return the top element of the stack.

3. **peek()**: Return the top element without removing it.
4. **isEmpty()**: Check if the stack is empty.
5. **isFull()**: Check if the stack is full.

Array-Based implementation:

```
public class ArrayStack <T>{
    private T[] stack;
    private int top;
    private int capacity;

    // Constructor to initialize the stack
    @SuppressWarnings("unchecked") // Tells the compiler to ignore the wa
    public ArrayStack(int capacity){
        this.capacity = capacity;
        stack = (T[]) new Object[capacity];
        top = -1;
    }

    // Method to add an element to the top of the stack
    public void push(T data){
        if(isFull()){
            throw new StackOverflowError("Stack is full");
        }
        stack[++top] = data;
    }

    // Method to remove and return the top element of the stack
    public T pop(){
        if(isEmpty())
            throw new IllegalStateException("Stack is empty");
        return stack[top--];
    }

    // Method to return the top element without removing it
```

```
    public T peek(){
        if(isEmpty())
            throw new IllegalStateException("Stack is empty");
        return stack[top];
    }

    // Method to check if the stack is full
    public boolean isFull(){
        return(top == capacity - 1);
    }

    // Method to check if the stack is empty
    public boolean isEmpty(){
        return (top == -1);
    }

    // Method to display the stack elements
    public void display(){
        if(isEmpty()){
            System.out.println("Stack is empty");
            return;
        }
        for(int i = 0; i <= top;i++){
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
```

Run class:

```
// Run class
public class Main {
    public static void main(String[] args) {
        ArrayStack<Integer> stack = new ArrayStack<>(5);
```

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

        stack.display(); // Output: 10 20 30

        System.out.println(stack.pop()); // Output: 30
        stack.display(); // Output: 10 20

        System.out.println(stack.peek()); // Output: 20
    }
}
```

Stack Implementation Using Linked List

In a linked list-based implementation, nodes are dynamically allocated to store the stack elements. This approach avoids the fixed size limitation of the array-based stack.

Linked List-Based Stack Methods

1. **push(T data)**: Add an element to the top of the stack.
2. **pop()**: Remove and return the top element of the stack.
3. **peek()**: Return the top element without removing it.
4. **isEmpty()**: Check if the stack is empty.

Linked List-Based Stack Implementation:

```
// Node class for Linked List
class Node<T> {
    T data;
    Node<T> next;
```

```
    Node(T data) {
        this.data = data;
        this.next = null;
    }
}
public class LinkedListStack<T> {
    private Node<T> top;

    // Constructor to initialize the stack
    public LinkedListStack() {
        top = null;
    }

    // Method to add an element to the top of the stack
    public void push(T data) {
        Node<T> newNode = new Node<>(data);
        newNode.next = top;
        top = newNode;
    }

    // Method to remove and return the top element of the stack
    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        T data = top.data;
        top = top.next;
        return data;
    }

    // Method to return the top element without removing it
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("Stack is empty");
        }
        return top.data;
```

```
    }

    // Method to check if the stack is empty
    public boolean isEmpty() {
        return top == null;
    }

    // Method to display the stack elements
    public void display() {
        if (isEmpty()) {
            System.out.println("Stack is empty");
            return;
        }
        Node<T> current = top;
        while (current != null) {
            System.out.print(current.data + " ");
            current = current.next;
        }
        System.out.println();
    }

    public static void main(String[] args) {
        LinkedListStack<Integer> stack = new LinkedListStack<>();

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

        stack.display(); // Output: 30 20 10

        System.out.println(stack.pop()); // Output: 30
        stack.display(); // Output: 20 10

        System.out.println(stack.peek()); // Output: 20
```

```
    }
}
```

Summary

Array-Based Stack

- Uses a fixed-size array.
- Simple and fast operations with constant time complexity.
- Limited by the initial capacity of the array.

Linked List-Based Stack

- Uses a dynamic linked list.
- No size limitation as nodes are dynamically allocated.
- More memory overhead due to storing references/pointers.

Both implementations provide the basic stack operations (`push`, `pop`, `peek`, `isEmpty`) and additional functionalities (`display` method) to visualize the stack contents. The choice between these two implementations depends on the specific requirements and constraints of your application.

Complexity Analysis of Operations on Stack Data Structure:

Operations	Time Complexity	Space Complexity
push()	O(1)	O(1)
pop()	O(1)	O(1)
top() or peek()	O(1)	O(1)
isEmpty()	O(1)	O(1)
isFull()	O(1)	O(1)

Advantages of Stack Data Structure:

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

Disadvantages of Stack Data Structure:

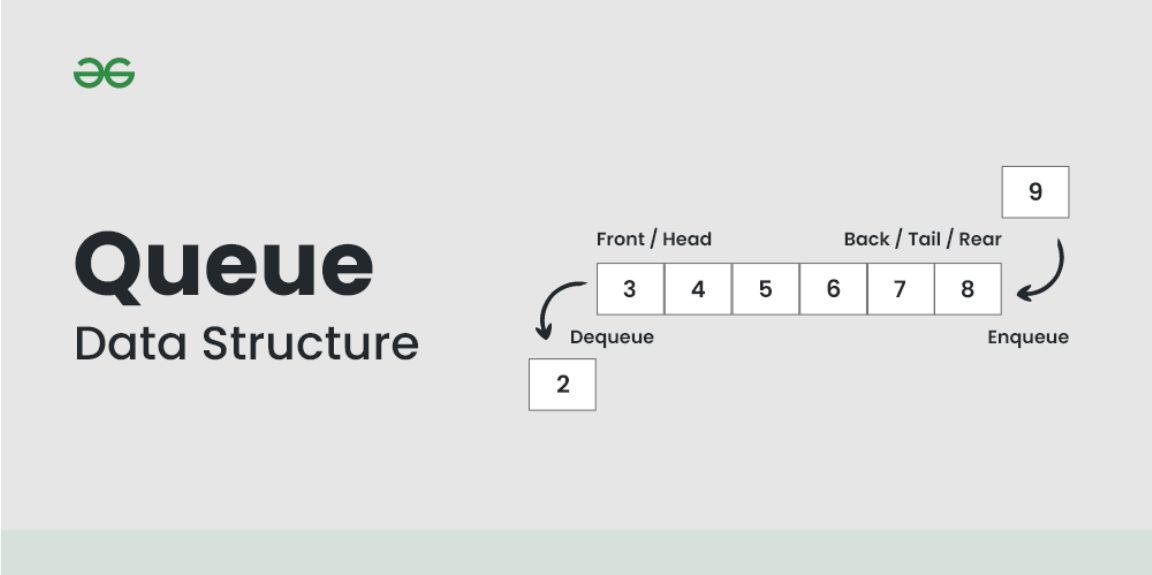
- **Limited access:** Elements in a stack can only be accessed from the top, making it difficult to retrieve or modify elements in the middle of the stack.
- **Potential for overflow:** If more elements are pushed onto a stack than it can hold, an overflow error will occur, resulting in a loss of data.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

Applications of Stack Data Structure:

- **Infix to Postfix** /Prefix conversion
- Redo-undo features at many places like editors, photoshop.
- Forward and backward features in web browsers
- In Memory management, any modern computer uses a stack as the primary management for a running purpose. Each program that is running in a computer system has its own memory allocations.
- Stack also helps in implementing function call in computers. The last called function is always completed first.

▼ Queue Data Structure

A **Queue Data Structure** is a fundamental concept in computer science used to store and manage data in a specific order. It follows the **"First in, First out" (FIFO) principle**, where the first element added to the queue is the first one to be removed. Queues are commonly used in various algorithms and applications for their simplicity and efficiency in managing data flow.

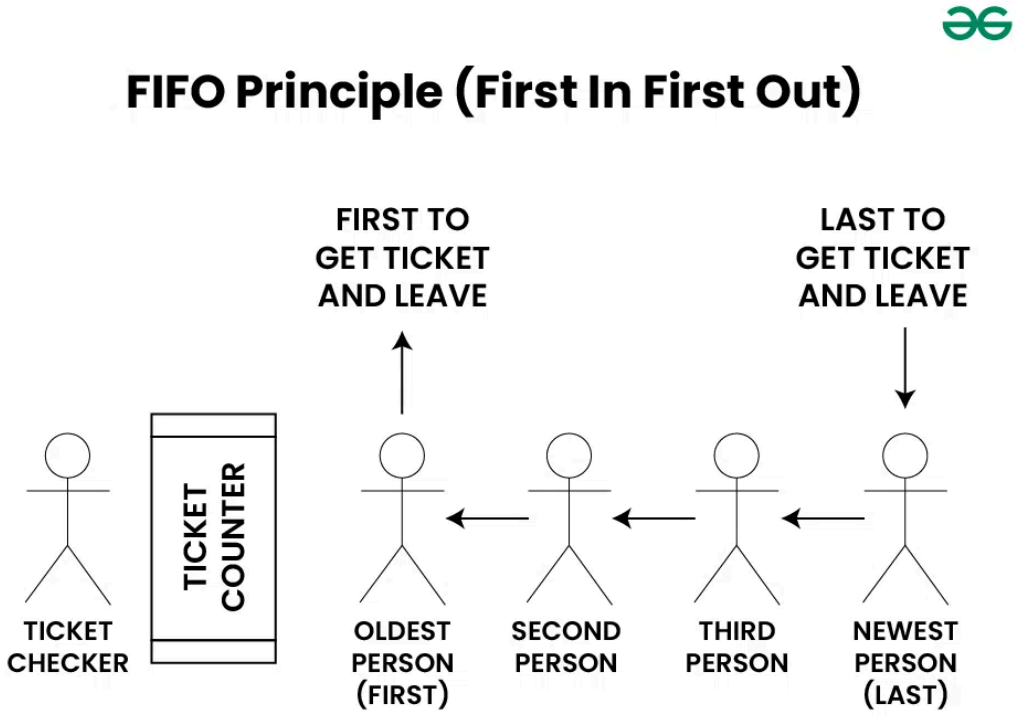


Queue Data Structure

We define a queue to be a list in which all additions to the list are made at one end (**back of the queue**), and all deletions from the list are made at the other end(**front of the queue**). The element which is first pushed into the order, the delete operation is first performed on that.

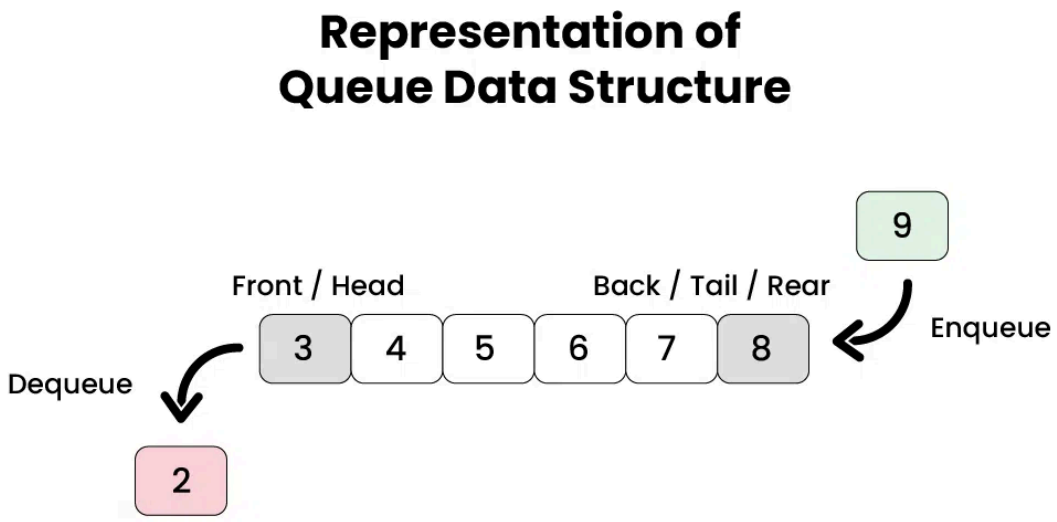
FIFO Principle of queue data structure:

- A Queue is like a line waiting to purchase tickets, where the first person in line is the first person served. (i.e. First Come First Serve).
- The 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(sometimes, the **head** of the queue). Similarly, the position of the last entry in the queue, that is, the one most recently added, is called the **rear** (or the **tail**) of the queue.



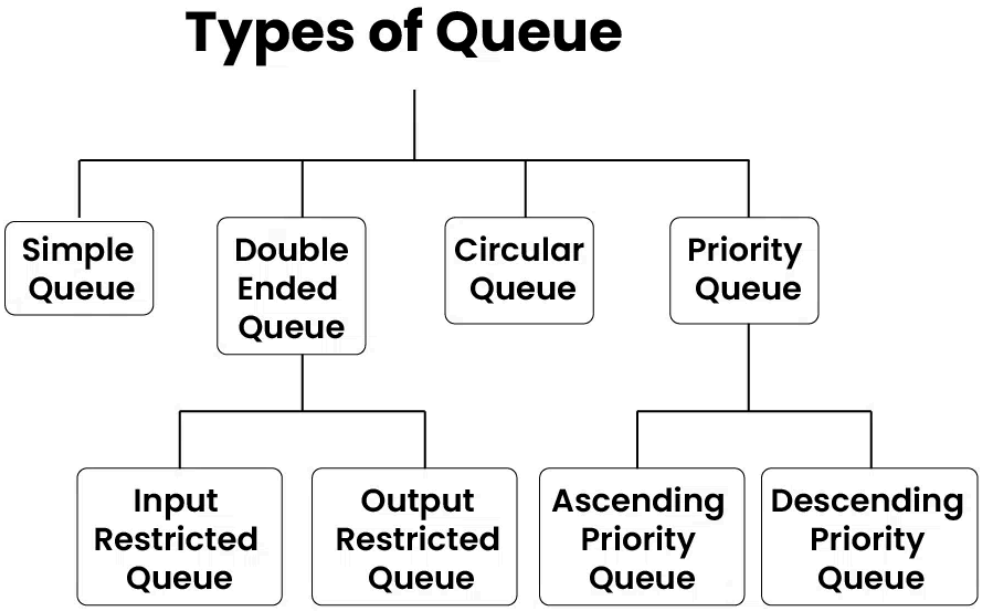
Representation of Queue Data Structure:

The image below shows how we represent Queue Data Structure:



Types of Queue Data Structure:

Queue data structure can be classified into 4 types:x`



There are different types of queues:

1. **Simple Queue:** Simple Queue simply follows **FIFO** Structure. We can only insert the element at the back and remove the element from the front of the queue.
2. **Double-Ended Queue (Dequeue):** In a double-ended queue the insertion and deletion operations, both can be performed from both ends. They are of two types:
   - **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.

3. **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.
4. **Priority Queue:** A priority queue is a special queue where the elements are accessed based on the priority assigned to them. They are of two types:
    - **Ascending Priority Queue:** In Ascending Priority Queue, the elements are arranged in increasing order of their priority values. Element with smallest priority value is popped first.
    - **Descending Priority Queue:** In Descending Priority Queue, the elements are arranged in decreasing order of their priority values. Element with largest priority is popper first.

▼ Simple Queue

Basic Operations in Queue Data Structure:

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

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

There are a few supporting operations (auxiliary operations):

1. Enqueue Operation in Queue Data Structure:

Enqueue() operation in Queue **adds (or stores) an element to the end of the queue**.

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

- **Step 1:** Check if the queue is full.
- **Step 2:** If the queue is full, return overflow error and exit.
- **Step 3:** If the queue is not full, increment the rear pointer to point to the next empty space.
- **Step 4:** Add the data element to the queue location, where the rear is pointing.
- **Step 5:** return success.

2. Dequeue Operation in Queue Data Structure:

Removes (or access) the first element from the queue.

The following steps are taken to perform the dequeue operation:

- **Step 1:** Check if the queue is empty.
- **Step 2:** If the queue is empty, return the underflow error and exit.
- **Step 3:** If the queue is not empty, access the data where the front is pointing.
- **Step 4:** Increment the front pointer to point to the next available data element.

- **Step 5:** The Return Success.

3. Front Operation in Queue Data Structure:

This operation returns the element at the front end without removing it.

4. Rear Operation in Queue Data Structure:

This operation returns the element at the rear end without removing it.

5. isEmpty Operation in Queue Data Structure:

This operation returns a boolean value that indicates whether the queue is empty or not.

6. isFull Operation in Queue Structure:

This operation returns a boolean value that indicates whether the queue is full or not.

Implementation of Queue Data Structure:

Queue can be implemented using following data structures:

1. Arrays
2. Linked List

implementation using arrays:

```
public class ArrayQueue<T> {
    private T[] array;
    private int front;
    private int rear;
    private int size;
    private static final int INITIAL_CAPACITY = 10;

    @SuppressWarnings("unchecked")
    public ArrayQueue() {
        array = (T[]) new Object[INITIAL_CAPACITY];
        front = 0;
        rear = -1;
        size = 0;
    }

    // Enqueue: Add an element to the end of the queue
    public void enqueue(T element) {
        if (size == array.length) {
            resize();
        }
        rear = (rear + 1) % array.length;
        array[rear] = element;
        size++;
    }

    // Dequeue: Remove and return the element from the front of the que
    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("Queue is empty");
        }
        T element = array[front];
        front = (front + 1) % array.length;
        size--;
        return element;
    }
```

```
// Peek: Return the element from the front of the queue without remo
public T peek() {
    if (isEmpty()) {
        throw new IllegalStateException("Queue is empty");
    }
    return array[front];
}

// IsEmpty: Check if the queue is empty
public boolean isEmpty() {
    return size == 0;
}

// Size: Get the number of elements in the queue
public int size() {
    return size;
}

// Resize the array when it is full
@SuppressWarnings("unchecked")
private void resize() {
    T[] newArray = (T[]) new Object[array.length * 2];
    for (int i = 0; i < size; i++) {
        newArray[i] = array[(front + i) % array.length];
    }
    array = newArray;
    front = 0;
    rear = size - 1;
}
// Main method for testing
public static void main(String[] args) {
    ArrayQueue<Integer> queue = new ArrayQueue<>();

    // Enqueue elements
    queue.enqueue(1);
```

```
        queue.enqueue(2);
        queue.enqueue(3);

        // Peek at the front element
        System.out.println("Peek: " + queue.peek()); // Output: Peek: 1

        // Dequeue elements
        System.out.println("Dequeue: " + queue.dequeue()); // Output: De
        System.out.println("Dequeue: " + queue.dequeue()); // Output: De

        // Check if the queue is empty
        System.out.println("Is Empty: " + queue.isEmpty()); // Output: Is Em

        // Get the size of the queue
        System.out.println("Size: " + queue.size()); // Output: Size: 1

        // Enqueue more elements to test resizing
        for (int i = 4; i <= 15; i++) {
            queue.enqueue(i);
        }

        // Print the size after resizing
        System.out.println("Size after resizing: " + queue.size()); // Output
    }
}
```

Implementation using Linked List:

```
public class Queue <T>{
    private LinkedList<T> list = new LinkedList<>();

    // Enqueue: Add an element to the end of the queue
    public void enqueue(T element){
        list.addLast(element);
    }
```

```
    // Dequeue: Remove and return the element from the front of the que
    public T dequeue(){
        if(isEmpty()){
            throw new IllegalStateException("Queue is empty");
        }
        return list.removeLast();
    }

    // IsEmpty: Check if the queue is empty
    public boolean isEmpty(){
        return list.isEmpty();
    }

    // Peek: Return the element from the front of the queue without remo
    public T peek(){
        if(isEmpty())
            throw new IllegalStateException("Queue is empty");
        return list.getFirst();
    }

    // Size: Get the number of elements in the queue
    public int size() {
        return list.size();
    }

}
// Main method for testing
    public static void main(String[] args) {
        Queue<Integer> queue = new Queue<>();

        // Enqueue elements
        queue.enqueue(1);
        queue.enqueue(2);
        queue.enqueue(3);
```

```
        // Peek at the front element
        System.out.println("Peek: " + queue.peek()); // Output: Peek: 1

        // Dequeue elements
        System.out.println("Dequeue: " + queue.dequeue()); // Output: De
        System.out.println("Dequeue: " + queue.dequeue()); // Output: De

        // Check if the queue is empty
        System.out.println("Is Empty: " + queue.isEmpty()); // Output: Is Em

        // Get the size of the queue
        System.out.println("Size: " + queue.size()); // Output: Size: 1
    }
}
```

Complexity Analysis of Operations on Queue Data Structure:

Operations	Time Complexity	Space Complexity
Enqueue	O(1)	O(1)
Dequeue	O(1)	O(1)
Front	O(1)	O(1)
Back	O(1)	O(1)
isEmpty	O(1)	O(1)
isFull	O(1)	O(1)

Applications of Queue Data Structure:

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

create other tasks, which must be done in turn by keeping them in a queue.

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

Advantages of Queue Data Structure:

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

Disadvantages of Queue Data Structure:

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

▼ Double-Ended Queue

Operations on Deque: Below is a table showing some basic operations along with their time complexity, performed on deques.

Operation	Description	Time Complexity
push_front()	Inserts the element at the beginning.	**O(1)**
push_back()	Adds element at the end.	**O(1)**
pop_front()	Removes the first element from the deque.	**O(1)**
pop_back()	Removes the last element from the deque.	**O(1)**
front()	Gets the front element from the deque.	**O(1)**
back()	Gets the last element from the deque.	**O(1)**
empty()	Checks whether the deque is empty or not.	**O(1)**
size()	Determines the number of elements in the deque.	**O(1)**

Other operations performed on deques are explained as follows:

clear(): Remove all the elements from the deque. It leaves the deque with a size of 0.

erase(): Remove one or more elements from the deque. It takes an iterator specifying the position of the first element to be removed, and an optional second iterator specifying the position of the last element to be removed.

swap(): Swap the contents of one deque with another deque.

`emplace_front()` : Insert a new element at the front of the deque. It is similar to the insert operation, but it avoids the copy constructor of the element being inserted.

`emplace_back()`: Insert a new element at the back of the deque. It is similar to the insert operation, but it avoids the copy constructor of the element being inserted.

`resize()` : Change the number of elements in the deque to a specific number. If the new size is larger than the current size, new elements are appended to the deque. If the new size is smaller than the current size, elements are removed from the deque.

`assign()`: Assign new values to the elements in the deque. It replaces the current contents of the deque with new elements.

`reverse()`: Reverse the order of the elements in the deque.

`sort()`: Sort the elements in the deque in ascending order. It uses the less-than operator to compare the elements.

Applications of Deque: Since Deque supports both stack and queue operations, it can be used as both. The Deque data structure supports clockwise and anticlockwise rotations in O(1) time which can be useful in certain applications. Also, the problems where elements need to be removed and or added to both ends can be efficiently solved using Deque. For example see the **Maximum of all subarrays of size k problem.**, **0-1 BFS,** and **Find the first circular tour that visits all petrol pumps**. See the **wiki page** for another example of the A-Steal job scheduling algorithm where Deque is used as deletions operation is required at both ends.

Some Practical Applications of Deque:

- Applied as both stack and queue, as it supports both operations.
- Storing a web browser’s history.
- Storing a software application’s list of undo operations.
- Job scheduling algorithm

Monotonic Deque :

- It is deque which stores elements in strictly increasing order or in strictly decreasing order
- To maintain monotonicity, we need to delete elements
  - For example – Consider monotonic(decreasing) deque **dq** = {5, 4, 2, 1}
  - Insert 3 into dq
    - So we need to delete elements till dq.back() < 3 to insert 3 into dq (*2,1 are the deleted elements)*
    - Resulting dq = {5, 4, 3}
- Applications of monotonic deque :
  - It can be used to get **next maximum** in a subarray (**sliding-window-maximum-of-all-subarrays-of-size-k**) by using monotonically decreasing deque
  - Like this it can be used to get **previous maximum** also in a subarray
  - It is frequently used in **sliding window problems (hard)**

Other Applications:

Deques have several other applications, some of which include:

- **Palindrome checking:** Deques can be used to check if a word or phrase is a palindrome. By inserting each character of the word or phrase into a deque, it is possible to check if the word or phrase is a palindrome by comparing the first and last characters, the second and second-to-last characters, and so on.
- **Graph traversal:** Deques can be used to implement Breadth-First Search (BFS) on a graph. BFS uses a queue to keep track of the vertices to be visited next, and a deque can be used as an alternative to a queue in this case.
- **Task scheduler:** Deques can be used to implement a task scheduler that keeps track of tasks to be executed. Tasks can be added to the back of the deque, and the scheduler can remove tasks from the front of the deque and execute them.
- **Multi-level undo/redo functionality:** Deques can be used to implement undo and redo functionality in applications. Each time a user acts, the current state of the application is pushed onto the deque. When the user undoes an action, the front of the deque is popped, and the previous state is restored. When the user redoes an action, the next state is popped from the deque.
- In computer science, deque can be used in many algorithms like *LRU Cache*, *Round Robin Scheduling*, and *Expression Evaluation*.

▼ Circular Queue

What is a Circular Queue?

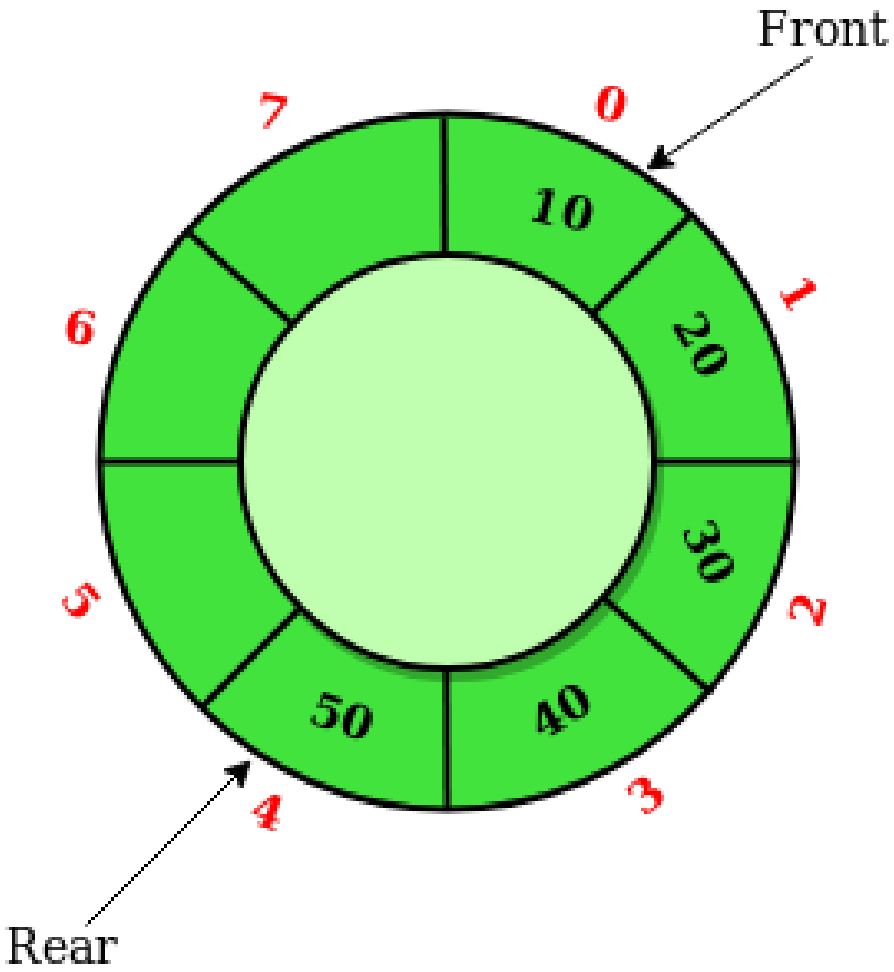
> A Circular Queue is an extended version of a normal queue where the last element of the

> queue is connected to the first element of the queue forming a circle.

The implemented normal queue above is circular because the very normal queue has limitations like:

- **Fixed Capacity**: The array has a fixed size, which limits the number of elements the queue can hold. If the queue reaches its capacity, no more elements can be enqueued unless the array is resized.
- **Wasted Space**: As elements are dequeued, the front of the queue moves forward in the array. Over time, this can leave unused space at the beginning of the array, which cannot be reused without shifting elements.
- **Inefficient Resizing**: When the array is full, resizing requires creating a new array and copying all elements from the old array to the new one. This operation is time-consuming and can lead to performance issues if the queue is resized frequently.
- **Element Shifting**: Without circular indexing, when elements are dequeued, if you want to maintain the queue's front at the beginning of the array, you need to shift all remaining elements forward. This operation is O(n) and can be very inefficient for large queues.
- **Fragmentation**: If elements are enqueued and dequeued frequently, the array can become fragmented, with the front and rear indices potentially far apart. This fragmentation can reduce the efficiency of the queue operations.

The operations in circular are performed based on the FIFO (First In First Out) principle. It is also called **‘Ring Buffer’**.



In a normal Queue, we can insert elements until queue becomes full. But once queue becomes full, we can not insert the next element even if there is a space in front of queue.

Operations on Circular Queue:

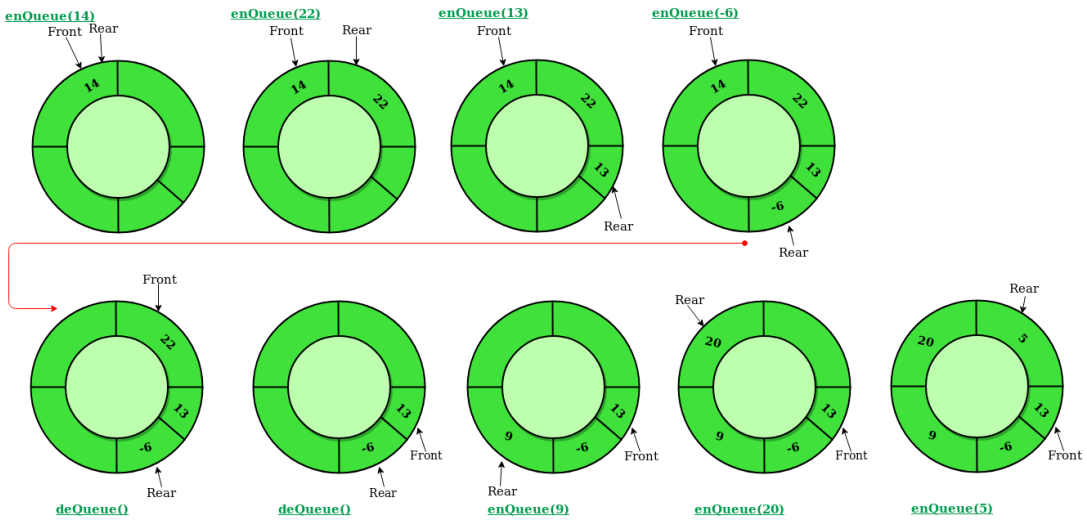
- **Front:** Get the front item from the queue.
- **Rear:** Get the last item from the queue.
- **enQueue(value)** This function is used to insert an element into the circular queue. In a circular queue, the new element is always inserted at the rear position.

- Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].
- If it is full then display Queue is full.
  - If the queue is not full then, insert an element at the end of the queue.

- **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.
  - Check whether the queue is Empty.
  - If it is empty then display Queue is empty.
    - If the queue is not empty, then get the last element and remove it from the queue.

Illustration of Circular Queue Operations:

Follow the below image for a better understanding of the enqueue and dequeue operations.



Working of Circular queue operations

▼ Priority Queue

What is a priority queue?

A priority queue is an abstract data type that behaves similarly to the normal queue except that each element has some priority, i.e., the element with the highest priority would come first in a priority queue. The priority of the elements in a priority queue will determine the order in which elements are removed from the priority queue.

The priority queue supports only comparable elements, which means that the elements are either arranged in ascending or descending order.

For example, suppose we have some values like 1, 3, 4, 8, 14, and 22 inserted in a priority queue with an ordering imposed on the values from least to greatest. Therefore, the 1 number would have the highest priority while 22 would have the lowest priority.

Characteristics of a Priority Queue

A priority queue is an extension of a queue that contains the following characteristics:

- Every element in a priority queue has some priority associated with it.
- An element with a higher priority will be deleted before the deletion of the lesser priority.
- If two elements in a priority queue have the same priority, they will be arranged using the FIFO principle.

Let's understand the priority queue through an example.

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the values are arranged in ascending order. Now, we will observe how the priority queue will look after performing the following operations:

- **poll():** This function will remove the highest priority element from the priority queue. In the above priority queue, the '1' element has the highest priority, so it will be removed from the priority queue.
- **add(2):** This function will insert a '2' element in a priority queue. As 2 is the smallest element among all the numbers it will obtain the highest priority.
- **poll():** It will remove '2' element from the priority queue as it has the highest priority queue.
- **add(5):** It will insert 5 element after 4 as 5 is larger than 4 and lesser than 8, so it will obtain the third highest priority in a priority queue.

Types of Priority Queue

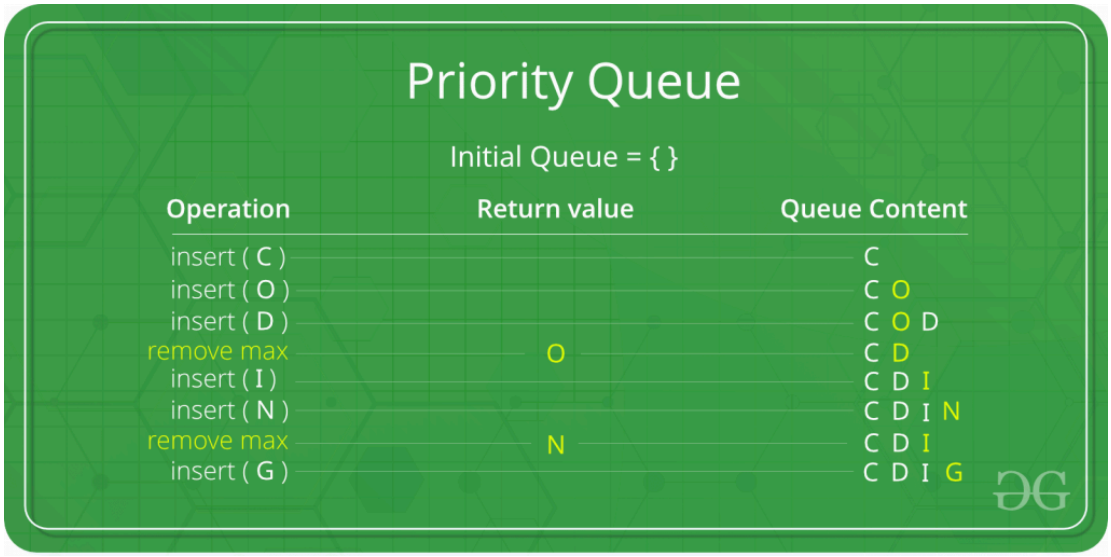
There are two types of priority queues:

- **Ascending order priority queue:** In ascending order priority queue, a lower priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in an ascending order like 1,2,3,4,5; therefore, the smallest number, i.e., 1 is given as the highest priority in a priority queue.

- **Descending order priority queue:** In descending order priority queue, a higher priority number is given as a higher priority in a priority. For example, we take the numbers from 1 to 5 arranged in descending order like 5, 4, 3, 2, 1; therefore, the largest number,

i.e., 5 is given as the highest priority in a priority queue.

In the below priority queue, an element with a maximum ASCII value will have the highest priority. The elements with higher priority are served first.



Representation of priority queue

Now, we will see how to represent the priority queue through a one-way list.

We will create the priority queue by using the list given below in which **INFO** list contains the data elements, **PRN** list contains the priority numbers of each data element available in the **INFO** list, and LINK basically contains the address of the next node.

Let's create the priority queue step by step.

In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority

number = higher priority.

Step 1: In the list, lower priority number is 1, whose data value is 300, so it will be inserted in the list as shown in the below diagram:

Step 2: After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

Step 3: After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

Step 4: After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.

Implementation of Priority Queue

The priority queue can be implemented in four ways that include arrays, linked list, heap data structure and binary search tree. The heap data structure is the most efficient way of implementing the priority queue, so we will implement the priority queue using a heap data structure in this topic. Now, first we understand the reason why heap is the most efficient way among all the other data structures.

Analysis of complexities using different implementations

Implementation	add	Remove	peek

Linked list	O(1)	O(n)	O(n)
Binary heap	O(logn)	O(logn)	O(1)
Binary search tree	O(logn)	O(logn)	O(1)

What is Heap?

A heap is a tree-based data structure that forms a complete binary tree, and satisfies the heap property. If A is a parent node of B, then A is ordered with respect to the node B for all nodes A and B in a heap. It means that the value of the parent node could be more than or equal to the value of the child node, or the value of the parent node could be less than or equal to the value of the child node. Therefore, we can say that there are two types of heaps:

- **Max heap:** The max heap is a heap in which the value of the parent node is greater than the value of the child nodes.

- **Min heap:** The min heap is a heap in which the value of the parent node is less than the value of the child nodes.

Both the heaps are the binary heap, as each has exactly two child nodes.

Priority Queue Operations

The common operations that we can perform on a priority queue are insertion, deletion and peek. Let's see how we can maintain the heap data structure.

- **Inserting the element in a priority queue (max heap)**

If we insert an element in a priority queue, it will move to the empty slot by looking from top to bottom and left to right.

If the element is not in a correct place then it is compared with the parent node; if it is found out of order, elements are swapped. This process continues until the element is placed in a correct position.

- **Removing the minimum element from the priority queue**

As we know that in a max heap, the maximum element is the root node. When we remove the root node, it creates an empty slot. The last inserted element will be added in this empty slot. Then, this element is compared with the child nodes, i.e., left-child and right child, and swap with the smaller of the two. It keeps moving down the tree until the heap property is restored.

Applications of Priority queue

The following are the applications of the priority queue:

- It is used in the Dijkstra's shortest path algorithm.
- It is used in prim's algorithm
- It is used in data compression techniques like Huffman code.
- It is used in heap sort.
- It is also used in operating system like priority scheduling, load balancing and interrupt handling.

FAQs (Frequently asked questions) on Queue Data Structure:

1. What data structure can be used to implement a priority queue?

> Priority queues can be implemented using a variety of data structures, including linked lists, arrays, binary search trees, and heaps. Priority queues are best implemented using the heap data structure.

3. In data structures, what is a double-ended queue?

> In a double-ended queue, elements can be inserted and removed at both ends.

4. What is better, a stack or a queue?

> If you want things to come out in the order you put them in, use a queue. Stacks are useful when you want to reorder things after putting them in.

5. Which principle is followed by Queue data structure?

> Queue follows FIFO (First-In-First-Out) principle, so the element which is inserted first will be deleted first.

6. What are the four types of Queue data structure?

> The four types of Queue are: Simple Queue, Double-ended queue, Circular Queue and Priority Queue.

7. What are some real-life applications of Queue data structure?

> Real-life applications of queue include: Cashier Line in Stores, CPU Scheduling, Disk Scheduling, Serving requests on a shared resource like Printer.

8. What are some limitations of Queue data structure?

> Some of the limitations of Queue are: deletion of some middle element is not possible until all the elements before it are not deleted. Also random access of any element takes linear time complexity.

9. What are the different ways to implement a Queue?

> Queue data structure can be implemented by using Arrays or by using Linked List. The Array implementation requires the size of the Queue to be specified at the time of declaration.

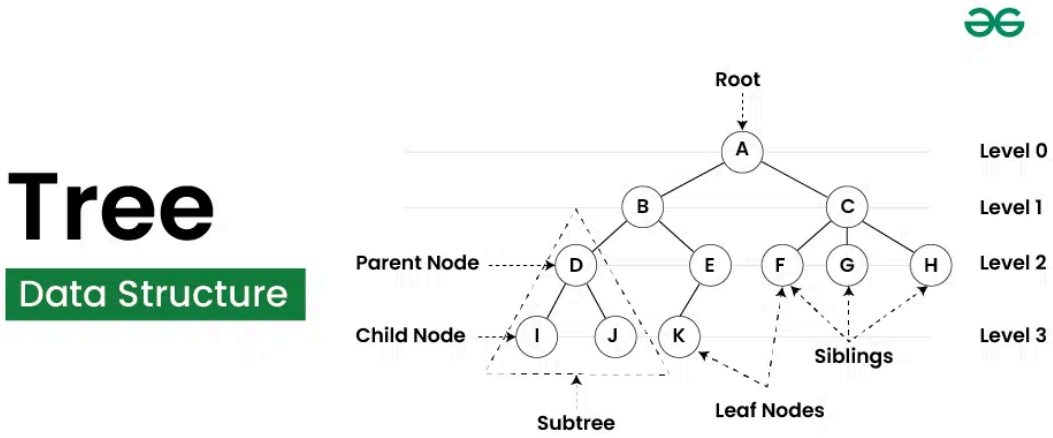
10. What are some common operations in Queue data structure?

> Some common operations on Queue are: Insertion or Enqueue, Deletion or Dequeue, Front, Rear, isFull, isEmpty, etc.

Non-Linear Data Structures

Tree Data Structure

Tree data structure is a specialized data structure to store data in hierarchical manner. It is used to organize and store data in the computer to be used more effectively. It consists of a central node, structural nodes, and sub-nodes, which are connected via edges. We can also say that tree data structure has roots, branches, and leaves connected.



Basic Terminologies In Tree Data Structure:

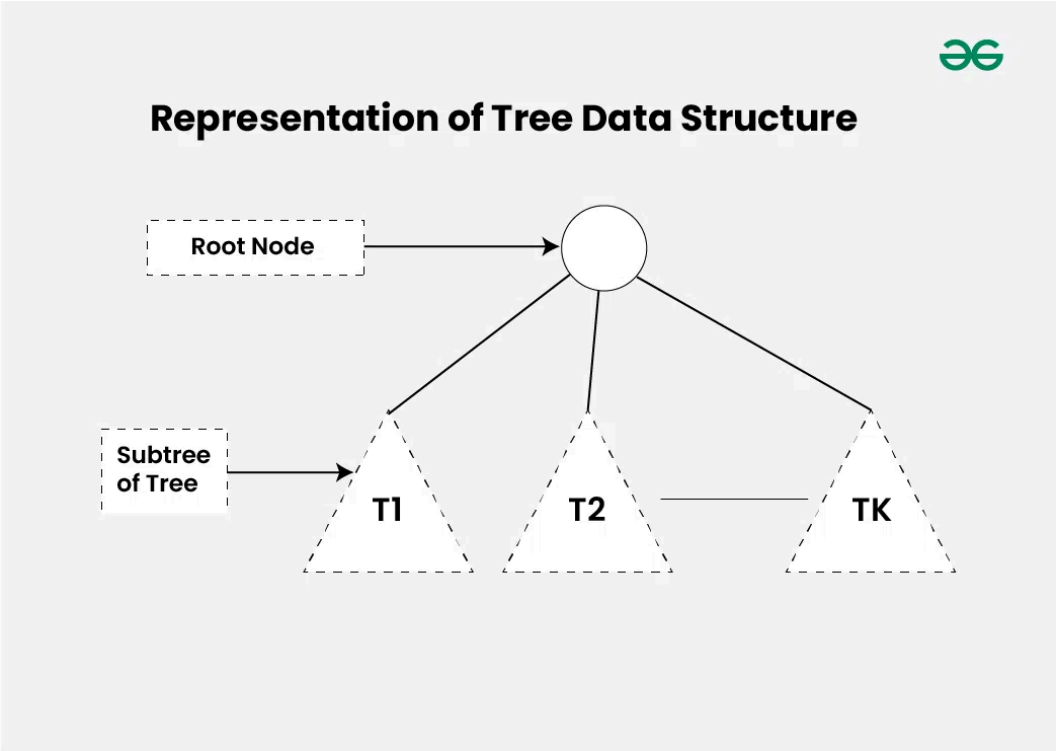
- **Parent Node:** The node which is a predecessor of a node is called the parent node of that node. **{B}** is the parent node of **{D, E}**.

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

Representation of Tree Data Structure:

A tree consists of a root node, and zero or more subtrees T1, T2, …, Tk such that there is an edge from the tree’s root node to the root node of each subtree. The subtree of

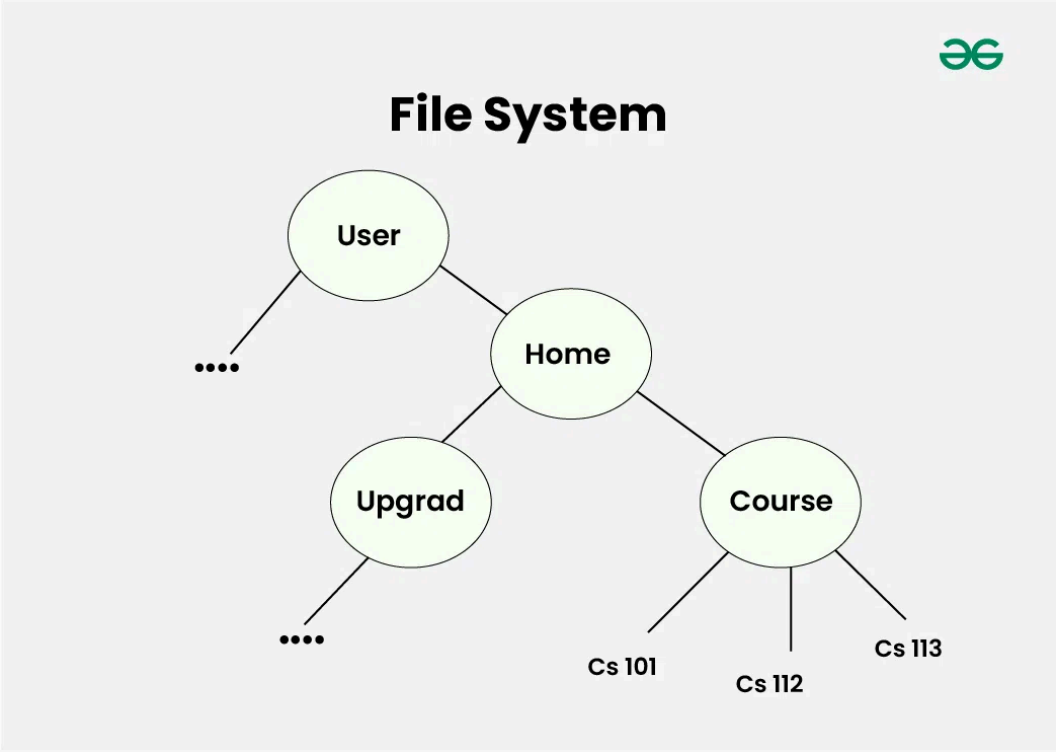
node X consists of all the nodes which have node X as the ancestor node.



Representation of Tree Data Structure

Importance for Tree Data Structure:

1. One reason to use trees might be because you want to store information that naturally forms a hierarchy. For example, the file system on a computer:

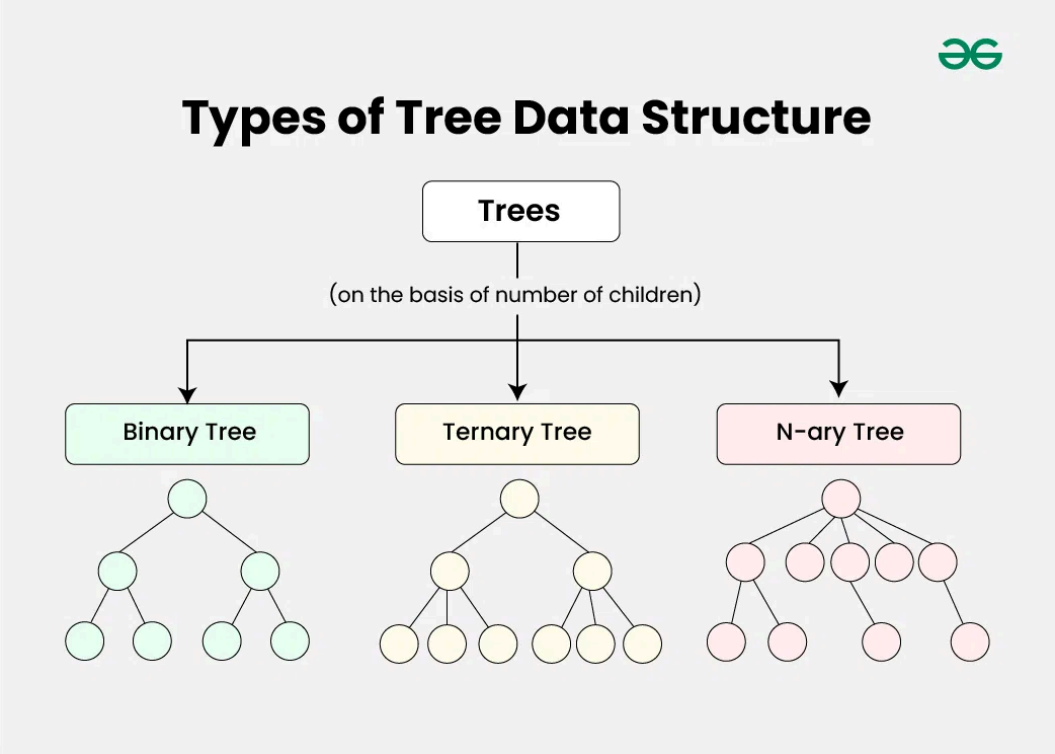


2. Trees (with some ordering e.g., BST) provide moderate access/search (quicker than Linked List and slower than arrays).

3. Trees provide moderate insertion/deletion (quicker than Arrays and slower than Unordered Linked Lists).

4. Like Linked Lists and unlike Arrays, Trees don’t have an upper limit on the number of nodes as nodes are linked using pointers.

Types of Tree data structures:



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

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

Most used Types of Trees:

- **Binary Tree**: A tree where each node has at most two children.
- **Binary Search Tree (BST)**: A binary tree with the property that the left child is less than the parent node, and the right child is greater.
- **Balanced Trees**: AVL trees, Red-Black trees, etc.
- **Other Trees**: N-ary trees, Trie, Segment Tree, Fenwick Tree, etc.

Basic Operations Of Tree Data Structure:

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

Properties of Tree Data Structure:

- **Number of edges:** An edge can be defined as the connection between two nodes. If a tree has N nodes then it will have (N-1) edges. There is only one path from each node to any other node of the tree.
- **Depth of a node:** The depth of a node is defined as the length of the path from the root to that node. Each edge adds 1 unit of length to the path. So, it can also be defined as the number of edges in the path from the root of the tree to the node.

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

Applications of Tree Data Structure:

- **File System:** This allows for efficient navigation and organization of files.
- **Data Compression**: **Huffman coding** is a popular technique for data compression that involves constructing a binary tree where the leaves represent characters and their frequency of occurrence. The resulting tree is used to encode the data in a way that minimizes the amount of storage required.
- **Compiler Design:** In compiler design, a syntax tree is used to represent the structure of a program.
- **Database Indexing**: B-trees and other tree structures are used in database indexing to efficiently search for and retrieve data.

Advantages of Tree Data Structure:

- Tree offer **Efficient Searching** Depending on the type of tree, with average search times of O(log n) for balanced trees like AVL.

- Trees provide a hierarchical representation of data, making it **easy to organize and navigate** large amounts of information.
- The recursive nature of trees makes them **easy to traverse and manipulate** using recursive algorithms.

Disadvantages of Tree Data Structure:

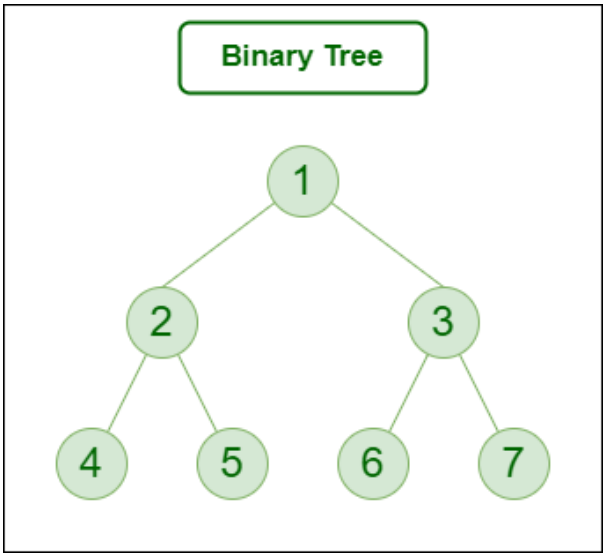
- Unbalanced Trees, meaning that the height of the tree is skewed towards one side, which can lead to **inefficient search times.**
- Trees demand **more memory space requirements** than some other data structures like arrays and linked lists, especially if the tree is very large.
- The implementation and **manipulation of trees can be complex** and require a good understanding of the algorithms.

▼ Binary Tree

> A binary Tree is defined as a Tree data structure with at most 2 children. Since each element in a binary tree can have only 2 children, we typically name them the left and right child.

Example:

Consider the tree below. Since each node of this tree has only 2 children, it can be said that this tree is a Binary Tree



Binary Tree

Types of Binary Tree:

- Binary Tree consists of the following types **based on the number of children**:
  1. **Full Binary Tree**
  2. Degenerate Binary Tree
- **Based on the completion of levels**, the binary tree can be divided into the following types:
  1. **Complete Binary Tree**
  2. **Perfect Binary Tree**
  3. **Balanced Binary Tree**

▼ Basics of Binary Search Trees (BSTs)

Let's break down the foundational concepts of Binary Search Trees (BSTs) step by step:

1. What is a Binary Search Tree?

A Binary Search Tree is a type of binary tree where each node has at most two children. The tree is organized in a

specific way that makes it efficient for searching, insertion, and deletion operations.

2. Key Properties of a BST

- **Binary Structure**: Each node can have at most two children: a left child and a right child.
- **Ordering Property**:
  - For any given node:
    - **Left Subtree**: All the values in the left subtree are less than the value of the node.
    - **Right Subtree**: All the values in the right subtree are greater than the value of the node.
- **No Duplicates**: Generally, BSTs do not allow duplicate values (though this can vary depending on the implementation).

3. Basic Operations on a BST

- **Insertion**:
  - Start at the root.
  - Compare the value to be inserted with the current node’s value.
  - If it’s smaller, move to the left child; if it’s larger, move to the right child.
  - Repeat this process until you find an empty spot where the new node can be inserted.
- **Search**:
  - Begin at the root.
  - Compare the target value with the current node’s value.

- Move left if the target is smaller, right if it’s larger.
- Repeat until you find the target value or reach a null pointer, indicating the value isn’t in the tree.

- **Deletion**:

  There Are 3 Cases of the node you want to delete:

  - **Case 1: No children** (leaf node): Simply remove the node.
  - **Case 2: One child**: Remove the node and replace it with its child.
  - **Case 3: Two children**: Find the node’s in-order predecessor or successor (the largest value in the left subtree or the smallest value in the right subtree), replace the node’s value with this value, and then delete the predecessor/successor node.

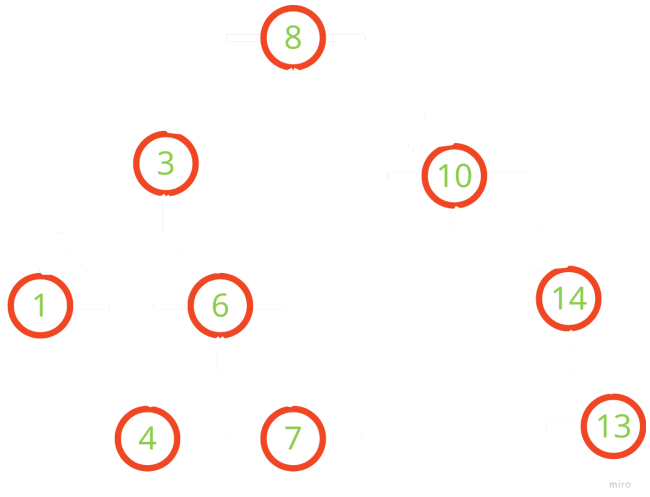
4. Tree Traversal Methods

- **In-order Traversal (Left, Root, Right)**:
  - This traversal method visits the nodes in ascending order. (Down arrow)
- **Pre-order Traversal (Root, Left, Right)**:
  - Visits the root node first, then the left subtree, and finally the right subtree. (Left arrow)
- **Post-order Traversal (Left, Right, Root)**:
  - Visits the left subtree, then the right subtree, and finally the root node. (Right arrow)

5. Example of a Binary Search Tree

Consider the following sequence of numbers: `8, 3, 10, 1, 6, 14, 4, 7, 13`.

If we insert them one by one into a BST, the structure would look like this:



- The root node is `8`.
- `3` is less than `8`, so it becomes the left child of `8`.
- `10` is greater than `8`, so it becomes the right child of `8`, and so on.

6. Why is BST Efficient?

- **Time Complexity**:
  - Search, insertion, and deletion all have an average-case time complexity of O(log n) because the tree's height is logarithmic relative to the number of nodes.
  - However, in the worst case (e.g., the tree degenerates into a linked list), the time complexity can degrade to O(n).
- **Balanced Trees**: Ensuring that the tree remains balanced (equal number of nodes on both sides) is crucial for maintaining efficiency. Advanced structures like AVL trees or Red-Black trees address this issue.

▼ Code Implementation - Generic BST in Java

1. Define the Node Class

- Each node in the BST will store a value and have references to its left and right children.

```
class Node<T extends Comparable<T>> {
    T value;
    Node<T> left, right;

    public Node(T value) {
        this.value = value;
        left = right = null;
    }
}
```

2. Define the BST Class

- The BST class will contain the root node and methods for insertion, searching, and traversal.

```
public class BinarySearchTree<T extends Comparable<T>> {
    private Node<T> root;

    // Constructor
    public BinarySearchTree() {
        root = null;
    }

    // Insertion method
    public void insert(T value) {
        root = insertRec(root, value);
    }
```

```
// Recursive insertion method
private Node<T> insertRec(Node<T> root, T value) {
    if (root == null) {
        root = new Node<>(value);
        return root;
    }

    if (value.compareTo(root.value) < 0) {
        root.left = insertRec(root.left, value);
    } else if (value.compareTo(root.value) > 0) {
        root.right = insertRec(root.right, value);
    }

    return root;
}

// Search method
public boolean search(T value) {
    return searchRec(root, value);
}

// Recursive search method
private boolean searchRec(Node<T> root, T value) {
    if (root == null) {
        return false;
    }

    if (value.compareTo(root.value) == 0) {
        return true;
    }

    return value.compareTo(root.value) < 0 ?
            searchRec(root.left, value) :
        searchRec(root.right, value);
}
```

```
// In-order traversal method
public void inorder() {
    inorderRec(root);
}

// Recursive in-order traversal method
private void inorderRec(Node<T> root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.value + " ");
        inorderRec(root.right);
    }
}

 // Pre-order traversal
public void preorder() {
    preorderRec(root);
}

private void preorderRec(Node<T> root) {
    if (root != null) {
        System.out.print(root.value + " ");
        preorderRec(root.left);
        preorderRec(root.right);
    }
}

// Post-order traversal
public void postorder() {
    postorderRec(root);
}

private void postorderRec(Node<T> root) {
    if (root != null) {
        postorderRec(root.left);
        postorderRec(root.right);
```

```
        System.out.print(root.value + " ");
    }
  }
}
```

3. Usage Example

- Here’s how you might use this generic BST class with different data types.

```
public class Main {
    public static void main(String[] args) {
        // Integer BST
        BinarySearchTree<Integer> intTree = new BinarySearchTree<>();
        intTree.insert(50);
        intTree.insert(30);
        intTree.insert(20);
        intTree.insert(40);
        intTree.insert(70);
        intTree.insert(60);
        intTree.insert(80);

        System.out.println("In-order traversal of Integer BST:");
        intTree.inorder();
        System.out.println();

        System.out.println("\nPre-order traversal:");
        intTree.preorder();
        System.out.println();

        System.out.println("\nPost-order traversal:");
        intTree.postorder();
        System.out.println();

        System.out.println("Search 40: " + intTree.search(40)); // true
        System.out.println("Search 100: " + intTree.search(100)); // false
```

```
        // String BST
        BinarySearchTree<String> stringTree = new BinarySearchTree<>();
        stringTree.insert("apple");
        stringTree.insert("banana");
        stringTree.insert("cherry");
        stringTree.insert("date");
        stringTree.insert("elderberry");

        System.out.println("\nIn-order traversal of String BST:");
        stringTree.inorder();
        System.out.println();

        System.out.println("\nPre-order traversal:");
        stringTree.preorder();
        System.out.println();

        System.out.println("\nPost-order traversal:");
        stringTree.postorder();
        System.out.println();

        System.out.println("Search 'cherry': " + stringTree.search("cherry"));
        System.out.println("Search 'fig': " + stringTree.search("fig")); // false
    }
}
```

Explanation:

- **Generics**: We've used generics (`<T extends Comparable<T>>`) to ensure that the BST can store any type of data that can be compared. This is why we can use the `compareTo` method to maintain the BST properties.
- **Insertion**: The `insert` method is a public method that calls a private recursive method `insertRec` to handle the actual insertion.

- **Search**: Similarly, the `search` method uses recursion to traverse the tree and find the target value.
- **Traversal**: The `inorder` method provides an in-order traversal, which will print the BST's values in ascending order.

Deletion Method:

The deletion process in a BST is more complex because it involves handling three different cases:

- **Case 1: The node to be deleted has no children (leaf node)**.
- **Case 2: The node to be deleted has one child**.
- **Case 3: The node to be deleted has two children**.

Here's how you can implement the deletion method

```
public class BinarySearchTree<T extends Comparable<T>> {
    private Node<T> root;

    // ... (existing methods: insert, search, etc.)

    // Deletion method
    public void delete(T value) {
        root = deleteRec(root, value);
    }

    // Recursive deletion method
    private Node<T> deleteRec(Node<T> root, T value) {
        if (root == null) {
            return root;
        }

        // Traverse the tree to find the node to be deleted
        if (value.compareTo(root.value) < 0) {
```

```
            root.left = deleteRec(root.left, value);
        } else if (value.compareTo(root.value) > 0) {
            root.right = deleteRec(root.right, value);
        } else {
            // Node with only one child or no child
            if (root.left == null) {
                return root.right;
            } else if (root.right == null) {
                return root.left;
            }

            // Node with two children: Get the in-order successor (smallest in th
            root.value = minValue(root.right);

            // Delete the in-order successor
            root.right = deleteRec(root.right, root.value);
        }

        return root;
    }

    // Helper method to find the minimum value in the right subtree
    private T minValue(Node<T> root) {
        T minValue = root.value;
        while (root.left != null) {
            minValue = root.left.value;
            root = root.left;
        }
        return minValue;
    }

    // ... (other existing methods)
}
```

Certainly! Removing a node with two children in a Binary Search Tree (BST) is the most complex case among the

deletion scenarios. The key idea is to replace the node to be deleted with a value that preserves the BST property, ensuring that all nodes in the left subtree are still smaller and all nodes in the right subtree are still larger after the deletion.

Detailed Explanation: Deleting a Node with Two Children

1. Identify the Node to Delete

- Suppose you want to delete a node `N` with two children.
- `N` has a left child and a right child. We need to remove `N` and replace it with a suitable node so that the tree remains a valid BST.

2. Finding a Replacement Node

- To maintain the BST property after deletion, you need to find a node in the BST that can replace `N` while keeping the tree structure intact.
- **Two common strategies**:
  1. **In-order Predecessor**: The largest node in `N`'s left subtree.
  2. **In-order Successor**: The smallest node in `N`'s right subtree.
- **Why these nodes?**
  - **In-order Predecessor**: It's the largest node in the left subtree, so it's smaller than `N` but larger than all other nodes in the left subtree.
  - **In-order Successor**: It's the smallest node in the right subtree, so it's larger than `N` but smaller than all other nodes in the right subtree.

3. Choosing the In-order Successor

- For this explanation, we'll use the in-order successor, which is often preferred because it’s easier to find and manage.
- **Finding the In-order Successor**:
  - The in-order successor of `N` is the node with the smallest value in the right subtree of `N`.
  - To find it, go to the right child of `N` and keep moving to the left child until you reach the leftmost node.
- **Example**:
  - If `N` is 50, and the right subtree starts at 70, the in-order successor might be the leftmost child in the right subtree.

4. Replacing the Node

- Once you've identified the in-order successor (`S`):
  1. **Copy the value of `S` to `N`.**
  2. **Delete the original `S` node.** Note that `S` will have at most one child (it could be a leaf node, or have a right child). → The Right child is bigger than the successor it self
- **Why Delete the Original `S`?**
  - Since you've copied `S`'s value into `N`, the original `S` node is now redundant and needs to be removed.
  - The original `S` is either a leaf or has one right child, so its deletion is simpler (handled by cases 1 or 2 of the deletion process).

5. Implementation of Deletion with Two Children in Code

Here’s how this process looks in code:

Detailed Walkthrough with an Example

Let’s go through a concrete example:

Initial Tree:

```
   50
  / \
 30  70
 /\  /\
20 40 60 80
```

- Suppose we want to delete the node with value `50`.

Step 1: Identify the Node (`N = 50`) to Delete

- Node `50` has two children (`30` and `70`).

Step 2: Find the In-order Successor

- Look in the right subtree (`70`), and find the smallest node there (`60`).

Step 3: Replace Node `50` with `60`

- Replace the value of node `50` with `60`.

Step 4: Delete the Original `60` Node

- Node `60` was the in-order successor, and it had no left child. It can be removed directly.

Final Tree after Deletion:

```
   60
  / \
 30  70
 /\   \
20 40  80
```

- The BST properties are maintained: all nodes to the left of `60` are smaller, and all nodes to the right are larger.

Key Points to Remember:

- **The in-order successor ensures that the tree remains a valid BST after deletion**.
- **The process of deleting the original in-order successor node** is typically simpler since it has at most one child.
- **Alternatively, you could use the in-order predecessor** (largest node in the left subtree), and the procedure would be similar.

2. Height Calculation

The height of the tree is a critical measure, especially when considering balance. Here's how to calculate it:

```
public class BinarySearchTree<T extends Comparable<T>> {
    // ... (existing methods)

    // Method to calculate the height of the BST
    public int height() {
        return heightRec(root);
    }

    // Recursive method to calculate height
    private int heightRec(Node<T> root) {
        if (root == null) {
            return 0;
        } else {
            int leftHeight = heightRec(root.left);
            int rightHeight = heightRec(root.right);
```

```
            return Math.max(leftHeight, rightHeight) + 1;
        }
    }
}
```

▼ ⇒Draft

▼ String Literals

When first learning about `String`s in Java, you probably learned that you can create one by just putting it in quotes: `"THIS IS A STRING!"`. This specific process of creating a `String` is actually creating something called a *String literal*. Similarly to primitives, `String`s are a very common typing, so rather than creating new objects every time a `String` is created, it just creates **literals/constants** that are stored in a `String` pool for faster access. `String`s created in this way act very similarly to primitives in that there is no difference between value/reference equality! In fact, since they have the `String` typing, which extends `Object`, you can alternatively use the .equals() method if you want instead.

The other way to create a `String` is similar to any other `Object`, using the new keyword. This causes them to act like any other `Object` in terms of value/reference equality, allowing you to differentiate between the two based on whether you use `.equals()` or `==`. You won't need to know these interactions super well for this course, but it is good to understand the distinction.

```
String literal = "This is a string.";
String object = new String("This is a string.");
String unequal = "Nope.";
```

```
// Using == On Value Equal Strings
literal == object;                      // ⇒ false
literal == "This is a string.";         // ⇒ true
      "This is a string." == "This is a string.";  // ⇒ true
object == "This is a string.";          // ⇒ false
object == new String("This is a string.");   // ⇒ false

// Using .equals() On Value Equal Strings
literal.equals(object);                 // ⇒ true
object.equals(literal);                 // ⇒ true
literal.equals("This is a string.");    // ⇒ true
object.equals("This is a string.");     // ⇒ true
"This is a string.".equals("This is a string.");          // ⇒ true
"This is a string.".equals(new String("This is a string."));   // ⇒ true

// Using == and .equals() on Unequal Strings
literal == unequal;       // ⇒ false
object == unequal;        // ⇒ false
literal.equals(unequal);  // ⇒ false
object.equals(unequal);   // ⇒ false
```

▼ Null Checking

The primary use of == in this course is to check if an `Object` is `null`. This is usually done to check that the input to a method is valid, or to check for an exceptional situation. This is necessary because if you try to invoke a method or field of a `null` object, then the code will throw a `NullPointerException`. Since `.equals()` is a method, calling it on a null object will throw the `NullPointerException` as well. We sometimes use NPE to represent the phrase null pointer exception.

In debugging your code in this course, the most common Exception you will see is the `NullPointerException`. We encourage you to try and experiment with what causes it to be thrown

and what doesn't below. You may find the documentation for the String class useful.

```
String nullObject = null;
String normalObject = "normal";

// The correct way of checking if an object is null
nullObject == null;   // ⇒ true
normalObject == null; // ⇒ false

// The String class checks if argument is null, so it will throw NPE.
// If a .equals() method doesn't check for a null argument, it will crash and
// Here, we are comparing the value stored in normalObject to the values r
normalObject.equals(null);      // ⇒ false
normalObject.equals(nullObject); // ⇒ false

// However, here we invoking the .equals() method on a null object stored i
// This will cause a NPE to be thrown, regardless of the parameter in the .e
nullObject.equals(normalObject);  // causes NullPointerException
nullObject.equals(null);        // causes NullPointerException
```

Autoboxing and unboxing in Java are features that simplify the conversion between primitive types and their corresponding wrapper classes. Here's a brief overview:

Autoboxing

Definition: Autoboxing is the automatic conversion of a primitive type (e.g., `int`, `char`, `double`) into its corresponding wrapper class (e.g., `Integer`, `Character`, `Double`).

```
int primitiveInt = 5;
Integer wrapperInt = primitiveInt;  // Autoboxing occurs here
```

Explanation: Java automatically converts the `int` primitive type to an `Integer` object when needed, such as when assigning

it to a variable of type `Integer`.

Unboxing

Definition: Unboxing is the automatic conversion of a wrapper class (e.g., `Integer`, `Character`, `Double`) back to its corresponding primitive type (e.g., `int`, `char`, `double`).

Example:

```
Integer wrapperInt = 10;
int primitiveInt = wrapperInt;  // Unboxing occurs here
```

Explanation: Java automatically converts the `Integer` object back to the `int` primitive type when needed, such as when assigning it to a variable of type `int`.

Usage in Collections

Autoboxing and unboxing are commonly used in collections. For example, when using `ArrayList`, which can only hold objects, autoboxing converts primitives to their wrapper class equivalents so that they can be stored in the list.

```
ArrayList<Integer> list = new ArrayList<>();
list.add(42);  // Autoboxing converts int to Integer
int value = list.get(0);  // Unboxing converts Integer to int
```

Notes

- **Performance**: While autoboxing and unboxing simplify code, they can introduce performance overhead due to the creation of wrapper objects and boxing/unboxing operations.
- **Null Handling**: Be cautious with wrapper classes because they can be `null`, while primitive types cannot. This can lead to `NullPointerException` if not handled properly.

Autoboxing and unboxing are convenient features in Java that enhance code readability and reduce the need for explicit conversion between primitives and objects.

▼ Wrapper Objects

Recall how our String literals and `String` objects are related and operate? Well, we can do that with primitive types. Each of the primitive types have a wrapper `Object` class (i.e. `int` -> `Integer`, `double` -> `Double`, etc.). While primitives are optimized, there are times where you need an `Object` for the code to work. For example, generic typings implictly require the type to inherit from the `Object` class, meaning that if you wanted to make some Collection of `int`s, then you'd need to make `Integer` objects to add rather than primitives. If you've never noticed this before, that's because Java has features like autoboxing/unboxing in order to streamline the experience. These wrapper class typings behave much like `String`s do with respect to literals and objects with some slight differences. Again, you do not need to know these interactions well, but you should understand that there is a difference in behavior.

```
Integer primitive = 1;
Integer object = new Integer(1);
Integer unequal = 2;

// Using == On Value Equal Integers
primitive == object;       // ⇒ false
primitive == 1;            // ⇒ true
object == 1;               // ⇒ true
object == new Integer(1);  // ⇒ false

// Using .equals() On Value Equal Integers
primitive.equals(object);              // ⇒ true
object.equals(primitive);              // ⇒ true
primitive.equals(1);                   // ⇒ true
```

```
object.equals(1);                    // ⇒ true
(new Integer(1)).equals(1);          // ⇒ true
(new Integer(1)).equals(new Integer(1));  // ⇒ true

// Using == and .equals() on Unequal Integers
primitive == unequal;       // ⇒ false
object == unequal;          // ⇒ false
primitive.equals(unequal);  // ⇒ false
object.equals(unequal);     // ⇒ false

// If run, the following do not work since 1 is considered a primitive without
1.equals(primitive);  // causes Error
1.equals(1);          // causes Error
```

▼ Iterator & Iterable

We know that `Iterable` allows a class to be iterated over, either by the returned `Iterator` from the `iterator()` method or by a for-each loop (which internally, uses an `Iterator`).

```
List<String> structures = new ArrayList<>();
structures.add("BST");
structures.add("HashMap");
structures.add("Graph");

// simple for each loop that iterates through the ArrayList "structures"
for (String structure: structures) {
   System.out.println("I love " + structure + "s!");
}
```

Here's a summary of the `Iterator` abstract class, `Iterable` interface, `Comparable` interface, and `Comparator` interface in Java:

1. Iterator Interface

Purpose: Provides a way to access elements of a collection sequentially without exposing the underlying structure.

Key Methods:

- `boolean hasNext()`: Returns `true` if there are more elements to iterate over.
- `E next()`: Returns the next element in the iteration.
- `void remove()`: Removes the last element returned by the iterator (optional operation).

Usage:

```
Iterator<String> iterator = list.iterator();
while (iterator.hasNext()) {
    String element = iterator.next();
    // Process element
}
```

2. Iterable Interface

Purpose: Represents a collection of elements that can be iterated over. It provides an `Iterator` for its elements.

Key Method:

- `Iterator<E> iterator()`: Returns an iterator over elements of type `E`.

Usage:

- Classes that implement `Iterable` can be used in enhanced for-loops (for-each loops).

```
for (String element : list) {
    // Process element
}
//examble implementation:
```

```
public class MyCollection implements Iterable<String> {
    private List<String> data = new ArrayList<>();
    @Override
    public Iterator<String> iterator() {
        return data.iterator();
    }
}
```

3. Comparable Interface

Purpose: Defines a natural ordering for objects of a class. Classes implementing this interface should provide a `compareTo` method.

Key Method:

- `int compareTo(T o)`: Compares the current object with the specified object to determine their relative ordering.

Usage:

- Useful for sorting objects naturally (e.g., using `Collections.sort()`).

Example:'

```
public class Person implements Comparable<Person> {
    private String name;
    public int compareTo(Person other) {
        return this.name.compareTo(other.name);
    }
}
```

4. Comparator Interface

Purpose: Defines a custom ordering for objects. It is used to compare two objects of a class when you need a different ordering than the natural one.

Key Methods:

- `int compare(T o1, T o2)`: Compares two objects `o1` and `o2` for order.
- `boolean equals(Object obj)`: Checks if another comparator is equal to this one (optional).

Usage:

- Useful for defining multiple sorting criteria or when the class does not implement `Comparable`.

Example:

```
public class PersonNameComparator implements Comparator<Person> {
    public int compare(Person p1, Person p2) {
        return p1.getName().compareTo(p2.getName());
    }
}
//Using with sorting
List<Person> people = new ArrayList<>();
// Add people to the list
Collections.sort(people, new PersonNameComparator());
```

Summary

- `Iterator`: Provides a way to traverse through a collection.
- `Iterable`: Allows a class to be iterated over using an `Iterator`.
- `Comparable`: Provides a natural ordering for objects of a class.
- `Comparator`: Allows for custom ordering of objects, useful for multiple or alternative sorting criteria.

▼ Generics in java

▼ Problem Before Generics

Before generics, Java code often used `Object` references to store any type of object, which had several drawbacks:

1. **Type Safety:** Since `Object` can hold any type, there was no compile-time checking for type errors. This could lead to `ClassCastException` at runtime.
2. **Type Casting:** Programmers had to cast objects back to their original types, which was cumbersome and error-prone.
3. **Code Reusability:** Without generics, code had to be duplicated for different types, leading to redundancy and maintenance difficulties.

▼ How Generics Solve These Problems

1. **Type Safety:** Generics provide compile-time type checking, ensuring that code is type-safe. Errors are caught during compilation rather than at runtime.
2. **Elimination of Casts:** With generics, explicit type casting is no longer necessary. The compiler automatically handles type conversion.
3. **Code Reusability:** Generics enable the creation of classes, interfaces, and methods that can operate on any data type, reducing code duplication.

▼ How Generics Work

Generics are implemented using a process called type erasure, which replaces all type parameters with their bounds or `Object` if the type parameters are unbounded. The compiler then inserts appropriate casts to maintain type safety.

▼ Generic Classes

Simple generic class

```
public class Printer <T>{
    T thingToPrint;
    public Printer(T thingToPrint){
        this.thingToPrint = thingToPrint;
    }
    public void print(){
        System.out.println(thingToPrint);
    }
}
```

Generic class with Certain type of classes

```
public class Printer <T extends Animal>{
    T thingToPrint;
    public Printer(T thingToPrint){
        this.thingToPrint = thingToPrint;
    }
    public void print(){
        System.out.println(thingToPrint);
    }
}
// this class is used only for the classes that extentds Animal Class
```

▼ Generic Methods

Simple generic method

```
public static <T> void printArray(T[] inputArray) {
    for (T element : inputArray) {
        System.out.printf("%s ", element);
    }
```

```
    System.out.println();
}
// this method print arrays of any type you want
```

Bounded Type Parameters

```
public <T extends Number> void inspect(T t) {
    System.out.println("T: " + t.getClass().getName());
}
// this method works on any type of classes that extends Number Clas
```

▼ Bounded Generics

Bounded Generics in Java

Bounded generics in Java allow you to specify constraints on the types that can be used as type arguments. This means you can restrict the type parameters to a specific range of types. There are two types of bounds in generics: **upper bounds** and **lower bounds**.

Upper Bounded Wildcards

Upper bounded wildcards restrict the type parameter to be a subtype of a specific type. This is done using the `extends` keyword.

Example of Upper Bounded Wildcards:

```
public class UpperBoundedExample {
    public static void printNumbers(List<? extends Number> list) {
        for (Number num : list) {
            System.out.println(num);
        }
```

```
    }

    public static void main(String[] args) {
        List<Integer> intList = Arrays.asList(1, 2, 3, 4);
        List<Double> doubleList = Arrays.asList(1.1, 2.2, 3.3, 4.4);

        printNumbers(intList); // Works, Integer is a subclass of Number
        printNumbers(doubleList); // Works, Double is a subclass of Numb
    }
}
```

In this example, `printNumbers` accepts a list of any type that is a subclass of `Number`.

Lower Bounded Wildcards

Lower bounded wildcards restrict the type parameter to be a supertype of a specific type. This is done using the `super` keyword.

Example of Lower Bounded Wildcards:

```
public class LowerBoundedExample {
    public static void addNumbers(List<? super Integer> list) {
        list.add(1);
        list.add(2);
    }

    public static void main(String[] args) {
        List<Number> numList = new ArrayList<>();
        List<Object> objList = new ArrayList<>();

        addNumbers(numList); // Works, Number is a supertype of Intege
```

```
        addNumbers(objList); // Works, Object is a supertype of Integer

        System.out.println(numList);
        System.out.println(objList);
    }
}
```

In this example, `addNumbers` accepts a list of any type that is a supertype of `Integer`.

Bounded Type Parameters

You can also use bounds in generic type parameters when defining classes or methods.

Example of Bounded Type Parameters in Classes

```
public class BoundedGenericClass<T extends Number> {
    private T t;

    public BoundedGenericClass(T t) {
        this.t = t;
    }

    public void display() {
        System.out.println("Value: " + t);
    }

    public static void main(String[] args) {
        BoundedGenericClass<Integer> intObj = new BoundedGenericCla
        BoundedGenericClass<Double> doubleObj = new BoundedGener

        intObj.display(); // Output: Value: 100
```

```
        doubleObj.display(); // Output: Value: 99.99
    }
}
```

⇒ In this example, `BoundedGenericClass` only accepts type parameters that are subclasses of `Number`.

Example of Bounded Type Parameters in Methods:

```
public class BoundedGenericMethod {
    public static <T extends Comparable<T>> T findMax(T[] array)
    /* Checks if the input parameter can be compared*/
    {
        T max = array[0];
        for (T element : array) {
            if (element.compareTo(max) > 0) {
                max = element;
            }
        }
        return max;
    }

    public static void main(String[] args) {
        Integer[] intArray = {1, 2, 3, 4, 5};
        String[] strArray = {"apple", "orange", "banana"};

        System.out.println(findMax(intArray)); // Output: 5
        System.out.println(findMax(strArray)); // Output: orange
    }
}
```

In this example, the

`findMax` method only accepts arrays of elements that implement the `Comparable` interface.

Wildcards

- **Unbounded Wildcard:** `<?>` represents any type.
- **Bounded Wildcards:**
  - `<? extends T>` represents a type that is a subtype of `T`.
  - `<? super T>` represents a type that is a supertype of `T`.

Benefits of Generics

1. **Strong Type Checking:** Compile-time type checking prevents type errors during runtime.
2. **Code Reusability:** Write once and use with different data types without code duplication.
3. **Elimination of Type Casting:** Reduces the need for explicit type casting, making code cleaner and easier to read.
4. **Generic Algorithms:** Enables the creation of algorithms that work on collections of different types, promoting code reuse and robustness.

Before Generics:

```
List list = new ArrayList();
list.add("Hello");
String s = (String) list.get(0);  // Type casting required
```

After Generics:

```
List<String> list = new ArrayList<>();
list.add("Hello");
String s = list.get(0);  // No type casting required
```

There is something called Autoboxing that is used automatically when you make a generic class with the wrapper of a primitive data type

Example:

```
Stack<Integer> s = new Stack<Integer>();
s.push(17); // s.push(Integer.valueOf(17));
int a = s.pop(); // int a = s.pop().intValue();
```