Bubble Sort Algorithm:

Bubble Sort is a simple comparison-based sorting algorithm. It repeatedly compares adjacent elements in the list and swaps them if they are in the wrong order. This process is repeated until the list is sorted.

Algorithm Steps:

1. Start at the beginning of the array.
2. Compare the current element with the next element.
3. If the current element is greater than the next element, swap them.
4. Repeat this process for every pair of adjacent elements in the array.
5. Reduce the range of comparison by one in each iteration as the largest elements "bubble up" to their correct position.
6. Stop when no swaps are needed, indicating the array is sorted.

Time Complexity:

1. **Best Case:** O(n)
    - Occurs when the array is already sorted, requiring no swaps.
    - Optimized implementations check for this condition by tracking swaps.
2. **Average Case:** O(n^2)
    - Most of the time, elements will require partial swaps for sorting.
3. **Worst Case:** O(n^2)
    - Happens when the array is in reverse order, requiring the maximum number of swaps.

Space Complexity:

- **Space:** O(1) (in-place sorting algorithm; no additional memory is required).

```
import java.util.Scanner;

public class BubbleSortSteps {

   public static void main(String[] args) {

      Scanner scanner = new Scanner(System.in);

      // Input the size of the array
```

```
    System.out.print("Enter the number of elements: ");
    int n = scanner.nextInt();
    // Input array elements
    int[] arr = new int[n];
    System.out.println("Enter " + n + " elements:");
    for (int i = 0; i < n; i++) {
        arr[i] = scanner.nextInt();
    }
    System.out.println("Original array:");
    printArray(arr);
    // Perform Bubble Sort with steps
    bubbleSortWithSteps(arr);
    System.out.println("Sorted array:");
    printArray(arr);
    scanner.close();
}
// Bubble sort with step-by-step output
public static void bubbleSortWithSteps(int[] arr) {
    int n = arr.length;
    boolean swapped;

    for (int i = 0; i < n - 1; i++) {
        swapped = false;
        System.out.println("\nPass " + (i + 1) + ":");
        for (int j = 0; j < n - 1 - i; j++) {
            if (arr[j] > arr[j + 1]) {
                // Swap arr[j] and arr[j + 1]
                int temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;
```

```
                swapped = true;
            }
            // Print array after each comparison
            printArray(arr);
        }
        // If no swaps occurred, array is sorted
        if (!swapped) {
            System.out.println("No swaps needed. Array is sorted.");
            break;
        }
    }
}
// Utility method to print the array
public static void printArray(int[] arr) {
    for (int num : arr) {
        System.out.print(num + " ");
    }
    System.out.println();
}
}
```

How It Works:

1. The program asks the user to input the size of the array and its elements.
2. The bubbleSortWithSteps method sorts the array step by step, printing the array's state after every comparison and after each pass.
3. If a pass results in no swaps, the program terminates early, as the array is already sorted.

Example Input and Output:

Input:

Enter the number of elements: 5

Enter 5 elements:

64 34 25 12 22

Output:

Original array:

64 34 25 12 22

Pass 1:

34 64 25 12 22

34 25 64 12 22

34 25 12 64 22

34 25 12 22 64

Pass 2:

25 34 12 22 64

25 12 34 22 64

25 12 22 34 64

Pass 3:

12 25 22 34 64

12 22 25 34 64

Pass 4:

12 22 25 34 64

Sorted array:

12 22 25 34 64

Selection Sort Algorithm:

Selection Sort is a simple sorting algorithm. It works by dividing the array into a sorted and an unsorted region. The smallest (or largest, depending on sorting order) element from the unsorted region is selected and swapped with the first element of the unsorted region, expanding the sorted region by one.

Algorithm Steps:

1. Start with the first element of the array.
2. Find the smallest element in the unsorted portion of the array.
3. Swap the smallest element with the first element of the unsorted portion.
4. Move the boundary of the sorted region one element to the right.
5. Repeat until the entire array is sorted.

Time Complexity:

1. Best Case: O(n^2)
   - Even if the array is already sorted, the algorithm performs all comparisons.

2. Average Case: O(n^2)
   - The algorithm always compares all elements, regardless of initial order.

3. Worst Case:O(n^2)
   - Same as the average case because the algorithm's structure doesn't change with input.

Space Complexity:

Space:O(1) (in-place sorting algorithm).

Java Program: Selection Sort with Step-by-Step Output

```
import java.util.Scanner;
public class SelectionSortSteps {
   public static void main(String[] args) {
      Scanner scanner = new Scanner(System.in);

      // Input the size of the array
      System.out.print("Enter the number of elements: ");
      int n = scanner.nextInt();

      // Input array elements
      int[] arr = new int[n];
      System.out.println("Enter " + n + " elements:");
      for (int i = 0; i < n; i++) {
```

```
            arr[i] = scanner.nextInt();
        }

        System.out.println("Original array:");
        printArray(arr);

        // Perform Selection Sort with steps
        selectionSortWithSteps(arr);

        System.out.println("Sorted array:");
        printArray(arr);
        scanner.close();
    }

    // Selection sort with step-by-step output
    public static void selectionSortWithSteps(int[] arr) {
        int n = arr.length;

        for (int i = 0; i < n - 1; i++) {
            int minIndex = i; // Index of the smallest element
            for (int j = i + 1; j < n; j++) {
                if (arr[j] < arr[minIndex]) {
                    minIndex = j;
                }
            }
            // Swap the smallest element with the first element of the unsorted
portion
            int temp = arr[minIndex];
            arr[minIndex] = arr[i];
            arr[i] = temp;

            // Print the array after each pass
            System.out.println("\nAfter pass " + (i + 1) + ":");
            printArray(arr);
        }
    }
    // Utility method to print the array
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

How It Works:

1. The program takes the size and elements of the array as input from the user.
2. The `selectionSortWithSteps` method:
   - Finds the smallest element in the unsorted part of the array.
   - Swaps it with the first element of the unsorted portion.
   - Prints the state of the array after each pass.
3. The process repeats until the array is sorted.

Example Input and Output:

Input:

Enter the number of elements: 5
Enter 5 elements:
64 34 25 12 22

Output:
Original array:
64 34 25 12 22

After pass 1:
12 34 25 64 22

After pass 2:
12 22 25 64 34

After pass 3:
12 22 25 64 34

After pass 4:
12 22 25 34 64

Sorted array:
12 22 25 34 64

Insertion Sort Algorithm:

Insertion Sort is a simple and efficient algorithm for small datasets. It works by building a sorted portion of the array one element at a time. Each new element is compared with the elements in the sorted portion and inserted in its correct position.

Algorithm Steps:

1. Start with the second element (index 1), assuming the first element is already sorted.
2. Compare the current element with the elements in the sorted portion.
3. Shift elements of the sorted portion that are larger than the current element to the right.
4. Insert the current element in its correct position within the sorted portion.
5. Repeat for all elements in the array.

Time Complexity:

1. Best Case:O(n)
- Happens when the array is already sorted, requiring only one comparison per element.

2. **Average Case:O(n^2)**
- Elements are partially out of order, requiring multiple comparisons and shifts.

3. **Worst Case:O(n^2)**
- Occurs when the array is sorted in reverse order, requiring maximum comparisons and shifts.

Space Complexity:
- Space: O(1) (in-place sorting algorithm).

Java Program: Insertion Sort with Step-by-Step Output

```
import java.util.Scanner;

public class InsertionSortSteps {

    public static void main(String[] args) {

        Scanner scanner = new Scanner(System.in);

        // Input the size of the array

        System.out.print("Enter the number of elements: ");

        int n = scanner.nextInt();

        // Input array elements

        int[] arr = new int[n];

        System.out.println("Enter " + n + " elements:");
```

```
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        System.out.println("Original array:");
        printArray(arr);
        // Perform Insertion Sort with steps
        insertionSortWithSteps(arr);
        System.out.println("Sorted array:");
        printArray(arr);
        scanner.close();
    }
    // Insertion sort with step-by-step output
    public static void insertionSortWithSteps(int[] arr) {
        int n = arr.length;
        for (int i = 1; i < n; i++) {
            int key = arr[i]; // Current element to be inserted
            int j = i - 1;
            // Move elements of the sorted portion that are greater than key
            while (j >= 0 && arr[j] > key) {
                arr[j + 1] = arr[j];
                j--;
            }
            // Insert the current element at the correct position
            arr[j + 1] = key;
            // Print the array after each insertion
            System.out.println("\nAfter inserting element " + key + ":");
            printArray(arr);
        }
    }
// Utility method to print the array
```

```
    public static void printArray(int[] arr) {
        for (int num : arr) {
            System.out.print(num + " ");
        }
        System.out.println();
    }
}
```

How It Works:

1. The program takes input for the array size and elements from the user.

2. The `insertionSortWithSteps` method:

   - Iterates through the array starting from the second element.

   - Finds the correct position for the current element by shifting larger elements in the sorted portion to the right.

   - Inserts the current element and prints the array state after each insertion.

3. The sorted array is displayed at the end.

Example Input and Output:

Input:

```
Enter the number of elements: 5
Enter 5 elements:
64 34 25 12 22
```

Output:

```
Original array:
64 34 25 12 22
After inserting element 34:
34 64 25 12 22
After inserting element 25:
25 34 64 12 22
After inserting element 12:
12 25 34 64 22
After inserting element 22:
12 22 25 34 64

Sorted array:
12 22 25 34 64
```

Shell Sort Algorithm:

Shell Sort is an advanced version of the Insertion Sort. It improves the efficiency of insertion sort by allowing the comparison and exchange of elements that are far apart. This is done using a sequence of gaps between compared elements that gradually decreases to 1, when it performs a final Insertion Sort pass.

Algorithm Steps:

1. Choose an initial gap size: The gap size is typically chosen as half of the array length, and it is reduced progressively (e.g., divided by 2).

2. Perform gapped insertion sort: For each gap size, perform an insertion sort where instead of comparing adjacent elements, elements that are gap positions apart are compared.

3. Reduce the gap: After each pass, the gap size is reduced (usually by dividing by 2), and the process is repeated until the gap becomes 1, at which point a final insertion sort is performed.

Time Complexity:

- Best Case: $O(n \log n)$

  - If the gap sequence is chosen well (e.g., using the Hibbard or Sedgewick sequences), the time complexity can approach $O(n \log n)$.

- Average Case: $O(n^{1.3})$ to $O(n^2)$

  - Depends on the choice of gap sequence. The standard gap sequence often results in $O(n^{1.3})$ in practice.

- Worst Case: $O(n^2)$

  - Occurs with certain gap sequences (like the original sequence used by Shell), leading to $O(n^2)$ behavior.

Space Complexity:

- Space: $O(1)$

  - Shell Sort is an in-place sorting algorithm, meaning it requires no extra space apart from the input array.

Java Program: Shell Sort with Step-by-Step Output

```
import java.util.Scanner;
public class ShellSortSteps {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input the size of the array
        System.out.print("Enter the number of elements: ");
        int n = scanner.nextInt();
        // Input array elements
        int[] arr = new int[n];
        System.out.println("Enter " + n + " elements:");
        for (int i = 0; i < n; i++) {
            arr[i] = scanner.nextInt();
        }
        System.out.println("Original array:");
        printArray(arr);
        // Perform Shell Sort with steps
        shellSortWithSteps(arr);
        System.out.println("Sorted array:");
        printArray(arr);
        scanner.close();
    }
    // Shell Sort with step-by-step output
    public static void shellSortWithSteps(int[] arr) {
        int n = arr.length;
        // Start with a large gap, then reduce the gap
        for (int gap = n / 2; gap > 0; gap /= 2) {
            System.out.println("\nGap = " + gap + ":");
                    // Perform a gapped insertion sort
            for (int i = gap; i < n; i++) {
```

```
            int temp = arr[i];
            int j;
        // Shift earlier gap-sorted elements up until the correct location is found
            for (j = i; j >= gap && arr[j - gap] > temp; j -= gap) {
                arr[j] = arr[j - gap];
            }
            // Put temp (the original arr[i]) in its correct location
            arr[j] = temp;
            // Print the array after each insertion
            printArray(arr);
        }
    }
}
// Utility method to print the array
public static void printArray(int[] arr) {
    for (int num : arr) {
        System.out.print(num + " ");
    }
    System.out.println();
}
}
```

How It Works:

1. User Input: The program takes the array size and its elements as input from the user.

2. Shell Sort Process:

- The program first selects a gap (typically \(n/2\)) and then performs a gapped insertion sort.

- After each pass, the gap is reduced (usually halved).

- For each gap, the program shows the array after sorting with that gap.

3. Final Sorting: When the gap reaches 1, a final pass is performed, and the array is fully sorted.

Example Input and Output:

Input:

Enter the number of elements: 5

Enter 5 elements:

64 34 25 12 22

Output:

Original array:

64 34 25 12 22

Gap = 2:

25 34 64 12 22

25 34 64 12 22

25 34 22 12 64

Gap = 1:

22 25 34 12 64

22 25 12 34 64

22 12 25 34 64

12 22 25 34 64

Sorted array:

12 22 25 34 64

Explanation of Output:

1. Gap = 2: Elements are compared with a gap of 2. After the first pass, some elements are shifted, and the array becomes closer to being sorted.

2. Gap = 1: After the gap is reduced to 1, it performs a final insertion sort, where elements are compared adjacent to each other, and the array is fully sorted.

Key Features of Shell Sort:

- Improved Performance: Shell Sort significantly reduces the number of comparisons compared to the regular Insertion Sort by allowing comparisons between elements far apart.

- Gap Sequence: The choice of gap sequence is critical for its efficiency. The program uses a simple gap halving strategy.

Radix Sort in Detail

Radix Sort is a non-comparative sorting algorithm that works by distributing elements into buckets according to their individual digits. It processes the digits from the least significant to the most significant (LSD) or vice versa (MSD). Radix Sort is particularly useful for sorting integers or strings where the keys can be broken down into digits or characters.

Algorithm

1. Find the maximum number to know the number of digits.
2. Use counting sort (a stable sorting algorithm) for each digit.
    - Sort numbers by each digit, starting with the least significant digit.
3. Repeat the process for every digit until all the digits are processed.

Steps of Radix Sort

Let's say we have an array: [170, 45, 75, 90, 802, 24, 2, 66].

1. Find the maximum number:
   Maximum = 802 → Number of digits = 3.
2. Sort by the least significant digit (unit place):
   After sorting: [802, 2, 24, 45, 66, 170, 75, 90].
3. Sort by the next digit (tens place):
   After sorting: [802, 2, 24, 45, 66, 75, 90, 170].
4. Sort by the most significant digit (hundreds place):
   After sorting: [2, 24, 45, 66, 75, 90, 170, 802].

```
import java.util.Arrays;
public class RadixSort {
   // Main function to implement Radix Sort
   public static void radixSort(int[] arr) {
      // Find the maximum number to determine the number of digits
      int max = Arrays.stream(arr).max().getAsInt();
      // Apply counting sort for each digit
      for (int exp = 1; max / exp > 0; exp = 10) {
         countingSort(arr, exp);
```

```
        }
    }
    Counting sort used as a subroutine
    private static void countingSort(int[] arr, int exp) {
        int n = arr.length;
        int[] output = new int[n]; // Output array to store sorted numbers
        int[] count = new int[10]; // Count array for digits 0-9
        // Count the occurrences of each digit at the current position
        for (int i = 0; i < n; i++) {
            int digit = (arr[i] / exp) % 10;
            count[digit]++;
        }
        // Update count[i] to store the actual position of this digit in output
        for (int i = 1; i < 10; i++) {
            count[i] += count[i - 1];
        }
        // Build the output array by placing numbers in sorted order of the
current digit
        for (int i = n - 1; i >= 0; i--) {
            int digit = (arr[i] / exp) % 10;
            output[count[digit] - 1] = arr[i];
            count[digit]--;
        }
       // Copy the sorted numbers back to the original array
        System.arraycopy(output, 0, arr, 0, n);
    }
   // Driver function to test the algorithm
    public static void main(String[] args) {
        int[] arr = {170, 45, 75, 90, 802, 24, 2, 66};
        System.out.println("Original Array: " + Arrays.toString(arr));
```

```
        radixSort(arr);
        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }
}
```

Step-by-Step Execution

1. Input Array: [170, 45, 75, 90, 802, 24, 2, 66]
2. Sorting by Unit Place (exp = 1):
   Count: [2, 1, 0, 0, 1, 1, 1, 0, 1, 1]
   Sorted: [802, 2, 24, 45, 66, 170, 75, 90]
3. Sorting by Tens Place (exp = 10):
   Count: [2, 1, 0, 0, 2, 1, 0, 1, 0, 1]
   Sorted: [802, 2, 24, 45, 66, 75, 90, 170]
4. Sorting by Hundreds Place (exp = 100):
   Count: [4, 2, 2, 0, 0, 0, 0, 0, 0, 0]
   Sorted: [2, 24, 45, 66, 75, 90, 170, 802]

Applications of Radix Sort

1. Sorting Integers or Strings: Especially useful for large datasets where keys are uniformly distributed.
2. Telephone Directories: Efficient for sorting numbers and names based on multi-key fields.
3. Digital Data: Used in applications involving sorting IP addresses, postal codes, or dates.
4. Preprocessing for Histogram Equalization: Helps in graphics or image processing.
5. Big Data: Used in parallel processing environments due to its non-comparative nature.

Advantages and Limitations

Advantages:

1. Linear Time Complexity: O(nk)O(nk) where nn is the number of elements, and kk is the number of digits.
2. Stable Sort: Preserves the order of equal elements.

3. Non-Comparative: Does not use comparison operators, making it fast for integers.

Limitations:

1. Not In-Place: Requires extra space for counting sort.
2. Depends on Data Type: Works efficiently only for data that can be digitized (like integers or fixed-length strings).
3. Large Range Handling: Inefficient for datasets with very large keys or floating-point numbers.

Quick Sort in Detail

Quick Sort is a divide-and-conquer sorting algorithm that selects a "pivot" element, partitions the array into elements smaller than the pivot and elements greater than the pivot, and recursively sorts the subarrays.

Algorithm

1. Choose a Pivot: Select an element as the pivot (commonly the last element, first element, or middle element).
2. Partition the Array: Rearrange the array so that:
    - All elements smaller than the pivot are on the left.
    - All elements larger than the pivot are on the right.
3. Recursively Apply Quick Sort: Apply the above steps to the left and right subarrays.

Steps of Quick Sort

Let’s sort: [10, 80, 30, 90, 40, 50, 70].

1. Choose Pivot: Select the last element as the pivot (70).
2. Partition: Rearrange the array:
    - Elements smaller than 70: [10, 30, 40, 50].
    - Pivot (70).
    - Elements larger than 70: [80, 90]. Result: [10, 30, 40, 50, 70, 80, 90].
3. Recursively Sort Subarrays: Sort [10, 30, 40, 50] and [80, 90].

Java Code Example

```
import java.util.Arrays;
public class QuickSort {
    // Function to perform Quick Sort
    public static void quickSort(int[] arr, int low, int high) {
        if (low < high) {
            // Partition the array and get the pivot index
            int pi = partition(arr, low, high);
            // Recursively sort elements before and after the pivot
            quickSort(arr, low, pi - 1);
            quickSort(arr, pi + 1, high);
        }
    }
    // Partition function
    private static int partition(int[] arr, int low, int high) {
        int pivot = arr[high]; // Pivot element
        int i = low - 1; // Index of smaller element
        for (int j = low; j < high; j++) {
            // If the current element is smaller than the pivot
            if (arr[j] < pivot) {
                i++;
                // Swap arr[i] and arr[j]
                int temp = arr[i];
                arr[i] = arr[j];
                arr[j] = temp;
            }
        }
        // Swap arr[i+1] and arr[high] (the pivot)
        int temp = arr[i + 1];
        arr[i + 1] = arr[high];
```

```
        arr[high] = temp;
        return i + 1; // Return the pivot index
    }
    // Main method to test the Quick Sort
    public static void main(String[] args) {
        int[] arr = {10, 80, 30, 90, 40, 50, 70};
        System.out.println("Original Array: " + Arrays.toString(arr));
        quickSort(arr, 0, arr.length - 1);
        System.out.println("Sorted Array: " + Arrays.toString(arr));
    }
}
```

Step-by-Step Execution

Input Array: [10, 80, 30, 90, 40, 50, 70]

1. First Partition:
    - Pivot = 70.
    - Rearrange: [10, 30, 40, 50, 70, 80, 90].
    - Pivot index = 4.
2. Left Subarray: [10, 30, 40, 50].
    - Pivot = 50.
    - Rearrange: [10, 30, 40, 50].
    - Pivot index = 3.
3. Right Subarray: [80, 90].
    - Pivot = 90.
    - Rearrange: [80, 90].
    - Pivot index = 6.

Final sorted array: [10, 30, 40, 50, 70, 80, 90].

Applications of Quick Sort

1. General Purpose Sorting: Used in many programming libraries due to its efficiency.
2. Data Processing: For sorting data in large-scale applications.
3. Searching Algorithms: Prepares data for binary searches or merge-like operations.
4. Database Management: Frequently used for query optimization and index sorting.
5. Distributed Systems: Adapted for parallel and distributed sorting tasks.

Advantages and Disadvantages

Advantages:

1. Efficient: Performs well for large datasets.
2. In-Place: Requires minimal additional memory.
3. Adaptable: Works well with parallelization.

Disadvantages:

1. Worst Case: Poor pivot choices can degrade performance to O(n^2).
2. Not Stable: Relative order of equal elements may not be preserved.

Linear Search and Binary Search

Linear Search and Binary Search are two fundamental searching algorithms. Linear Search works on any collection, while Binary Search requires the collection to be sorted.

Linear Search

Algorithm

1. Start from the first element of the array.
2. Compare the current element with the target value.
3. If they match, return the index of the current element.
4. If no match is found after traversing the entire array, return -1 (element not found).

Example Code in Java

```
public class LinearSearch {
   // Function to perform Linear Search
   public static int linearSearch(int[] arr, int target) {
      for (int i = 0; i < arr.length; i++) {
         if (arr[i] == target) {
            return i; // Return the index if found
         }
```

```
        }
        return -1; // Return -1 if not found
    }
    public static void main(String[] args) {
        int[] arr = {10, 20, 30, 40, 50};
        int target = 30;
        int result = linearSearch(arr, target);
        if (result != -1) {
            System.out.println("Element found at index: " + result);
        } else {
            System.out.println("Element not found.");
        }
    }
}
```

Step-by-Step Execution

1. Input Array: [10, 20, 30, 40, 50], Target: 30.
2. Start at index 0:
    - Compare arr[0] (10) with 30. Not a match.
3. Move to index 1:
    - Compare arr[1] (20) with 30. Not a match.
4. Move to index 2:
    - Compare arr[2] (30) with 30. Match found at index 2.

Output: Element found at index: 2.

Applications of Linear Search

1. Unsorted Arrays: Works on any unsorted dataset.
2. Small Data: Efficient for small datasets.
3. Flexible: Useful when the dataset does not support sorting or indexing.

Binary SearchAlgorithm

1. Sort the array (if not already sorted).
2. Initialize two pointers: low = 0 and high = arr.length - 1.
3. Calculate the middle index: mid = low + (high - low) / 2.
4. Compare the target value with arr[mid]:
    - If the target is equal to arr[mid], return mid.
    - If the target is smaller than arr[mid], search the left subarray.
    - If the target is larger than arr[mid], search the right subarray.
5. Repeat until the target is found or low > high.

Example Code in Java

```
import java.util.Arrays;
public class BinarySearch {
   // Function to perform Binary Search
   public static int binarySearch(int[] arr, int target) {
      int low = 0, high = arr.length - 1;
      while (low <= high) {
         int mid = low + (high - low) / 2;
         if (arr[mid] == target) {
            return mid; // Return the index if found
         } else if (arr[mid] < target) {
            low = mid + 1; // Search in the right subarray
         } else {
            high = mid - 1; // Search in the left subarray
         }
      }
      return -1; // Return -1 if not found
   }
   public static void main(String[] args) {
      int[] arr = {10, 20, 30, 40, 50};
      int target = 30;
```

```
        int result = binarySearch(arr, target);
        if (result != -1) {
            System.out.println("Element found at index: " + result);
        } else {
            System.out.println("Element not found.");
        }
    }
}
```

Step-by-Step Execution

1. Input Array: [10, 20, 30, 40, 50] (Sorted), Target: 30.
2. First Iteration:
    - low = 0, high = 4, mid = 2.
    - Compare arr[mid] (30) with 30. Match found at index 2.

Output: Element found at index: 2.

Applications of Binary Search

1. Large Sorted Arrays: Efficient for datasets where nn is large.
2. Databases: Used for searching in sorted database tables.
3. Finding Bounds: Used in algorithms like finding the smallest/largest element satisfying a condition.
4. Strings: Works on sorted strings for dictionary-like operations.

Comparison of Linear and Binary Search

Feature	**Linear Search**	**Binary Search**
Data Requirement	Works on unsorted data	Requires sorted data
Best Case	O(1)	O(1)
Worst Case	O(n)	O(log n)

Feature	Linear Search	Binary Search
Space Complexity	O(1)	O(1)
Applications	Small, unsorted data	Large, sorted data

Stack Data Structure

A stack is a linear data structure that follows the Last In First Out (LIFO) principle. The element added last is the first one to be removed. It can be visualized as a collection of elements where you can only insert or remove items from the top (the last added item).

Operations in Stack:

1. Push: Adds an element to the top of the stack.
2. Pop: Removes the top element from the stack.
3. Peek (Top): Returns the top element without removing it.
4. isEmpty: Checks if the stack is empty.
5. Size: Returns the size of the stack.

Time Complexity of Stack Operations:

- Push Operation: \(O(1)\)
- Pop Operation: \(O(1)\)
- Peek Operation: \(O(1)\)
- isEmpty Operation: \(O(1)\)
- Size Operation: \(O(1)\)

Since the stack is implemented using a linked list, each operation (push, pop, peek) takes constant time, as it only involves modifying or checking the top node of the list.

Implementing Stack Using Linked List in Java

Here's a Java program that implements a stack using a linked list. The program is menu-driven and allows the user to perform operations such as push, pop, peek, check if the stack is empty, and get the stack size.

```
import java.util.Scanner;
// Node class to represent each element in the linked list
class Node {
    int data;
    Node next;
    // Constructor
    Node(int data) {
        this.data = data;
        this.next = null;
    }
}
// Stack class that implements stack operations using a linked list
class Stack {
    private Node top;
    private int size;
    // Constructor
    public Stack() {
        this.top = null;
        this.size = 0;
    }
    // Push operation
    public void push(int data) {
        Node newNode = new Node(data);
        newNode.next = top;  // Point new node to current top
        top = newNode;       // Make new node the top
        size++;
        System.out.println(data + " pushed to stack");
    }
    // Pop operation
    public void pop() {
```

```
    if (isEmpty()) {
        System.out.println("Stack is empty. Nothing to pop.");
        return;
    }
    int popped = top.data;
    top = top.next;      // Remove top node
    size--;
    System.out.println(popped + " popped from stack");
}
// Peek operation
public void peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty. Nothing to peek.");
        return;
    }
    System.out.println("Top element is: " + top.data);
}
// Check if stack is empty
public boolean isEmpty() {
    return top == null;
}
// Get size of stack
public void getSize() {
    System.out.println("Size of stack: " + size);
}
// Display all elements of the stack
public void display() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
        return;
```

```
        }
        Node temp = top;
        System.out.print("Stack elements: ");
        while (temp != null) {
            System.out.print(temp.data + " ");
            temp = temp.next;
        }
        System.out.println();
    }
}
public class StackUsingLinkedList {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        Stack stack = new Stack();
        int choice;
        // Menu-driven program
        do {
            System.out.println("\nStack Operations:");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Peek");
            System.out.println("4. Check if Stack is Empty");
            System.out.println("5. Get Stack Size");
            System.out.println("6. Display Stack");
            System.out.println("7. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    // Push operation
```

```
            System.out.print("Enter element to push: ");
            int data = scanner.nextInt();
            stack.push(data);
            break;
        case 2:
            // Pop operation
            stack.pop();
            break;
        case 3:
            // Peek operation
            stack.peek();
            break;
        case 4:
            // Check if stack is empty
            if (stack.isEmpty()) {
                System.out.println("Stack is empty.");
            } else {
                System.out.println("Stack is not empty.");
            }
            break;
        case 5:
            // Get size of stack
            stack.getSize();
            break;
        case 6:
            // Display stack
            stack.display();
            break;
        case 7:
            System.out.println("Exiting...");
```

```
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    } while (choice != 7);
    scanner.close();
  }
}
```

How the Code Works:

1. Node Class: This class represents a node in the linked list, containing the data and a reference to the next node.

2. Stack Class: This class implements stack operations using a linked list. The stack has:

   - A reference to the top node (`top`).

   - A counter for the stack size (`size`).

   - Methods for push, pop, peek, isEmpty, getSize, and display.

3. Menu-Driven Main Program: The user can interact with the stack using a menu. Based on the user's input, it performs the corresponding operation on the stack.

Operations:

- Push: Adds an element to the top of the stack.

- Pop: Removes the top element of the stack.

- Peek: Displays the top element without removing it.

- isEmpty: Checks if the stack is empty.

- Size: Returns the current size of the stack.

- Display: Shows all the elements in the stack from top to bottom.

Example Output:

Stack Operations:

1. Push
2. Pop
3. Peek
4. Check if Stack is Empty
5. Get Stack Size
6. Display Stack
7. Exit

Enter your choice: 1

Enter element to push: 10

10 pushed to stack

Stack Operations:

1. Push
2. Pop
3. Peek
4. Check if Stack is Empty
5. Get Stack Size
6. Display Stack
7. Exit

Enter your choice: 1

Enter element to push: 20

20 pushed to stack

Stack Operations:

1. Push
2. Pop
3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 3

Top element is: 20

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 2

20 popped from stack

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 6

Stack elements: 10

Key Points:

1. Push and Pop operations are \(O(1)\) as we only interact with the top node.

2. Peek, isEmpty, getSize operations are also \(O(1)\).

3. Stack Implementation using a linked list ensures that no space is wasted, unlike array-based implementations, where resizing might be required.

Stack Data Structure with Array Implementation

A stack is a linear data structure that follows the Last In, First Out (LIFO) principle. The element that is added last is the first one to be removed. It can be visualized as a collection of elements where you can only insert or remove items from the top (the last added item).

Operations in Stack:

1. Push: Adds an element to the top of the stack.

2. Pop: Removes the top element from the stack.

3. Peek (Top): Returns the top element without removing it.

4. isEmpty: Checks if the stack is empty.

5. Size: Returns the size of the stack.

Time Complexity of Stack Operations:

- Push Operation: \(O(1)\)

- Adding an element to the top of the stack is done in constant time.

- Pop Operation: \(O(1)\)

- Removing the top element from the stack is done in constant time.

- Peek Operation: \(O(1)\)

- Returning the top element without removing it is done in constant time.

isEmpty Operation: \(O(1)\)

- Checking if the stack is empty takes constant time.

Size Operation: \(O(1)\)

- The size can be maintained and checked in constant time.

Java Program: Stack Using Array (Menu-Driven)

Here's a Java program that implements a stack using an array. The program provides a menu-driven interface that allows the user to perform operations such as push, pop, peek, check if the stack is empty, and get the stack size.

```
import java.util.Scanner;

// Stack class using an array
class Stack {
    private int maxSize;
    private int top;
    private int[] stack;
    // Constructor to initialize the stack with a maximum size
    public Stack(int size) {
        maxSize = size;
        stack = new int[maxSize];
        top = -1;  // Stack is empty when top is -1
    }
    // Push operation
    public void push(int data) {
        if (top == maxSize - 1) {
            System.out.println("Stack Overflow. Cannot push " + data);
        } else {
            stack[++top] = data;
            System.out.println(data + " pushed to stack");
        }
    }
    // Pop operation
    public void pop() {
        if (isEmpty()) {
            System.out.println("Stack Underflow. Nothing to pop.");
        } else {
            int popped = stack[top--];
            System.out.println(popped + " popped from stack");
```

```
    }
}
// Peek operation
public void peek() {
    if (isEmpty()) {
        System.out.println("Stack is empty. Nothing to peek.");
    } else {
        System.out.println("Top element is: " + stack[top]);
    }
}
// Check if stack is empty
public boolean isEmpty() {
    return top == -1;
}
// Get the size of the stack
public void getSize() {
    System.out.println("Size of stack: " + (top + 1));
}
// Display all elements of the stack
public void display() {
    if (isEmpty()) {
        System.out.println("Stack is empty.");
    } else {
        System.out.print("Stack elements: ");
        for (int i = 0; i <= top; i++) {
            System.out.print(stack[i] + " ");
        }
        System.out.println();
    }
}
```

```
}
public class StackUsingArray {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
          // Get the stack size from the user
        System.out.print("Enter the maximum size of the stack: ");
        int size = scanner.nextInt();
        Stack stack = new Stack(size);
        int choice;
        // Menu-driven program
        do {
            System.out.println("\nStack Operations:");
            System.out.println("1. Push");
            System.out.println("2. Pop");
            System.out.println("3. Peek");
            System.out.println("4. Check if Stack is Empty");
            System.out.println("5. Get Stack Size");
            System.out.println("6. Display Stack");
            System.out.println("7. Exit");
            System.out.print("Enter your choice: ");
            choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    // Push operation
                    System.out.print("Enter element to push: ");
                    int data = scanner.nextInt();
                    stack.push(data);
                    break;
                case 2:
                    // Pop operation
```

```
                stack.pop();
                break;
            case 3:
                // Peek operation
                stack.peek();
                break;
            case 4:
                // Check if stack is empty
                if (stack.isEmpty()) {
                    System.out.println("Stack is empty.");
                } else {
                    System.out.println("Stack is not empty.");
                }
                break;
            case 5:
                // Get size of stack
                stack.getSize();
                break;
            case 6:
                // Display stack
                stack.display();
                break;
            case 7:
                System.out.println("Exiting...");
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    } while (choice != 7);
    scanner.close();
```

```
    }
}
```

How the Code Works:

1. Stack Class:

- The stack is represented using an array (`stack[]`).

- The top variable keeps track of the index of the top element. Initially, it is set to -1 to indicate that the stack is empty.

- The `push()`, `pop()`, `peek()`, `isEmpty()`, `getSize()`, and `display()` methods handle various operations on the stack.

2. Menu-Driven Program:

- The user is prompted to choose an operation from a menu.

- The user can input elements to push onto the stack, pop elements, check the top element, check if the stack is empty, and display all elements in the stack.

Example Output:

Input:

Enter the maximum size of the stack: 5

Menu Options:

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 1

Enter element to push: 10

10 pushed to stack

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 1

Enter element to push: 20

20 pushed to stack

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 3

Top element is: 20

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 2

20 popped from stack

Stack Operations:

1. Push

2. Pop

3. Peek

4. Check if Stack is Empty

5. Get Stack Size

6. Display Stack

7. Exit

Enter your choice: 6

Stack elements: 10

Key Points:

- Push and Pop operations are \(O(1)\) since they only involve adding or removing the element at the top.

- Peek, isEmpty, getSize, and display operations are also \(O(1)\), \(O(1)\), \(O(1)\), and \(O(n)\) respectively, where \(n\) is the number of elements in the stack.

- The stack is implemented using a fixed-size array. If the array is full, a Stack Overflow error is thrown when pushing an element. If the stack is empty, a Stack Underflow error is thrown when popping an element.

To implement Postfix evaluation

Applications of Stack

Stacks are widely used in computer science for various purposes, including:

1. Expression Evaluation and Conversion:
   - Converting infix to postfix or prefix expressions.
   - Evaluating postfix or prefix expressions.

2. Backtracking:

- For example, in mazes, puzzles, or navigating file systems.

3. Function Call Management:

- Used in recursion to store function calls in a call stack.

4. Undo Mechanisms:

- In text editors or applications where multiple undo levels are implemented.

5. Parsing:

- For parsing expressions, program compilation, or processing HTML/XML tags.

6. Tree Traversals:

- Non-recursive traversal of trees (e.g., inorder, preorder, postorder).

Prefix, Infix, and Postfix Notation

1. Infix Notation:

- Operators are placed between operands.
- Example: \(A + B \)

2. Prefix Notation (Polish Notation):

- Operators are placed before operands.
- Example: \(+ A B \)

3. Postfix Notation (Reverse Polish Notation):

- Operators are placed after operands.
- Example: \(A B + \)

Postfix Expression Evaluation

Postfix expressions do not require parentheses because the order of operations is unambiguous. The steps for evaluating a postfix expression using a stack are:

1. Traverse the expression from left to right.
2. Push operands onto the stack.
3. When an operator is encountered:
   - Pop two elements from the stack.
   - Apply the operator.
   - Push the result back onto the stack.
4. At the end, the stack will contain one element: the result.

Java Program: Postfix Expression Evaluation

java

```
import java.util.Scanner;
import java.util.Stack;
public class PostfixEvaluation {
    // Method to evaluate a postfix expression
    public static int evaluatePostfix(String expression) {
        Stack<Integer> stack = new Stack<>();
        // Traverse the expression
        for (int i = 0; i < expression.length(); i++) {
            char ch = expression.charAt(i);
            // If the character is a digit, push it to the stack
            if (Character.isDigit(ch)) {
                stack.push(ch - '0'); // Convert character to integer
            }
            // If the character is an operator, pop two elements, apply the operator, and push the result
```

```
        else {
            int operand2 = stack.pop(); // Second operand
            int operand1 = stack.pop(); // First operand

            switch (ch) {
                case '+':
                    stack.push(operand1 + operand2);
                    break;
                case '-':
                    stack.push(operand1 - operand2);
                    break;
                case '':
                    stack.push(operand1  operand2);
                    break;
                case '/':
                    stack.push(operand1 / operand2);
                    break;
            }
        }
    }
    // The result is the only element left in the stack
    return stack.pop();
}
public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
    System.out.println("Enter a postfix expression (e.g., 2354+): ");
    String expression = scanner.nextLine();
    int result = evaluatePostfix(expression);
    System.out.println("The result of the postfix expression is: " + result);
```

```
        scanner.close();
    }
}
```

How the Program Works

1. Input:

   The program takes a postfix expression as input (e.g., `2354+`).

2. Stack Operations:

   - If a character is a digit, it is pushed onto the stack.

   - If a character is an operator, two operands are popped from the stack, and the operation is applied. The result is then pushed back onto the stack.

3. Result:

   At the end of the expression, the stack contains only one element, which is the result.

Example Execution

Input:

2354+

Step-by-Step Execution:

Character	Stack After Execution	Operation
2	[2]	Push 2
3	[2, 3]	Push 3
	[6]	Pop 2, 3; Push 6
5	[6, 5]	Push 5
4	[6, 5, 4]	Push 4
	[6, 20]	Pop 5, 4; Push 20
+	[26]	Pop 6, 20; Push 26

Output:

The result of the postfix expression is: 26

Key Points:

1. Time Complexity:

- Traversal of Expression: (O(n)), where (n) is the length of the expression.

- Stack operations (push pop) are (O(1)).

2. **Space Complexity:**

- Depends on the stack size, which in the worst case is proportional to the number of operands (O(n)).

Balancing Parentheses Using a Stack

Balancing parentheses is a common problem where we check whether the given string containing different types of brackets ({}, [], ()) is properly nested and balanced. A **stack** is an ideal data structure for solving this problem because of its LIFO (Last In, First Out) nature.

Steps to Check Parentheses Balance

1. **Traverse the Expression**:
    - Read each character of the string one at a time.
2. **Push Open Brackets**:
    - If the character is an opening bracket ((, [, {), push it onto the stack.
3. **Match Closing Brackets**:
    - If the character is a closing bracket (),], }):
        - Check if the stack is empty (indicating an unmatched closing bracket).
        - Pop the top of the stack and check if it matches the closing bracket.
4. **Final Check**:
    - After traversing the string, the stack should be empty. If not, it indicates unmatched opening brackets.

Algorithm

1. Create an empty stack.
2. Traverse each character of the string:
    - If it is an opening bracket, push it onto the stack.
    - If it is a closing bracket, check:
        - If the stack is empty, the string is unbalanced.
        - Pop the stack and check if the popped element matches the closing bracket. If not, the string is unbalanced.
3. After traversing the string, check if the stack is empty. If not, the string is unbalanced.

Java Code

```
import java.util.Stack;
public class ParenthesisBalancing {
   // Function to check if parentheses are balanced
   public static boolean isBalanced(String expr) {
      Stack<Character> stack = new Stack<>();
      // Traverse the string
      for (char ch : expr.toCharArray()) {
         // Push open brackets onto the stack
         if (ch == '(' || ch == '[' || ch == '{') {
            stack.push(ch);
         }
         // Check for closing brackets
         else if (ch == ')' || ch == ']' || ch == '}') {
            // If stack is empty, it's unbalanced
            if (stack.isEmpty()) {
               return false;
            }
            // Pop the top element and check if it matches
            char top = stack.pop();
```

```
            if (!isMatchingPair(top, ch)) {
                return false;
            }
        }
    }
    // If the stack is not empty, it's unbalanced
    return stack.isEmpty();
}
// Helper function to check if two brackets are matching pairs
private static boolean isMatchingPair(char open, char close) {
    return (open == '(' && close == ')') ||
           (open == '[' && close == ']') ||
           (open == '{' && close == '}');
}
// Main method to test the function
public static void main(String[] args) {
    String expr = "{[()]}";
    if (isBalanced(expr)) {
        System.out.println("The expression is balanced.");
    } else {
        System.out.println("The expression is not balanced.");
    }
}
```

Example Execution

Input: {[()]}

1. Read {: Push onto stack → Stack: [{.
2. Read [: Push onto stack → Stack: [{[.
3. Read (: Push onto stack → Stack: [{[(.
4. Read): Pop and match with (→ Stack: [{[.

5. Read]: Pop and match with [→ Stack: [{.
6. Read }: Pop and match with { → Stack: [].
7. **Final Check**: Stack is empty → Balanced.

Output: The expression is balanced.

Time and Space Complexity

1. **Time Complexity**:
    - Traversing the string takes O(n)O(n), where nn is the length of the string.
    - Each stack operation (push and pop) takes O(1)O(1).
    - Overall: O(n)O(n).
2. **Space Complexity**:
    - The stack can hold at most n/2n/2 elements (if all are opening brackets).
    - Space complexity: O(n)O(n).

Applications

1. **Compiler Design**:
    - Used to validate syntax (matching parentheses, brackets, etc.).
2. **Expression Parsing**:
    - Ensures proper nesting in mathematical and logical expressions.
3. **Code Editors**:
    - Real-time checking for balanced parentheses in IDEs or text editors.
4. **XML/HTML Validation**:
    - Verifies proper nesting and closure of tags.

Queue Data Structure

A **Queue** is a linear data structure that follows the **FIFO (First In, First Out)** principle. It is used to process elements in the order they arrive.

Types of Queues

1. **Simple Queue**:
    - Basic FIFO queue where insertion happens at the rear and deletion happens at the front.
2. **Circular Queue**:
    - The rear connects back to the front to form a circular structure, making better use of space.
3. **Priority Queue**:
    - Elements are dequeued based on priority rather than arrival time.
4. **Deque (Double-Ended Queue)**:
    - Insertion and deletion can occur at both ends (front and rear).

Circular Queue

A **Circular Queue** overcomes the limitations of the Simple Queue where unused spaces may arise after multiple dequeue operations. In a Circular Queue:

- The last position is connected to the first position.
- The queue operates in a circular fashion.

Algorithm for Circular Queue

Operations:

1. **Enqueue (Insertion)**:
    - Check if the queue is full: (rear + 1) % size == front.
    - If not full:
        - Insert the element at rear.
        - Update rear = (rear + 1) % size.
2. **Dequeue (Deletion)**:
    - Check if the queue is empty: front == rear.
    - If not empty:

- Remove the element at front.
- Update front = (front + 1) % size.

3. **Peek**:
    - Return the element at front if the queue is not empty.
4. **IsEmpty**:
    - Return true if front == rear.
5. **IsFull**:
    - Return true if (rear + 1) % size == front.

Java Implementation of Circular Queue with User Input

Below is the implementation where the queue size and operations are determined by user input:

```
import java.util.Scanner;
public class CircularQueueWithUserInput {
    private int[] queue; // Array to store the queue elements
    private int front;   // Points to the front element
    private int rear;    // Points to the next insertion position
    private int size;    // Size of the queue
    // Constructor to initialize the queue
    public CircularQueueWithUserInput(int size) {
        this.size = size;
        this.queue = new int[size];
        this.front = 0;
        this.rear = 0;
    }
    // Check if the queue is full
    public boolean isFull() {
        return (rear + 1) % size == front;
    }

    // Check if the queue is empty
```

```
public boolean isEmpty() {
    return front == rear;
}
// Add an element to the queue
public void enqueue(int element) {
    if (isFull()) {
        System.out.println("Queue is full. Cannot enqueue " + element);
        return;
    }
    queue[rear] = element;
    rear = (rear + 1) % size;
    System.out.println("Enqueued: " + element);
}
// Remove and return the front element from the queue
public int dequeue() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Cannot dequeue.");
        return -1;
    }
    int element = queue[front];
    front = (front + 1) % size;
    System.out.println("Dequeued: " + element);
    return element;
}
// Peek the front element without removing it
public int peek() {
    if (isEmpty()) {
        System.out.println("Queue is empty. Cannot peek.");
        return -1;
    }
```

```
        return queue[front];
    }
    // Display the queue elements
    public void display() {
        if (isEmpty()) {
            System.out.println("Queue is empty.");
            return;
        }
        System.out.print("Queue: ");
        int i = front;
        while (i != rear) {
            System.out.print(queue[i] + " ");
            i = (i + 1) % size;
        }
        System.out.println();
    }
    // Main method to test the Circular Queue with user input
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the size of the queue: ");
        int size = scanner.nextInt();
        CircularQueueWithUserInput cq = new CircularQueueWithUserInput(size
+ 1); // +1 to differentiate full vs empty

        while (true) {
            System.out.println("\nChoose an operation:");
            System.out.println("1. Enqueue");
            System.out.println("2. Dequeue");
            System.out.println("3. Peek");
            System.out.println("4. Display");
            System.out.println("5. Exit");
```

```
        System.out.print("Enter your choice: ");
        int choice = scanner.nextInt();

        switch (choice) {
            case 1:
                System.out.print("Enter the element to enqueue: ");
                int element = scanner.nextInt();
                cq.enqueue(element);
                break;
            case 2:
                cq.dequeue();
                break;
            case 3:
                int frontElement = cq.peek();
                if (frontElement != -1) {
                    System.out.println("Front Element: " + frontElement);
                }
                break;
            case 4:
                cq.display();
                break;
            case 5:
                System.out.println("Exiting...");
                scanner.close();
                return;
            default:
                System.out.println("Invalid choice. Please try again.");
        }
    }
}
```

}

Step-by-Step Execution

1. **Input**: Enter the size of the queue: 5.
   Queue size is initialized to 6 (5 + 1 for circular behavior).
2. **Operation 1 (Enqueue 10)**:
   - rear = 0, insert 10, update rear = 1.
   - Queue: [10, _, _, _, _, _].
3. **Operation 2 (Enqueue 20)**:
   - rear = 1, insert 20, update rear = 2.
   - Queue: [10, 20, _, _, _, _].
4. **Operation 3 (Dequeue)**:
   - front = 0, remove 10, update front = 1.
   - Queue: [10, 20, _, _, _, _] (logical state).
5. **Operation 4 (Display)**:
   - Output: 20.
6. **Operation 5 (Peek)**:
   - Front element: 20.
7. **Operation 6 (Enqueue 30)**:
   - rear = 2, insert 30, update rear = 3.
   - Queue: [10, 20, 30, _, _, _].

Applications of Circular Queue

1. **CPU Scheduling**: Round-robin scheduling algorithms.
2. **Traffic Management**: Managing circular queues in traffic lights.
3. **Buffer Management**: Multimedia streaming and buffering.
4. **Memory Allocation**: Efficient memory management in embedded systems.

Complexity Analysis

Operation	Time Complexity	Space Complexity
Enqueue	O(1)	O(n)
Dequeue	O(1)	O(n)
Peek	O(1)	O(n)

Priority Queue

A **Priority Queue** is a specialized data structure where each element is associated with a priority, and elements with higher priority are served before those with lower priority. If two elements have the same priority, they are served according to their order in the queue (depending on implementation).

Key Characteristics of a Priority Queue

1. **Prioritized Processing**:
    - Elements are dequeued in order of priority.
2. **Order of Insertion**:
    - If priorities are equal, insertion order may determine processing order.
3. **Heap-Based Implementation**:
    - Commonly implemented using a heap for efficient insertion and deletion.

Algorithm for Priority Queue

Operations:

1. **Enqueue**:
    - Insert the element based on its priority into the queue.
    - Maintain the priority order in the queue.
2. **Dequeue**:
    - Remove and return the element with the highest priority.
3. **Peek**:
    - Return the element with the highest priority without removing it.

Java Implementation of Priority Queue with User Input

Below is a user-interactive implementation of a priority queue using Java. The queue stores elements as pairs of value and priority.

Code Implementation

```
import java.util.PriorityQueue;
import java.util.Scanner;
class Element implements Comparable<Element> {
    int value;
    int priority;
    // Constructor
    public Element(int value, int priority) {
        this.value = value;
        this.priority = priority;
    }
    // Compare elements based on priority (higher priority comes first)
    @Override
    public int compareTo(Element other) {
        return Integer.compare(other.priority, this.priority); // Max-Heap style
    }
    @Override
    public String toString() {
        return "Value: " + value + ", Priority: " + priority;
    }
}
public class UserInputPriorityQueue {

    public static void main(String[] args) {
        PriorityQueue<Element> priorityQueue = new PriorityQueue<>();
        Scanner scanner = new Scanner(System.in);
        while (true) {
```

```
            System.out.println("\nChoose an operation:");
            System.out.println("1. Enqueue (Add Element)");
            System.out.println("2. Dequeue (Remove Highest Priority)");
            System.out.println("3. Peek (View Highest Priority)");
            System.out.println("4. Display All Elements");
            System.out.println("5. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();

            switch (choice) {
                case 1:
                    System.out.print("Enter value: ");
                    int value = scanner.nextInt();
                    System.out.print("Enter priority: ");
                    int priority = scanner.nextInt();
                    priorityQueue.add(new Element(value, priority));
                    System.out.println("Enqueued: " + value + " with priority " +
priority);
                    break;
                case 2:
                    if (priorityQueue.isEmpty()) {
                        System.out.println("Queue is empty. Cannot dequeue.");
                    } else {
                        Element removed = priorityQueue.poll();
                        System.out.println("Dequeued: " + removed);
                    }
                    break;
                case 3:
                    if (priorityQueue.isEmpty()) {
                        System.out.println("Queue is empty. Cannot peek.");
                    } else {
```

```
                System.out.println("Highest Priority Element: " +
priorityQueue.peek());
            }
            break;
        case 4:
            if (priorityQueue.isEmpty()) {
                System.out.println("Queue is empty.");
            } else {
                System.out.println("All Elements in Priority Queue:");
                for (Element e : priorityQueue) {
                    System.out.println(e);
                }
            }
            break;
        case 5:
            System.out.println("Exiting...");
            scanner.close();
            return;
        default:
            System.out.println("Invalid choice. Please try again.");
      }
    }
  }
}
```

Step-by-Step Execution

1. **Initialization**:
    - The queue is empty.
2. **Enqueue Operations**:
    - **Input**: Value = 10, Priority = 3.
        - Add (10, 3) to the queue.

        - Queue: [(10, 3)].
    - **Input**: Value = 20, Priority = 5.
        - Add (20, 5) to the queue.
        - Queue: [(20, 5), (10, 3)].
    - **Input**: Value = 30, Priority = 4.
        - Add (30, 4) to the queue.
        - Queue: [(20, 5), (10, 3), (30, 4)].
3. **Peek Operation**:
    - **Output**: Highest Priority Element: (20, 5).
4. **Dequeue Operation**:
    - Remove (20, 5) (highest priority).
    - Queue: [(30, 4), (10, 3)].
5. **Display Operation**:
    - **Output**:
    - Value: 30, Priority: 4
    - Value: 10, Priority: 3

Applications of Priority Queue

1. **CPU Scheduling**:
    - Used in operating systems to manage tasks based on priority.
2. **Graph Algorithms**:
    - Used in Dijkstra's and Prim's algorithms for finding shortest paths and minimum spanning trees.
3. **Event Simulation**:
    - Simulating events where events are processed in priority order.
4. **Network Routing**:
    - Managing packets in routers based on priority.
5. **Data Compression**:
    - Used in Huffman encoding to build trees with prioritized nodes.

Complexity Analysis

Operation	Time Complexity	Space Complexity
Enqueue	O(log n)	O(n)
Dequeue	O(log n)	O(n)
Peek	O(1)	O(n)

Implementation of all types of linked list

Linked List Overview

A **Linked List** is a dynamic data structure consisting of nodes. Each node contains:

1. **Data**: The actual value.
2. **Pointer/Reference**: The address of the next (and sometimes the previous) node.

Types of Linked Lists

1. **Singly Linked List (SLL)**:
    - Each node points to the next node.
    - Traversal is one-directional.
2. **Doubly Linked List (DLL)**:
    - Each node points to both the next and the previous nodes.
    - Traversal can be bidirectional.
3. **Circular Linked List (CLL)**:
    - The last node points to the first node, forming a circle.
    - Can be implemented as singly or doubly linked.

Operations on Linked List

1. **Insertion**:
    - At the beginning, end, or specific position.
2. **Deletion**:
    - From the beginning, end, or specific position.

3. **Traversal**:
    - Visit each node to display its value.
4. **Search**:
    - Find a specific value in the list.

Applications of Linked Lists

1. **Dynamic Memory Management**: Efficient for data with frequent insertions/deletions.
2. **Implementation of Data Structures**: Stacks, queues, and graphs.
3. **Navigation Systems**: Music playlists, undo-redo functionalities.

Complexity Analysis

Operation	**Singly Linked List**	**Doubly Linked List**	**Circular Linked List**
Traversal	O(n)	O(n)	O(n)
Insertion (Head)	O(1)	O(1)	O(1)
Insertion (End)	O(n)	O(1) (if tail)	O(1) (if tail)
Deletion (Head)	O(1)	O(1)	O(1)
Deletion (End)	O(n)	O(1) (if tail)	O(1) (if tail)

Java Implementation with User Input

1. Singly Linked List

```
import java.util.Scanner;
class SinglyLinkedList {
    // Node class
    static class Node {
        int data;
        Node next;
        Node(int data) {
```

```
        this.data = data;
        this.next = null;
    }
}
private Node head;
// Insert at the end
public void insert(int data) {
    Node newNode = new Node(data);
    if (head == null) {
        head = newNode;
    } else {
        Node temp = head;
        while (temp.next != null) {
            temp = temp.next;
        }
        temp.next = newNode;
    }
    System.out.println("Inserted: " + data);
}
// Delete the first occurrence of a value
public void delete(int data) {
    if (head == null) {
        System.out.println("List is empty. Cannot delete.");
        return;
    }
    if (head.data == data) {
        head = head.next;
        System.out.println("Deleted: " + data);
        return;
    }
```

```
    Node temp = head;
    while (temp.next != null && temp.next.data != data) {
        temp = temp.next;
    }
    if (temp.next == null) {
        System.out.println("Element not found.");
    } else {
        temp.next = temp.next.next;
        System.out.println("Deleted: " + data);
    }
}
// Traverse and display
public void display() {
    if (head == null) {
        System.out.println("List is empty.");
        return;
    }
    System.out.print("Singly Linked List: ");
    Node temp = head;
    while (temp != null) {
        System.out.print(temp.data + " -> ");
        temp = temp.next;
    }
    System.out.println("null");
}
public static void main(String[] args) {
    SinglyLinkedList sll = new SinglyLinkedList();
    Scanner scanner = new Scanner(System.in);
    while (true) {
        System.out.println("\n1. Insert");
```

```
            System.out.println("2. Delete");
            System.out.println("3. Display");
            System.out.println("4. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter data to insert: ");
                    int data = scanner.nextInt();
                    sll.insert(data);
                    break;
                case 2:
                    System.out.print("Enter data to delete: ");
                    int deleteData = scanner.nextInt();
                    sll.delete(deleteData);
                    break;
                case 3:
                    sll.display();
                    break;
                case 4:
                    scanner.close();
                    System.out.println("Exiting...");
                    return;
                default:
                    System.out.println("Invalid choice. Try again.");
            }
        }
    }
}
```

2. Doubly Linked List

```
import java.util.Scanner;
class DoublyLinkedList {
    static class Node {
        int data;
        Node prev, next;
        Node(int data) {
            this.data = data;
            this.prev = this.next = null;
        }
    }
    private Node head;
    // Insert at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (head == null) {
            head = newNode;
        } else {
            Node temp = head;
            while (temp.next != null) {
                temp = temp.next;
            }
            temp.next = newNode;
            newNode.prev = temp;
        }
        System.out.println("Inserted: " + data);
    }
    // Traverse and display
    public void display() {
        if (head == null) {
```

```
            System.out.println("List is empty.");
            return;
        }
        System.out.print("Doubly Linked List: ");
        Node temp = head;
        while (temp != null) {
            System.out.print(temp.data + " <-> ");
            temp = temp.next;
        }
        System.out.println("null");
    }
    public static void main(String[] args) {
        DoublyLinkedList dll = new DoublyLinkedList();
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("\n1. Insert");
            System.out.println("2. Display");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1:
                    System.out.print("Enter data to insert: ");
                    int data = scanner.nextInt();
                    dll.insert(data);
                    break;
                case 2:
                    dll.display();
                    break;
                case 3:
```

```
                scanner.close();
                System.out.println("Exiting...");
                return;
            default:
                System.out.println("Invalid choice. Try again.");
        }
    }
  }
}
```

3. Circular Linked List

```
import java.util.Scanner;
class CircularLinkedList {
    static class Node {
        int data;
        Node next;
        Node(int data) {
            this.data = data;
            this.next = null;
        }
    }
    private Node last;
    // Insert at the end
    public void insert(int data) {
        Node newNode = new Node(data);
        if (last == null) {
            last = newNode;
            last.next = last;
        } else {
```

```
            newNode.next = last.next;
            last.next = newNode;
            last = newNode;
        }
        System.out.println("Inserted: " + data);
    }

    // Traverse and display
    public void display() {
        if (last == null) {
            System.out.println("List is empty.");
            return;
        }
        System.out.print("Circular Linked List: ");
        Node temp = last.next;
        do {
            System.out.print(temp.data + " -> ");
            temp = temp.next;
        } while (temp != last.next);
        System.out.println("(back to head)");
    }
    public static void main(String[] args) {
        CircularLinkedList cll = new CircularLinkedList();
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("\n1. Insert");
            System.out.println("2. Display");
            System.out.println("3. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
```

```
            switch (choice) {
                case 1:
                    System.out.print("Enter data to insert: ");
                    int data = scanner.nextInt();
                    cll.insert(data);
                    break;
                case 2:
                    cll.display();
                    break;
                case 3:
                    scanner.close();
                    System.out.println("Exiting...");
                    return;
                default:
                    System.out.println("Invalid choice. Try again.");
            }
        }
    }
}
```

8. Demonstrate application of linked list
   a) To implement Polynomial Addition

Polynomial addition using linked lists in Java involves creating a data structure to represent terms of the polynomial, implementing methods to traverse and add corresponding terms, and displaying the result

Step-by-Step Implementation

1. **Create a Node Class** Each node represents a term in the polynomial, containing:
   - The coefficient.
   - The exponent.
   - A reference to the next node.

2. **Create a LinkedList Class** The class manages the polynomial and provides methods to:
    - Insert terms into the list.
    - Add two polynomials.
    - Display the polynomial.
3. **Take User Input** Allow the user to enter the coefficients and exponents for two polynomials.
4. **Add Polynomials** Traverse both linked lists, comparing exponents and adding coefficients of terms with matching exponents.
5. **Output the Result** Display the resulting polynomial after addition.

Code

```
import java.util.Scanner;
class Node {
    int coefficient, exponent;
    Node next;
    Node(int coefficient, int exponent) {
        this.coefficient = coefficient;
        this.exponent = exponent;
        this.next = null;
    }
}
class Polynomial {
    Node head;
    // Insert a term in the polynomial in sorted order by exponent
    public void insertTerm(int coefficient, int exponent) {
        Node newNode = new Node(coefficient, exponent);

        if (head == null || head.exponent < exponent) {
            newNode.next = head;
            head = newNode;
```

```
    } else {
        Node current = head;
        while (current.next != null && current.next.exponent > exponent) {
            current = current.next;
        }
        newNode.next = current.next;
        current.next = newNode;
    }
}
// Display the polynomial
public void display() {
    if (head == null) {
        System.out.println("0");
        return;
    }
    Node current = head;
    while (current != null) {
        System.out.print(current.coefficient + "x^" + current.exponent);
        if (current.next != null) System.out.print(" + ");
        current = current.next;
    }
    System.out.println();
}
// Add two polynomials
public static Polynomial add(Polynomial p1, Polynomial p2) {
    Polynomial result = new Polynomial();
    Node t1 = p1.head, t2 = p2.head;

    while (t1 != null && t2 != null) {
        if (t1.exponent == t2.exponent) {
```

```
            int sumCoeff = t1.coefficient + t2.coefficient;
            if (sumCoeff != 0) {
                result.insertTerm(sumCoeff, t1.exponent);
            }
            t1 = t1.next;
            t2 = t2.next;
        } else if (t1.exponent > t2.exponent) {
            result.insertTerm(t1.coefficient, t1.exponent);
            t1 = t1.next;
        } else {
            result.insertTerm(t2.coefficient, t2.exponent);
            t2 = t2.next;
        }
    }
    // Add remaining terms
    while (t1 != null) {
        result.insertTerm(t1.coefficient, t1.exponent);
        t1 = t1.next;
    }
    while (t2 != null) {
        result.insertTerm(t2.coefficient, t2.exponent);
        t2 = t2.next;
    }

    return result;
  }
}
public class PolynomialAddition {
  public static void main(String[] args) {
    Scanner scanner = new Scanner(System.in);
```

```
        // Create two polynomials
        Polynomial p1 = new Polynomial();
        Polynomial p2 = new Polynomial();
        // Input first polynomial
        System.out.println("Enter the number of terms for the first polynomial:");
        int n1 = scanner.nextInt();
        System.out.println("Enter terms (coefficient and exponent):");
        for (int i = 0; i < n1; i++) {
            int coeff = scanner.nextInt();
            int exp = scanner.nextInt();
            p1.insertTerm(coeff, exp);
        }
        // Input second polynomial
        System.out.println("Enter the number of terms for the second
polynomial:");
        int n2 = scanner.nextInt();
        System.out.println("Enter terms (coefficient and exponent):");
        for (int i = 0; i < n2; i++) {
            int coeff = scanner.nextInt();
            int exp = scanner.nextInt();
            p2.insertTerm(coeff, exp);
        }
        // Display input polynomials
        System.out.println("First Polynomial:");
        p1.display();
        System.out.println("Second Polynomial:");
        p2.display();
        // Add the polynomials
        Polynomial result = Polynomial.add(p1, p2);
       // Display result
        System.out.println("Resultant Polynomial after addition:");
```

```
        result.display();
    }
}
```

Step-by-Step Execution

1. **User Input**
   Example Input:
   - First Polynomial: 2x^3 + 3x^2
   - Second Polynomial: 4x^3 + 5x^1
     User enters:
2. Number of terms: 2
3. Coefficient and Exponent: 2 3
4. Coefficient and Exponent: 3 2
5. Number of terms: 2
6. Coefficient and Exponent: 4 3
7. Coefficient and Exponent: 5 1
8. **Insert Terms into Linked Lists**
   After insertion:
   - First Polynomial: 2x^3 -> 3x^2
   - Second Polynomial: 4x^3 -> 5x^1
9. **Addition Logic**
   - Compare exponents:
     - 2x^3 + 4x^3 = 6x^3
     - 3x^2 = 3x^2
     - 5x^1 = 5x^1
   - Resultant polynomial: 6x^3 + 3x^2 + 5x^1.
10. **Display Output**
11. First Polynomial: 2x^3 + 3x^2
12. Second Polynomial: 4x^3 + 5x^1
13. Resultant Polynomial: 6x^3 + 3x^2 + 5x^1

How the Code Works

1. **Insertion**: Terms are inserted in descending order of exponents to simplify addition.
2. **Addition**:
    - Traverse both lists simultaneously.
    - Add coefficients of terms with matching exponents.
    - Append terms with non-matching exponents to the result.
3. **Output**: Displays the formatted polynomial by traversing the result linked list.

9. Demonstrate application of linked list
b) To implement Sparse Matrix

Implementing a sparse matrix using a linked list in Java involves representing non-zero elements as nodes in a linked list. Each node stores the row index, column index, and value of the non-zero element.

Step-by-Step Implementation

1. **Understand Sparse Matrix Representation** A sparse matrix contains mostly zero values. Instead of storing the entire matrix, we store only the non-zero values along with their row and column indices.
2. **Create a Node Class** Each node represents a non-zero element and includes:
    - The row index.
    - The column index.
    - The value of the element.
    - A reference to the next node.
3. **Create a SparseMatrix Class** This class manages the linked list and provides methods to:
    - Add a non-zero element to the sparse matrix.
    - Display the matrix in a readable format.
    - Perform operations like addition or transpose (optional).
4. **Take User Input** Allow the user to specify matrix dimensions and enter non-zero elements.
5. **Display the Sparse Matrix** Show the non-zero elements in a readable format and optionally convert it back to the full matrix format for visualization.

Code

```
import java.util.Scanner;
class Node {
    int row, col, value;
    Node next;

    Node(int row, int col, int value) {
        this.row = row;
        this.col = col;
        this.value = value;
        this.next = null;
    }
}
class SparseMatrix {
    Node head;
    // Add a non-zero element to the sparse matrix
    public void addElement(int row, int col, int value) {
        Node newNode = new Node(row, col, value);
        if (head == null) {
            head = newNode;
        } else {
            Node current = head;
            while (current.next != null) {
                current = current.next;
            }
            current.next = newNode;
        }
    }
    // Display the sparse matrix as a list of non-zero elements
```

```
    public void displayAsList() {
        if (head == null) {
            System.out.println("The sparse matrix is empty.");
            return;
        }
        System.out.println("Row\tCol\tValue");
        Node current = head;
        while (current != null) {
            System.out.println(current.row + "\t" + current.col + "\t" +
current.value);
            current = current.next;
        }
    }
    // Display the full matrix for visualization
    public void displayAsMatrix(int rows, int cols) {
        int[][] matrix = new int[rows][cols];
        Node current = head;
        while (current != null) {
            matrix[current.row][current.col] = current.value;
            current = current.next;
        }
        System.out.println("Full Matrix Representation:");
        for (int i = 0; i < rows; i++) {
            for (int j = 0; j < cols; j++) {
                System.out.print(matrix[i][j] + " ");
            }
            System.out.println();
        }
    }
}
public class SparseMatrixImplementation {
```

```
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        // Input matrix dimensions
        System.out.println("Enter the number of rows and columns of the matrix:");
        int rows = scanner.nextInt();
        int cols = scanner.nextInt();
        // Create a sparse matrix
        SparseMatrix sparseMatrix = new SparseMatrix();
        // Input number of non-zero elements
        System.out.println("Enter the number of non-zero elements:");
        int nonZeroElements = scanner.nextInt();
        // Input non-zero elements
        System.out.println("Enter the row, column, and value for each non-zero element:");
        for (int i = 0; i < nonZeroElements; i++) {
            int row = scanner.nextInt();
            int col = scanner.nextInt();
            int value = scanner.nextInt();
            sparseMatrix.addElement(row, col, value);
        }
        // Display the sparse matrix
        System.out.println("\nSparse Matrix Representation (as list):");
        sparseMatrix.displayAsList();
        // Display the full matrix for visualization
        System.out.println("\nSparse Matrix Representation (as matrix):");
        sparseMatrix.displayAsMatrix(rows, cols);
    }
}
```

Step-by-Step Execution

1. **User Input** Example Input:

2. Enter the number of rows and columns of the matrix:

   4 5

3. Enter the number of non-zero elements:

   4

4. Enter the row, column, and value for each non-zero element:

   0 1 5

   1 2 8

   3 0 6

   3 4 7

5. **Adding Elements to the Linked List** After input, the linked list will look like:
   - Node(0, 1, 5) -> Node(1, 2, 8) -> Node(3, 0, 6) -> Node(3, 4, 7)
6. **Display as List**
7. Row Col Value

   0 1 5

   1 2 8

   3 0 6

   3 4 7

8. **Convert and Display as Full Matrix** Construct the full matrix from the linked list:
9. Full Matrix Representation:

   0 5 0 0 0

   0 0 8 0 0

   0 0 0 0 0

   6 0 0 0 7

How the Code Works

1. **Node Creation**
   - The Node class holds information about each non-zero element (row, column, value) and links to the next node.
2. **Adding Elements**
   - The addElement method inserts a new non-zero element at the end of the linked list.
3. **Display as List**

- The displayAsList method iterates through the linked list and prints each node's data.

4. **Convert to Full Matrix**
   - The displayAsMatrix method initializes a 2D array with zeros.
   - It populates the array by traversing the linked list and placing non-zero values at their respective positions.
5. **Efficiency**
   - Only non-zero elements are stored, saving memory compared to using a full matrix.

Output

For the above input, the output will be:

Sparse Matrix Representation (as list):

Row Col Value

0 1 5
1 2 8
3 0 6
3 4 7

Sparse Matrix Representation (as matrix):

0 5 0 0 0
0 0 8 0 0
0 0 0 0 0
6 0 0 0 7

9. Create and perform various operations on BST
a) Inserting node in BST
b) Deleting the node from BST
c) To find height of Tree
d) To perform Inorder
e) To perform Preorder
f) To perform Postorder
g) To find Maximum value of tree

A **Binary Search Tree (BST)** is a binary tree where each node satisfies the following properties:

1. **Key Property**:
   - The value of the left child of a node is **less than** the value of the node.

- The value of the right child of a node is **greater than** the value of the node.
- This rule applies recursively to all nodes in the tree.

2. **Binary Tree Structure**:
   - Each node can have at most two children: left and right.
3. **Inorder Traversal Property**:
   - An inorder traversal of a BST produces a sorted sequence of elements in ascending order.

Applications of Binary Search Tree

1. **Searching**:
   - Efficiently find elements in O(log⁡n)O(\log n) time in a balanced BST.
   - Example: Lookups in a symbol table, dictionary, or database index.
2. **Sorting**:
   - Inorder traversal produces sorted data.
3. **Dynamic Data Structure**:
   - Insertions and deletions are performed efficiently.
4. **Key Applications**:
   - **Database indexing**: BST can store and retrieve data efficiently.
   - **Priority queues**: BSTs like AVL trees or Red-Black trees support efficient insertion and deletion.
   - **Computer Networks**: Used for routing tables in network routers.
   - **Huffman Encoding**: BSTs form the basis for encoding algorithms.

Operations in Binary Search Tree

1. Insertion

- A new node is inserted such that the BST property is preserved.
- Steps:
  1. Start from the root.
  2. If the new value is smaller than the current node, move to the left subtree.

3. If the new value is larger, move to the right subtree.
4. Repeat until you reach a null position.

- Time Complexity: O(h), where h is the height of the tree.

2. Search

- Checks whether a given value exists in the BST.
- Steps:
  1. Start from the root.
  2. If the value matches the current node, return true.
  3. If the value is smaller, move to the left subtree.
  4. If the value is larger, move to the right subtree.
  5. Repeat until the value is found or the subtree is null.
- Time Complexity: O(h).

3. Deletion

- Deletes a node from the BST while maintaining the BST property.
- Cases:
  1. **Node has no children**: Simply remove the node.
  2. **Node has one child**: Replace the node with its child.
  3. **Node has two children**: Find the **inorder successor** (smallest value in the right subtree), replace the node with the successor, and delete the successor node.
- Time Complexity: O(h).

4. Inorder Traversal

- Visits nodes in ascending order: Left → Root → Right.
- Produces sorted output.
- Time Complexity: O(n).

5. Preorder Traversal

- Visits nodes in the order: Root → Left → Right.
- Useful for tree reconstruction.
- Time Complexity: O(n).

6. Postorder Traversal

- Visits nodes in the order: Left → Right → Root.

- Useful for deleting or freeing memory in trees.
- Time Complexity: O(n).

7. Finding the Minimum and Maximum

- **Minimum**: Traverse the leftmost subtree.
- **Maximum**: Traverse the rightmost subtree.
- Time Complexity: O(h).

8. Find Height

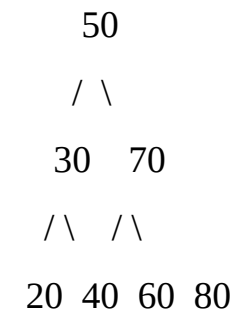
- The height of the BST is the longest path from the root to a leaf.
- Steps:
    1. Compute the height of the left and right subtrees recursively.
    2. Height = 1 + max(left height, right height).
- Time Complexity: O(n).

Example

Input:

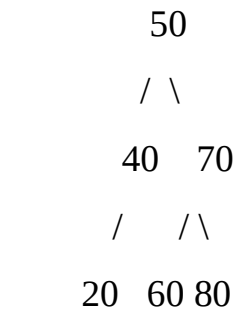
Insert the following values into a BST: **50, 30, 70, 20, 40, 60, 80**.

BST Structure:



Operations and Results:

1. **Search** for 60: Found.
2. **Delete** 30:



3. Inorder Traversal: 20, 40, 50, 60, 70, 80.

4. Preorder Traversal: 50, 40, 20, 70, 60, 80.

5. Postorder Traversal: 20, 40, 60, 80, 70, 50.

6. Height: 2.

7. Maximum Value: 80.

Code

```
import java.util.Scanner;
class Node {
    int data;
    Node left, right;
    Node(int data) {
        this.data = data;
        left = right = null;
    }
}
class BST {
    Node root;
    // a) Insert a node in BST
    public void insert(int value) {
        root = insertRec(root, value);
    }
    private Node insertRec(Node root, int value) {
        if (root == null) {
            root = new Node(value);
            return root;
        }
        if (value < root.data) {
            root.left = insertRec(root.left, value);
        } else if (value > root.data) {
```

```
        root.right = insertRec(root.right, value);
    }
    return root;
}
// b) Delete a node from BST
public void delete(int value) {
    root = deleteRec(root, value);
}
private Node deleteRec(Node root, int value) {
    if (root == null) return root;
    if (value < root.data) {
        root.left = deleteRec(root.left, value);
    } else if (value > root.data) {
        root.right = deleteRec(root.right, value);
    } else {
        // Node with one or no child
        if (root.left == null) return root.right;
        else if (root.right == null) return root.left;
        // Node with two children: get the inorder successor
        root.data = minValue(root.right);
        root.right = deleteRec(root.right, root.data);
    }
    return root;
}
private int minValue(Node root) {
    int minValue = root.data;
    while (root.left != null) {
        minValue = root.left.data;
        root = root.left;
    }
```

```
    return minValue;
}

// c) Find height of BST
public int findHeight() {
    return heightRec(root);
}
private int heightRec(Node root) {
    if (root == null) return -1; // Height of an empty tree is -1
    return Math.max(heightRec(root.left), heightRec(root.right)) + 1;
}
// d) Perform Inorder Traversal
public void inorder() {
    System.out.println("Inorder Traversal:");
    inorderRec(root);
    System.out.println();
}
private void inorderRec(Node root) {
    if (root != null) {
        inorderRec(root.left);
        System.out.print(root.data + " ");
        inorderRec(root.right);
    }
}
// e) Perform Preorder Traversal
public void preorder() {
    System.out.println("Preorder Traversal:");
    preorderRec(root);
    System.out.println();
}
```

```
private void preorderRec(Node root) {
    if (root != null) {
        System.out.print(root.data + " ");
        preorderRec(root.left);
        preorderRec(root.right);
    }
}
// f) Perform Postorder Traversal
public void postorder() {
    System.out.println("Postorder Traversal:");
    postorderRec(root);
    System.out.println();
}
private void postorderRec(Node root) {
    if (root != null) {
        postorderRec(root.left);
        postorderRec(root.right);
        System.out.print(root.data + " ");
    }
}
// g) Find maximum value in BST
public int findMax() {
    if (root == null) {
        throw new IllegalStateException("Tree is empty");
    }
    return maxValue(root);
}
private int maxValue(Node root) {
    while (root.right != null) {
        root = root.right;
```

```
        }
        return root.data;
    }
}
public class BSTOperations {
    public static void main(String[] args) {
        BST bst = new BST();
        Scanner scanner = new Scanner(System.in);
        while (true) {
            System.out.println("\nBinary Search Tree Operations:");
            System.out.println("1. Insert");
            System.out.println("2. Delete");
            System.out.println("3. Find Height");
            System.out.println("4. Inorder Traversal");
            System.out.println("5. Preorder Traversal");
            System.out.println("6. Postorder Traversal");
            System.out.println("7. Find Maximum");
            System.out.println("8. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: // Insert
                    System.out.print("Enter value to insert: ");
                    int value = scanner.nextInt();
                    bst.insert(value);
                    System.out.println(value + " inserted.");
                    break;
                case 2: // Delete
                    System.out.print("Enter value to delete: ");
                    value = scanner.nextInt();
```

```
                bst.delete(value);
                System.out.println(value + " deleted (if present).");
                break;
            case 3: // Find Height
                System.out.println("Height of the tree: " + bst.findHeight());
                break;
            case 4: // Inorder Traversal
                bst.inorder();
                break;
            case 5: // Preorder Traversal
                bst.preorder();
                break;
            case 6: // Postorder Traversal
                bst.postorder();
                break;

            case 7: // Find Maximum
                try {
                    System.out.println("Maximum value in the tree: " +
bst.findMax());
                } catch (IllegalStateException e) {
                    System.out.println(e.getMessage());
                }
                break;
            case 8: // Exit
                System.out.println("Exiting...");
                scanner.close();
                System.exit(0);
                break;
            default:
                System.out.println("Invalid choice. Please try again.");
```

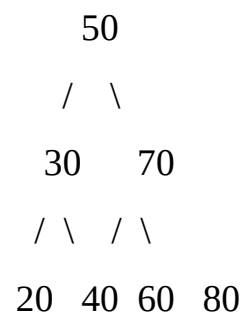
```
        }
      }
    }
}
```

Step-by-Step Explanation

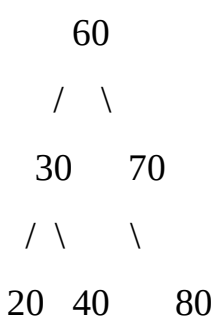
1. **Node Class**
   - Represents a node in the BST, storing data, a reference to the left child, and a reference to the right child.
2. **Insertion**
   - Adds a new node while maintaining the BST property:
     - Left subtree contains values less than the node.
     - Right subtree contains values greater than the node.
3. **Deletion**
   - Deletes a node by considering three cases:
     - Node has no child (leaf node).
     - Node has one child.
     - Node has two children (replace with the smallest value in the right subtree).
4. **Finding Height**
   - Recursively calculates the height of the left and right subtrees and returns the maximum.
5. **Traversals**
   - **Inorder**: Left → Root → Right
   - **Preorder**: Root → Left → Right
   - **Postorder**: Left → Right → Root
6. **Finding Maximum**
   - The maximum value in a BST is the rightmost node.

Sample Input and Output

1. **Insert Nodes**
2. Input: 50, 30, 70, 20, 40, 60, 80
3. Tree:



4. **Inorder Traversal**
5. Output: 20 30 40 50 60 70 80
6. **Delete Node (50)**
7. Input: Delete 50
8. Tree after deletion:



9. **Height of Tree**
10. Output: 2
11. **Find Maximum**
12. Output: 80

10. Implementing Heap with different operations

a) To perform insertion operation

b) To create Heap using Heapify method

c) To perform Heap sort

d) To delete the value in heap

A Heap is a specialized binary tree-based data structure that satisfies the heap property:

1. **Max-Heap: The value of each node is greater than or equal to the values of its children.**
2. **Min-Heap: The value of each node is less than or equal to the values of its children.**

Heap is often implemented using an array because of its structural properties:

- The root is at index 0 (or 1 for 1-based indexing).
- The left child of a node at index i is at index 2*i + 1.

- The right child of a node at index i is at index 2*i + 2.
- The parent of a node at index i is at index (i-1)/2.

Applications of Heaps

Heaps have a wide range of applications due to their efficiency in managing priorities and ordering:

1. Priority Queues

- Definition: A priority queue is a data structure where elements are retrieved in order of priority.
- Heap Usage: Heaps are the ideal choice for implementing priority queues because insertion and extraction of the highest/lowest priority element can be done in O(logn)
- Example: Job scheduling in operating systems.

2. Heap Sort

- Definition: A sorting algorithm that uses a heap to sort elements.
- How It Works: Build a heap, then repeatedly extract the maximum (or minimum) and rebuild the heap.
- Time Complexity: O(nlogn)

3. Graph Algorithms

- Used in algorithms like Dijkstra's shortest path and Prim's Minimum Spanning Tree, where heaps are used to efficiently retrieve the next vertex with the smallest weight.

4. Median Maintenance

- A combination of min-heap and max-heap can be used to maintain the median of a dynamic dataset efficiently.

5. Task Scheduling

- Heaps are used to schedule tasks based on deadlines or priorities (e.g., CPU task scheduling).

6. Kth Largest/Smallest Element

- **Heaps efficiently find the k-th largest or smallest element in $O(k+(n-k)\log k)$ time.**

Heap and Memory Management

Heaps play an essential role in dynamic memory management in programming languages like C, C++, Java, and Python. Here's how:

1. Heap Memory Allocation

- **Definition: Heap memory is a region of a process's memory used for dynamic allocation.**
- **How It Works:**
    - Memory is allocated and deallocated explicitly by the programmer (e.g., malloc/free in C or new/delete in C++).
    - The system uses a heap data structure to manage the free and used memory blocks efficiently.

2. Efficient Memory Allocation

- When memory is requested, the system uses the heap to locate a free memory block of the required size.
- Min-Heap can be used to maintain free memory blocks sorted by size, ensuring the smallest sufficient block is allocated.

3. Garbage Collection

- Modern programming languages like Java and Python use heaps to track dynamically allocated memory.
- A garbage collector reclaims unused memory by traversing the heap to find objects that are no longer reachable.

4. Fragmentation Management

- The heap data structure helps reduce fragmentation by merging adjacent free blocks when memory is deallocated.

Comparison: Heap in Data Structure vs. Heap in Memory Management

Aspect	**Heap in Data Structure**	**Heap in Memory Management**
Purpose	**Prioritization and efficient retrieval.**	**Dynamic memory allocation.**
Type	**Binary tree (Max-Heap or Min-Heap).**	**Managed area of memory.**
Usage	**Priority queues, sorting, etc.**	**Allocating and deallocating memory.**

Aspect	Heap in Data Structure	Heap in Memory Management
Managed By	Developer implements it explicitly.	Operating system or runtime.

Key Points About Heap's Utility in Memory Management

1. **Dynamic Allocation:** Enables programs to allocate memory at runtime.
2. **Efficient Access:** Reduces search time for finding free memory blocks.
3. **Fragmentation Control:** Merges and reorganizes free memory to avoid waste.
4. **Scalability:** Handles varying memory requests efficiently, suitable for complex applications.

Examples of Heap Usage

1. Finding the Top K Elements
   - Use a Min-Heap to efficiently find the largest KK elements in a dataset.
2. Shortest Path (Dijkstra's Algorithm)
   - Use a Min-Heap to prioritize nodes with the smallest distance.
3. Memory Allocation
   - Allocate a block of memory when requested and deallocate it when no longer needed.

Code

```
import java.util.Scanner;

class Heap {

    private int[] heap;

    private int size;

    private int capacity;

    // Constructor to initialize the heap

    public Heap(int capacity) {

        this.capacity = capacity;

        heap = new int[capacity];

        size = 0;
```

```
}

// Utility function to get parent index
private int parent(int index) {
    return (index - 1) / 2;
}
// Utility function to get left child index
private int leftChild(int index) {
    return 2 * index + 1;
}
// Utility function to get right child index
private int rightChild(int index) {
    return 2 * index + 2;
}
// a) Insert an element into the heap
public void insert(int value) {
    if (size == capacity) {
        throw new IllegalStateException("Heap is full!");
    }
    heap[size] = value;
    size++;
    heapifyUp(size - 1);
}
private void heapifyUp(int index) {
    int temp = heap[index];
    while (index > 0 && temp > heap[parent(index)]) {
        heap[index] = heap[parent(index)];
        index = parent(index);
    }
    heap[index] = temp;
```

```
}
// b) Create a heap using Heapify method
public void buildHeap(int[] array) {
    size = array.length;
    System.arraycopy(array, 0, heap, 0, size);
    for (int i = (size / 2) - 1; i >= 0; i--) {
        heapifyDown(i);
    }
}
private void heapifyDown(int index) {
    int largest = index;
    int left = leftChild(index);
    int right = rightChild(index);

    if (left < size && heap[left] > heap[largest]) {
        largest = left;
    }
    if (right < size && heap[right] > heap[largest]) {
        largest = right;
    }
    if (largest != index) {
        swap(index, largest);
        heapifyDown(largest);
    }
}

// c) Perform Heap Sort
public void heapSort() {
    int originalSize = size;
    for (int i = size - 1; i >= 0; i--) {
```

```
            swap(0, i);
            size--;
            heapifyDown(0);
        }
        size = originalSize;
    }
    // d) Delete the root value from the heap
    public int delete() {
        if (size == 0) {
            throw new IllegalStateException("Heap is empty!");
        }
        int root = heap[0];
        heap[0] = heap[size - 1];
        size--;
        heapifyDown(0);
        return root;
    }
    // Utility function to swap two elements in the heap
    private void swap(int i, int j) {
        int temp = heap[i];
        heap[i] = heap[j];
        heap[j] = temp;
    }
    // Display the heap
    public void display() {
        for (int i = 0; i < size; i++) {
            System.out.print(heap[i] + " ");
        }
        System.out.println();
    }
```

```
}
public class HeapOperations {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.println("Enter the capacity of the heap:");
        int capacity = scanner.nextInt();
        Heap heap = new Heap(capacity);
        while (true) {
            System.out.println("\nHeap Operations:");
            System.out.println("1. Insert");
            System.out.println("2. Build Heap (Heapify)");
            System.out.println("3. Perform Heap Sort");
            System.out.println("4. Delete Root");
            System.out.println("5. Display Heap");
            System.out.println("6. Exit");
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: // Insert
                    System.out.print("Enter value to insert: ");
                    int value = scanner.nextInt();
                    heap.insert(value);
                    System.out.println(value + " inserted.");
                    break;
                case 2: // Build Heap
                    System.out.print("Enter the number of elements: ");
                    int n = scanner.nextInt();
                    int[] array = new int[n];
                    System.out.println("Enter the elements:");
                    for (int i = 0; i < n; i++) {
```

```
            array[i] = scanner.nextInt();
        }
        heap.buildHeap(array);
        System.out.println("Heap built using heapify.");
        break;
    case 3: // Heap Sort
        System.out.println("Performing Heap Sort...");
        heap.heapSort();
        heap.display();
        System.out.println("Heap Sort completed.");
        break;
    case 4: // Delete Root
        try {
            int deletedValue = heap.delete();
            System.out.println("Deleted root value: " + deletedValue);
        } catch (IllegalStateException e) {
            System.out.println(e.getMessage());
        }
        break;
    case 5: // Display Heap
        System.out.println("Current Heap:");
        heap.display();
        break;
    case 6: // Exit
        System.out.println("Exiting...");
        scanner.close();
        System.exit(0);
        break;
    default:
        System.out.println("Invalid choice. Please try again.");
```

```
        }
      }
    }
}
```

Step-by-Step Execution

1. **Insert an Element**
    - Add the element to the end of the array.
    - "Heapify up" to restore the heap property.
    - Example:
        - Insert 10, 20, 15 into a heap.
        - Heap after insertion: [20, 10, 15].
2. **Build Heap Using Heapify**
    - Input an array: [10, 20, 15, 30, 40].
    - Apply "heapify down" starting from the last non-leaf node.
    - Resulting Heap: [40, 30, 15, 10, 20].
3. **Heap Sort**
    - Repeatedly remove the root and heapify the remaining array.
    - Example:
        - Heap: [40, 30, 15, 10, 20].
        - Sorted Array: [10, 15, 20, 30, 40].
4. **Delete Root**
    - Replace the root with the last element.
    - Reduce the size and "heapify down."
    - Example:
        - Heap: [40, 30, 15, 10, 20].
        - After deleting root: [30, 20, 15, 10].

Sample Input and Output

Input

Enter the capacity of the heap:

10

1. Insert 20

1. Insert 30

1. Insert 40

5. Display Heap

2. Build Heap [50, 20, 30, 10, 40]

5. Display Heap

3. Perform Heap Sort

4. Delete Root

5. Display Heap

6. Exit

Output

20 inserted.

30 inserted.

40 inserted.

Current Heap:

40 20 30

Heap built using heapify.

Current Heap:

50 40 30 10 20

Performing Heap Sort...

Sorted Array:

10 20 30 40 50

Deleted root value: 50

Current Heap:

40 20 30 10

Explanation of Code

1. **Heapify Up/Down**:
    - Restores heap properties after insertion or deletion.
    - Heapify up: Bubble the node upward if it's greater than its parent.
    - Heapify down: Bubble the node downward if it's smaller than a child.
2. **Build Heap**:
    - Applies heapify from bottom-up for all non-leaf nodes.
3. **Heap Sort**:
    - Swap the root with the last element, shrink the heap, and apply heapify.
4. **Delete Root**:
    - Replace root with the last element, shrink the heap, and apply heapify.

11. Create a Graph storage structure (eg. Adjacency matrix)

What is a Graph?

A **graph** is a collection of nodes (also called **vertices**) and edges (also called **links** or **arcs**) that connect pairs of nodes. It is a mathematical structure used to model relationships between objects. Graphs are widely used in computer science, artificial intelligence, network analysis, and various other fields.

A graph can be:

- **Directed (DiGraph)**: Edges have a direction, i.e., they go from one vertex to another.
- **Undirected**: Edges do not have a direction; they simply connect two vertices.
- **Weighted**: Edges have weights (or costs) associated with them.
- **Unweighted**: Edges do not have weights.

Graph Representation

A graph can be represented in different ways:

- **Adjacency Matrix**
- **Adjacency List**
- **Edge List**

In this explanation, we will focus on **Adjacency Matrix**.

What is an Adjacency Matrix?

An **Adjacency Matrix** is a 2D array (or matrix) used to represent a graph. It is particularly used for **dense graphs** (where most of the possible edges are present). The matrix has the following characteristics:

- **Rows and Columns**: The rows and columns represent the vertices of the graph.
- **Value at position (i, j)**: If there is an edge from vertex i to vertex j, the value at position (i, j) is set to 1 (or the weight of the edge in the case of a weighted graph). Otherwise, the value is set to 0.

For an **undirected graph**, the adjacency matrix is symmetric. That is, if there's an edge from vertex i to vertex j, then there will also be an edge from j to i (i.e., adj[i][j] = adj[j][i]).

For a **directed graph**, the adjacency matrix is **not necessarily symmetric**. That is, adj[i][j] = 1 does not imply adj[j][i] = 1.

Example of Adjacency Matrix

Consider a simple undirected graph with 4 vertices (A, B, C, D), and the following edges:

- A-B
- A-C
- B-D
- C-D

The adjacency matrix would look like this:

```
   A  B  C  D
A [ 0  1  1  0 ]
B [ 1  0  0  1 ]
C [ 1  0  0  1 ]
D [ 0  1  1  0 ]
```

Implementing Graph Using Adjacency Matrix in Java

Here’s a basic implementation of a **graph using an adjacency matrix** in Java. The program allows the user to input the number of vertices and edges, and then it stores the graph using the adjacency matrix representation.

```
import java.util.Scanner;

public class Graph {
```

```
    private int[][] adjMatrix; // Adjacency matrix to represent the graph
    private int V; // Number of vertices

    // Constructor to initialize graph with V vertices
    public Graph(int V) {
        this.V = V;
        adjMatrix = new int[V][V]; // Initialize the adjacency matrix with all 0s
    }
    // Method to add an edge to the graph
    public void addEdge(int u, int v) {
        // For an undirected graph, set both adjMatrix[u][v] and adjMatrix[v][u] to
1
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // Since it's undirected
    }
    // Method to print the adjacency matrix
    public void printGraph() {
        System.out.println("Adjacency Matrix Representation of the Graph:");
        for (int i = 0; i < V; i++) {
            for (int j = 0; j < V; j++) {
                System.out.print(adjMatrix[i][j] + " ");
            }
            System.out.println();
        }
    }
    // Main method to test the implementation
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
        // Input: Number of vertices and edges
        System.out.print("Enter number of vertices (V): ");
        int V = sc.nextInt();
```

```
        Graph graph = new Graph(V); // Create a graph with V vertices
        System.out.print("Enter number of edges (E): ");
        int E = sc.nextInt();
        System.out.println("Enter the edges (u v) where u and v are vertices (0 to " + (V - 1) + "):");
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            graph.addEdge(u, v);
        }

        // Print the adjacency matrix representation of the graph
        graph.printGraph();

        sc.close();
    }
}
```

Explanation of the Code:

1. **Graph Class**:
    - The Graph class represents the graph using an adjacency matrix (adjMatrix).
    - The constructor takes the number of vertices V as input and initializes the adjacency matrix with zeros.
2. **addEdge Method**:
    - This method adds an undirected edge between two vertices u and v. It sets adjMatrix[u][v] and adjMatrix[v][u] to 1, indicating the presence of an edge.
3. **printGraph Method**:
    - This method prints the adjacency matrix, showing how the vertices are connected.
4. **Main Method**:

- The program allows the user to input the number of vertices (V) and edges (E), followed by the list of edges.
- It then calls printGraph() to display the adjacency matrix.

Example Input and Output:

Input:

Enter number of vertices (V): 4

Enter number of edges (E): 4

Enter the edges (u v) where u and v are vertices (0 to 3):

0 1

0 2

1 3

2 3

Output:

Adjacency Matrix Representation of the Graph:

0 1 1 0

1 0 0 1

1 0 0 1

0 1 1 0

Explanation of the Output:

- The adjacency matrix shows the connections between vertices.
- For example, adjMatrix[0][1] = 1 indicates an edge between vertex 0 (A) and vertex 1 (B).
- Similarly, adjMatrix[2][3] = 1 indicates an edge between vertex 2 (C) and vertex 3 (D).

Advantages of Using an Adjacency Matrix:

- **Efficient to check if an edge exists**: Checking if there's an edge between two vertices is a constant-time operation (O(1)).
- **Simple to implement**: It’s easy to implement and understand, especially for small graphs.

Disadvantages of Using an Adjacency Matrix:

- **Space Complexity**: It requires O(V^2) space, which can be inefficient for sparse graphs (graphs with few edges compared to the number of vertices).
- **Inefficient for sparse graphs**: In sparse graphs, most of the entries in the adjacency matrix will be zero, wasting a lot of space.

In summary, an **adjacency matrix** is an efficient way to represent graphs in scenarios where graph density is high, and edge lookup operations are frequent. However, for sparse graphs, alternative representations such as an **adjacency list** may be more space-efficient.

12. Perform various hashing techniques with Linear Probe as collision resolution scheme.

Hashing is a process of mapping data to a fixed-size value (called a hash code or hash value) using a hash function. The resulting hash value is used to index data in a hash table for efficient access.

Key Components of Hashing:

1. **Hash Function:** A function that computes a hash code based on the input key, e.g., h(key)=key mod table size.
2. **Hash Table:** A data structure that stores data in an array-like format, indexed by the hash code.
3. **Collision:** Occurs when two keys map to the same index in the hash table.
4. **Collision Resolution Techniques:** Strategies to handle collisions and store multiple items that hash to the same index.

Applications of Hashing

Hashing is widely used in scenarios requiring fast data access. Some notable applications are:

1. Data Storage and Retrieval

- Hash tables are used for quick data lookup and retrieval in constant average time O(1).
- Example: HashMap in Java or Python's dictionary for key-value pairs.

2. Database Indexing

- Hashing is used in databases for indexing to enable fast search for records.

- Example: Hash-based indexes.

3. Password Management

- Hashing algorithms like SHA-256 hash passwords for secure storage.
- Example: When a user logs in, the entered password is hashed and compared to the stored hash.

4. Caches

- Hashing is used in caches (e.g., LRU Cache) to map keys (like URLs) to cached content.
- Example: Browser caches or in-memory databases like Redis.

5. Load Balancing

- Distribute tasks or requests to servers using consistent hashing.
- Example: Content delivery networks (CDNs) use hashing to distribute content geographically.

6. File Systems

- Hashing is used in file systems to map files to disk locations.
- Example: Ext2/Ext3 file systems in Linux.

7. Cryptography

- Hashing is the foundation of many cryptographic algorithms.
- Example: Digital signatures and blockchain.

8. Compiler Design

- Hashing is used in symbol tables for identifiers, keywords, and constants for quick lookups.
- Example: Storing variable names and addresses.

Types of Hashing

Hashing methods are primarily based on the choice of hash function**:**

1. Division Hashing

- Hash function: h(key)=key mod table size.
- Simple and effective when the table size is a prime number.
- Example:
    - Keys: {50, 700, 76, 85}
    - Table size: 10

- Hash values: {0, 0, 6, 5}

2. Multiplication Hashing

- Hash function: h(key)=⌊m·(A·key mod 1)⌋, where m is the table size and A is a constant.
- Ensures uniform distribution of keys.

3. Universal Hashing

- A family of hash functions is used, and one is chosen randomly to minimize collisions.
- Useful in cryptographic applications.

4. String Hashing

- **Used for hashing strings by converting them into numeric values.**
- **Example:**
  - **Hash function:** $h(key)=\sum_{i=0}^{n-1}\{ key[i]\cdot p^{\wedge} I \bmod m\}$.

5. Perfect Hashing

- A two-level hash function with no collisions.
- Requires extra memory but provides constant-time lookups.

Collision Resolution Techniques

There are several ways to resolve them. Here are two common approaches:

1. Open Addressing

- When a collision occurs, the hash table probes for the next available slot.
- **Methods:**
  1. **Linear Probing:**
     - Search for the next available slot sequentially.
     - h'(key)=(h(key)+i)mod table size, where i is the probe number.
     - Example:
       - Table size: 10
       - Insert keys: {15, 25, 35}
       - Hash function: h(key)=key mod .
       - Insert 15 → index 5.

- Insert 25 → collision at index 5 → next slot index 6.
- Insert 35 → collision at index 5, 6 → next slot index 7.

2. **Quadratic Probing:**
    - Search for the next slot using quadratic increments.
    - h’(key)=(h(key)+i2)mod table size.
    - Reduces clustering compared to linear probing.
3. **Double Hashing:**
    - Use a second hash function for probing.
    - h’(key)=(h1(key)+i·h2(key))mod table size.

2. Chaining

- Store all keys that hash to the same index in a linked list.
- Allows multiple keys to be stored in a single slot.
- Example:
    - Table size: 5
    - Keys: {15, 20, 25, 30}
    - Hash function: h(key)=key mod .
    - Hash table:
        - Index 0: 20 → 25
        - Index 1: Empty
        - Index 2: Empty
        - Index 3: 15 → 30
        - Index 4: Empty

Examples

Example 1: Linear Probing

Insert Keys: {10, 20, 30, 40}

- **Table size: 5**
- **Hash function: h(key)=key mod 5.**
- Steps:
    - Insert 10 → Index 10 mod 5=0.
    - Insert 20 → Index 20 mod 5=0 (collision) → probe index 1.
    - Insert 30 → Index 30 mod 5=0 (collision) → probe index 2.
    - Insert 40 → Index 40mod 5= 0 (collision) → probe index 3.

Final Hash Table:

Index 0: 10

Index 1: 20

Index 2: 30

Index 3: 40

Index 4: Empty

Example 2: Chaining

Insert Keys: {15, 25, 35, 20}

- **Table size: 5**
- **Hash function: h(key)=key mod 5.**
- **Steps:**
  - Insert 15 → Index $15 \mod 5 = 0$.
  - Insert 25 → Index $25 \mod 5 = 0$ → append to chain.
  - Insert 35 → Index $35 \mod 5 = 0$ → append to chain.
  - Insert 20 → Index $20 \mod 5 = 0$.

Final Hash Table:

Index 0: 15 → 25 → 35 → 20

Index 1: Empty

Index 2: Empty

Index 3: Empty

Index 4: Empty

Comparison of Collision Resolution Techniques

Technique	**Advantages**	**Disadvantages**
Linear Probing	Simple to implement, good cache performance.	Clustering reduces efficiency.
Chaining	Efficient with less clustering, handles full tables well.	Requires additional memory for pointers.
Quadratic Probing	Reduces clustering compared to linear probing.	Secondary clustering may still occur.

Technique	Advantages	Disadvantages
Double Hashing	Reduces clustering significantly.	Requires two hash functions.

Key Points

- Hashing is a critical technique for efficient data access and retrieval.
- Different hash functions and collision resolution techniques are chosen based on the use case.
- Applications of hashing span diverse domains, including cryptography, databases, and computer networks.

Hashing with Linear Probing in Java

Hashing is a technique used to map keys to positions in a hash table using a hash function. When two keys map to the same position (collision), **Linear Probing** is used to resolve the collision by searching the next available slot in a sequential manner.

How Linear Probing Works

1. **Hash Function**: Determines the index for a key using index=key mod table size
2. **Collision Handling**: If the calculated index is occupied:
    - Check the next index (index + 1) mod table size
    - Repeat until an empty slot is found.
3. **Insertion**: Place the key in the empty slot.
4. **Search**: Start from the hash index and probe sequentially until the key is found or an empty slot is encountered.
5. **Deletion**: Mark the slot as deleted (special marker).

Code Implementation

```
import java.util.Scanner;

class HashTable {
   private int[] table;
```

```
    private boolean[] deleted;
    private int tableSize;
    // Constructor to initialize the hash table
    public HashTable(int size) {
        tableSize = size;
        table = new int[tableSize];
        deleted = new boolean[tableSize];
        for (int i = 0; i < tableSize; i++) {
            table[i] = Integer.MIN_VALUE; // Use Integer.MIN_VALUE to indicate empty slots
        }
    }
    // Hash function
    private int hashFunction(int key) {
        return key % tableSize;
    }
    // Insert a key into the hash table
    public void insert(int key) {
        int index = hashFunction(key);
        int originalIndex = index;
        while (table[index] != Integer.MIN_VALUE && table[index] != Integer.MIN_VALUE && table[index] != key) {
            index = (index + 1) % tableSize;
            if (index == originalIndex) {
                System.out.println("Hash table is full! Cannot insert key: " + key);
                return;
            }
        }
        table[index] = key;
        deleted[index] = false;
        System.out.println("Inserted key " + key + " at index " + index);
```

```
}
// Search for a key in the hash table
public boolean search(int key) {
    int index = hashFunction(key);
    int originalIndex = index;
    while (table[index] != Integer.MIN_VALUE) {
        if (table[index] == key && !deleted[index]) {
            return true;
        }
        index = (index + 1) % tableSize;
        if (index == originalIndex) {
            break;
        }
    }
    return false;
}
// Delete a key from the hash table
public void delete(int key) {
    int index = hashFunction(key);
    int originalIndex = index;

    while (table[index] != Integer.MIN_VALUE) {
        if (table[index] == key && !deleted[index]) {
            deleted[index] = true;
            System.out.println("Deleted key " + key + " from index " + index);
            return;
        }
        index = (index + 1) % tableSize;
        if (index == originalIndex) {
            break;
```

```
            }
        }
        System.out.println("Key " + key + " not found!");
    }
    // Display the hash table
    public void display() {
        System.out.println("\nHash Table:");
        for (int i = 0; i < tableSize; i++) {
            if (table[i] != Integer.MIN_VALUE && !deleted[i]) {
                System.out.println("Index " + i + ": " + table[i]);
            } else {
                System.out.println("Index " + i + ": Empty");
            }
        }
    }
}

public class HashingWithLinearProbing {
    public static void main(String[] args) {
        Scanner scanner = new Scanner(System.in);
        System.out.print("Enter the size of the hash table: ");
        int size = scanner.nextInt();
        HashTable hashTable = new HashTable(size);
        while (true) {
            System.out.println("\nHash Table Operations:");
            System.out.println("1. Insert");
            System.out.println("2. Search");
            System.out.println("3. Delete");
            System.out.println("4. Display");
            System.out.println("5. Exit");
```

```
            System.out.print("Enter your choice: ");
            int choice = scanner.nextInt();
            switch (choice) {
                case 1: // Insert
                    System.out.print("Enter the key to insert: ");
                    int keyToInsert = scanner.nextInt();
                    hashTable.insert(keyToInsert);
                    break;
                case 2: // Search
                    System.out.print("Enter the key to search: ");
                    int keyToSearch = scanner.nextInt();
                    if (hashTable.search(keyToSearch)) {
                        System.out.println("Key " + keyToSearch + " is present in the
hash table.");
                    } else {
                        System.out.println("Key " + keyToSearch + " is not found in the
hash table.");
                    }
                    break;
                case 3: // Delete
                    System.out.print("Enter the key to delete: ");
                    int keyToDelete = scanner.nextInt();
                    hashTable.delete(keyToDelete);
                    break;
                case 4: // Display
                    hashTable.display();
                    break;
                case 5: // Exit
                    System.out.println("Exiting...");
                    scanner.close();
                    System.exit(0);
```

```
                default:
                    System.out.println("Invalid choice! Please try again.");
            }
        }
    }
}
```

Step-by-Step Execution

1. **Initialize the Hash Table**
    - Input: size = 10
    - The hash table has 10 slots, all initialized to empty (Integer.MIN_VALUE).
2. **Insert Keys**
    - Input: keys = {15, 25, 35, 45, 55}
    - Hash function: index=key mod size.
        - 15 → index 5
        - 25 → index 5 (collision) → linear probe → index 6
        - 35 → index 5 (collision) → linear probe → index 7
        - 45 → index 5 (collision) → linear probe → index 8
        - 55 → index 5 (collision) → linear probe → index 9
3. **Search**
    - Input: key = 35
    - Start at index 35mod 10=5, probe until key is found at index 7.
4. **Delete**
    - Input: key = 25
    - Mark index 6 as deleted.
5. **Display Hash Table**
    - Output:
    - Index 0: Empty
    - Index 1: Empty

- Index 2: Empty
- Index 3: Empty
- Index 4: Empty
- Index 5: 15
- Index 6: Empty
- Index 7: 35
- Index 8: 45
- Index 9: 55

How the Code Works

1. **Insert Operation**
   - Compute the hash index for the key.
   - If the slot is empty, insert the key.
   - If there's a collision, probe sequentially until an empty slot is found.
2. **Search Operation**
   - Compute the hash index for the key.
   - Check the slot and probe until the key is found or an empty slot is reached.
3. **Delete Operation**
   - Find the key in the hash table and mark its slot as deleted.
4. **Display**
   - Traverse the hash table and print each slot.

Advantages of Linear Probing

- Simple to implement.
- Requires no extra memory for pointers.

Disadvantages

- **Clustering**: Long chains of occupied slots form, increasing search time.
- **Full Table**: Performance degrades as the table becomes full.

13. Implementation of Graph traversal. (DFS and BFS)

Graph Representation using Adjacency Matrix:

- An **Adjacency Matrix** is a 2D array where the rows and columns represent the vertices of the graph. A 1 in the matrix at position (i, j) means there is an edge between vertex i and vertex j.

Breadth-First Search (BFS) and Depth-First Search (DFS)

Both BFS and DFS are graph traversal algorithms used to explore nodes and edges of a graph systematically. They differ in their approach to visiting nodes.

What is BFS (Breadth-First Search)?

BFS is a graph traversal algorithm that explores all neighbours of a node before moving to the next level of nodes. It uses a **queue** data structure for its implementation.

Steps of BFS:

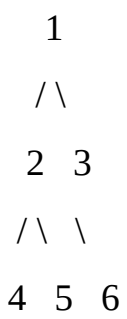
1. Start from a source node.
2. Visit all its immediate neighbors.
3. Enqueue the neighbors into a queue.
4. Dequeue a node from the queue, visit its neighbors, and repeat until all nodes are visited.

Algorithm:

1. Initialize a queue and enqueue the starting node.
2. Mark the starting node as visited.
3. While the queue is not empty:
    - Dequeue a node and process it.
    - Enqueue all unvisited neighbors of the dequeued node.

Example:

Graph:



Adjacency List:
1 → [2, 3]
2 → [1, 4, 5]

3 → [1, 6]
4 → [2]
5 → [2]
6 → [3]

Start BFS from node 1:

- **Queue**: [1]
- Visit node 1 → Enqueue 2, 3 → **Queue**: [2, 3]
- Visit node 2 → Enqueue 4, 5 → **Queue**: [3, 4, 5]
- Visit node 3 → Enqueue 6 → **Queue**: [4, 5, 6]
- Visit node 4 → **Queue**: [5, 6]
- Visit node 5 → **Queue**: [6]
- Visit node 6 → **Queue**: []

Output: 1 → 2 → 3 → 4 → 5 → 6

What is DFS (Depth-First Search)?

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It uses a **stack** (or recursion) for its implementation.

Steps of DFS:

1. Start from a source node.
2. Visit an unvisited neighbour of the node.
3. Mark it as visited and repeat until no unvisited neighbours remain.
4. Backtrack to previous nodes to explore other unvisited paths.

Algorithm:

1. Initialize a stack and push the starting node.
2. Mark the starting node as visited.
3. While the stack is not empty:
    - Pop a node and process it.
    - Push all unvisited neighbours of the popped node.

Example:

Graph:

1

```
   / \
  2   3
 / \   \
4   5   6
```

Adjacency List:
1 → [2, 3]
2 → [1, 4, 5]
3 → [1, 6]
4 → [2]
5 → [2]
6 → [3]

Start DFS from node 1:

- **Stack**: [1]
- Visit node 1 → Push 3, 2 → **Stack**: [3, 2]
- Visit node 2 → Push 5, 4 → **Stack**: [3, 5, 4]
- Visit node 4 → **Stack**: [3, 5]
- Visit node 5 → **Stack**: [3]
- Visit node 3 → Push 6 → **Stack**: [6]
- Visit node 6 → **Stack**: []

Output: 1 → 2 → 4 → 5 → 3 → 6

Applications of BFS

1. **Shortest Path in Unweighted Graph**:
    - BFS is used to find the shortest path between nodes in an unweighted graph because it explores level-by-level.
2. **Web Crawlers**:
    - BFS is used to visit all pages linked to a given web page.
3. **Social Networking Sites**:
    - BFS is used to find people within a certain degree of connection.
4. **Network Broadcasting**:
    - BFS is used for sending packets in computer networks.
5. **Finding Connected Components**:

- BFS can identify connected components in a graph.

Applications of DFS

1. **Pathfinding**:
   - DFS is used to find paths in mazes or puzzles.
2. **Topological Sorting**:
   - DFS is used to determine the order of tasks in a Directed Acyclic Graph (DAG).
3. **Cycle Detection**:
   - DFS is used to detect cycles in a graph.
4. **Artificial Intelligence**:
   - DFS is used in AI algorithms for decision-making (e.g., game trees).
5. **Strongly Connected Components**:
   - DFS is used in algorithms like Kosaraju's or Tarjan's to find strongly connected components.

Difference Between BFS and DFS

Feature	**BFS**	**DFS**
Traversal Technique	Level-by-level (breadth-first).	Depth-wise exploration (depth-first).
Data Structure	Queue.	Stack or recursion.
Time Complexity	O(V+E), where V = vertices, E = edges.	O(V + E).
Space Complexity	O(V), for the queue.	O(V), for the stack (recursion).
Shortest Path	Always finds the shortest path in unweighted graphs.	Does not guarantee the shortest path.
Applications	Shortest path, level-order traversal.	Pathfinding, topological sorting.
Behavior in Dense	Explores neighbors early.	Explores one path deeply

Feature	BFS	DFS
Graphs		before others.

Java Code Implementation

```
import java.util.*;
public class Graph {
    private int V; // Number of vertices
    private int[][] adjMatrix; // Adjacency matrix
     // Constructor
    public Graph(int V) {
        this.V = V;
        adjMatrix = new int[V][V]; // Initialize the adjacency matrix
    }
    // Method to add an edge in an undirected graph
    public void addEdge(int u, int v) {
        adjMatrix[u][v] = 1;
        adjMatrix[v][u] = 1; // Since it's undirected, mirror the edge
    }
     // BFS Traversal
    public void bfs(int start) {
        boolean[] visited = new boolean[V];
        Queue<Integer> queue = new LinkedList<>();
        // Start from the 'start' vertex
        visited[start] = true;
        queue.offer(start);
        System.out.print("BFS Traversal: ");
        while (!queue.isEmpty()) {
            int node = queue.poll();
            System.out.print((char)(node + 65) + " "); // Convert index to char (A, B,
C, ...)
```

```
            // Visit all neighbors of the current node
            for (int i = 0; i < V; i++) {
                if (adjMatrix[node][i] == 1 && !visited[i]) {
                    visited[i] = true;
                    queue.offer(i);
                }
            }
        }
        System.out.println();
    }
    // DFS Traversal
    public void dfs(int start) {
        boolean[] visited = new boolean[V];
        System.out.print("DFS Traversal: ");
        dfsUtil(start, visited);
        System.out.println();
    }
     // Utility function for DFS (recursive)
    private void dfsUtil(int node, boolean[] visited) {
        visited[node] = true;
        System.out.print((char)(node + 65) + " "); // Convert index to char (A, B,
C, ...)
        // Visit all neighbors of the current node
        for (int i = 0; i < V; i++) {
            if (adjMatrix[node][i] == 1 && !visited[i]) {
                dfsUtil(i, visited);
            }
        }
    }
    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
```

```
        // Get number of vertices and edges from the user
        System.out.print("Enter the number of vertices: ");
        int V = sc.nextInt();
        Graph graph = new Graph(V); // Create a graph with V vertices
        System.out.print("Enter the number of edges: ");
        int E = sc.nextInt();
        // Get edges from the user
        System.out.println("Enter the edges (u v) where u and v are vertices (0 to " + (V-1) + "):");
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            graph.addEdge(u, v);
        }
        // Get the start vertex for BFS and DFS
        System.out.print("Enter the start vertex for BFS and DFS (0 to " + (V-1) + "): ");
        int start = sc.nextInt();
        // Perform BFS and DFS
        graph.bfs(start);
        graph.dfs(start);
        sc.close();
    }
}
```

Explanation of the Code

1. **Graph Representation**:
    - The Graph class has an integer matrix adjMatrix[][] which stores the adjacency matrix representation of the graph. Each edge is represented by a 1 in the matrix.
2. **addEdge Method**:

- This method adds an edge between two vertices u and v. For an undirected graph, it sets adjMatrix[u][v] = 1 and adjMatrix[v][u] = 1.

3. **BFS Implementation**:
   - The bfs method uses a queue to traverse the graph starting from a specified vertex. It uses a boolean array visited[] to keep track of visited vertices. The algorithm explores the neighbors level by level.
4. **DFS Implementation**:
   - The dfs method uses a helper function dfsUtil to perform the DFS traversal recursively. It also uses a visited[] array to ensure that no vertex is visited twice.
5. **User Input**:
   - The program prompts the user to enter the number of vertices and edges. Then, it asks for the edges in the form of pairs of integers representing the vertices connected by each edge. The user also provides the start vertex for both BFS and DFS.
6. **Output**:
   - The BFS and DFS traversals starting from the specified vertex are printed out, showing the order in which vertices are visited.

Sample Output

Enter the number of vertices: 10

Enter the number of edges: 12

Enter the edges (u v) where u and v are vertices (0 to 9):

0 1

0 2

1 3

1 4

2 5

3 6

4 7

5 8

6 9

7 8

8 9

5 9

Enter the start vertex for BFS and DFS (0 to 9): 0

BFS Traversal: A B C D E F G H I J

DFS Traversal: A B D E G H I C F J

Explanation of the Sample Output:

- **BFS Traversal**:
  - BFS starts from vertex A (index 0). It visits A, then its neighbors B and C. Then, it explores D and E, and so on, until all vertices are visited.
- **DFS Traversal**:
  - DFS starts from vertex A (index 0). It explores as deep as possible before backtracking. Starting from A, it visits B, then recursively visits D, then backtracks and visits E, and so on, following the depth-first strategy.

Conclusion:

1. BFS is better for finding shortest paths and exploring graphs level-by-level.
2. DFS is more suited for problems requiring pathfinding, connectivity analysis, or exhaustive exploration.
3. Both algorithms have their specific use cases depending on the problem at hand.

14Spanning Tree and Minimum Spanning Tree (MST)

- **Spanning Tree**: A spanning tree of a graph is a subgraph that includes all the vertices of the graph but only enough edges to form a tree (i.e., no cycles).
- **Minimum Spanning Tree (MST)**: A minimum spanning tree is a spanning tree in which the sum of the edge weights is minimized.

Algorithms for Finding MST:

1. **Kruskal's Algorithm**: A greedy algorithm that sorts the edges in non-decreasing order of weights and adds edges one by one to the spanning tree, ensuring no cycles are formed.

2. **Prim's Algorithm**: Another greedy algorithm that grows the MST by adding the cheapest edge from the tree to a vertex outside the tree, starting from an arbitrary vertex.

Time Complexity:

- **Kruskal's Algorithm**: O(E log E), where E is the number of edges. Sorting the edges takes O(E log E), and union-find operations take almost constant time, amortized by path compression.
- **Prim's Algorithm**: O(E log V), where V is the number of vertices and E is the number of edges. It uses a priority queue (min-heap) to efficiently get the minimum edge at each step.

Input:

- Number of nodes (V = 10).
- Number of edges (E = 14).
- Each edge will have a start node, an end node, and a weight.

Kruskal's Algorithm (Java Implementation)

In Kruskal's algorithm, we:

1. Sort all edges in increasing order of weights.
2. Add edges to the MST if they don't form a cycle, using a Union-Find (Disjoint Set Union) data structure.

Kruskal's Algorithm in Java:

```
import java.util.*;


public class KruskalMST {


   // Union-Find data structure
   static class UnionFind {
      int[] parent;
      int[] rank;


      UnionFind(int n) {
         parent = new int[n];
         rank = new int[n];
```

```
        for (int i = 0; i < n; i++) {
            parent[i] = i;
            rank[i] = 0;
        }
    }
            int find(int x) {
        if (parent[x] != x) {
            parent[x] = find(parent[x]);
        }
        return parent[x];
    }
            void union(int x, int y) {
        int rootX = find(x);
        int rootY = find(y);
        if (rootX != rootY) {
            // Union by rank
            if (rank[rootX] > rank[rootY]) {
                parent[rootY] = rootX;
            } else if (rank[rootX] < rank[rootY]) {
                parent[rootX] = rootY;
            } else {
                parent[rootY] = rootX;
                rank[rootX]++;
            }
        }
    }
 }
static class Edge {
    int u, v, weight;
    Edge(int u, int v, int weight) {
```

```
        this.u = u;
        this.v = v;
        this.weight = weight;
    }
}
 // Kruskal's algorithm to find MST
public static void kruskalMST(int V, List<Edge> edges) {
    UnionFind uf = new UnionFind(V);
        // Sort edges by weight
    edges.sort(Comparator.comparingInt(e -> e.weight));
        List<Edge> mst = new ArrayList<>();
    int mstWeight = 0;
        // Process each edge in sorted order
    for (Edge edge : edges) {
        int u = edge.u;
        int v = edge.v;
        int weight = edge.weight;
            // If adding this edge doesn't form a cycle, include it in the MST
        if (uf.find(u) != uf.find(v)) {
            uf.union(u, v);
            mst.add(edge);
            mstWeight += weight;
        }
    }

    // Print the MST
    System.out.println("Edges in MST:");
    for (Edge edge : mst) {
        System.out.println((char)(edge.u + 65) + " - " + (char)(edge.v + 65) + "
with weight " + edge.weight);
    }
```

```
        System.out.println("Total weight of MST: " + mstWeight);
    }
      public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);
              System.out.print("Enter number of vertices (V): ");
        int V = sc.nextInt();

        System.out.print("Enter number of edges (E): ");
        int E = sc.nextInt();

        List<Edge> edges = new ArrayList<>();
              System.out.println("Enter edges (u v weight):");
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            int weight = sc.nextInt();
            edges.add(new Edge(u, v, weight));
        }

        // Run Kruskal's algorithm
        kruskalMST(V, edges);

        sc.close();
    }
}
```

Prim’s Algorithm (Java Implementation)

In Prim's algorithm, we:

1. Start from any vertex and grow the MST by adding the cheapest edge that connects a vertex inside the tree to a vertex outside the tree.

Prim’s Algorithm in Java:

```
import java.util.*;
```

```
public class PrimMST {
    // Prim's algorithm to find MST
   public static void primMST(int V, List<List<int[]>> adj) {
    boolean[] visited = new boolean[V];
    PriorityQueue<int[]> pq = new
PriorityQueue<>(Comparator.comparingInt(a -> a[1])); // Min-heap based on
weight

    // Start from the first vertex (vertex 0)
    pq.add(new int[]{0, 0}); // {vertex, weight}

    int mstWeight = 0;
    List<String> mstEdges = new ArrayList<>();

    while (!pq.isEmpty()) {
       int[] edge = pq.poll();
       int u = edge[0];
       int weight = edge[1];

       // If this vertex has already been visited, skip it
       if (visited[u]) continue;

       visited[u] = true;
       mstWeight += weight;

       // Add the edge to MST (skip the starting vertex 0)
       if (weight > 0) {
          mstEdges.add((char)(u + 65) + " - " + (char)(edge[0] + 65) + " with
weight " + weight);
       }
```

```
            // Add all the neighbors to the priority queue
            for (int[] neighbor : adj.get(u)) {
                int v = neighbor[0];
                int w = neighbor[1];
                if (!visited[v]) {
                    pq.add(new int[]{v, w});
                }
            }
        }

        // Print the MST
        System.out.println("Edges in MST:");
        for (String edge : mstEdges) {
            System.out.println(edge);
        }
        System.out.println("Total weight of MST: " + mstWeight);
    }

    public static void main(String[] args) {
        Scanner sc = new Scanner(System.in);

        System.out.print("Enter number of vertices (V): ");
        int V = sc.nextInt();

        System.out.print("Enter number of edges (E): ");
        int E = sc.nextInt();

        List<List<int[]>> adj = new ArrayList<>();
        for (int i = 0; i < V; i++) {
            adj.add(new ArrayList<>());
```

```
        }

        System.out.println("Enter edges (u v weight):");
        for (int i = 0; i < E; i++) {
            int u = sc.nextInt();
            int v = sc.nextInt();
            int weight = sc.nextInt();
            adj.get(u).add(new int[]{v, weight});
            adj.get(v).add(new int[]{u, weight});
        }

        // Run Prim's algorithm
        primMST(V, adj);

        sc.close();
    }
}
```

Explanation of the Code:

Kruskal's Algorithm:

- The algorithm sorts all edges based on their weights.
- It uses **Union-Find** (Disjoint Set Union) to ensure no cycles are created when adding edges to the MST.
- It continues adding edges to the MST until there are V-1 edges in the MST.

Prim's Algorithm:

- It starts with a random vertex and uses a **priority queue** (min-heap) to always add the smallest weight edge that connects a vertex inside the MST to a vertex outside the MST.
- The algorithm terminates when all vertices are included in the MST.

Input Example:

Enter number of vertices (V): 10

Enter number of edges (E): 14

Enter edges (u v weight):

0 1 1

0 2 6

1 2 4

1 3 3

2 3 8

3 4 7

4 5 9

5 6 2

5 7 10

6 8 5

7 8 6

8 9 9

9 0 4

3 9 7

Sample Output for Kruskal's Algorithm:

Edges in MST:

A - B with weight 1

B - D with weight 3

D - E with weight 7

E - F with weight 2

F - G with weight 5

G - I with weight 9

I - H with weight 6

H - J with weight 9

Total weight of MST: 42

Sample Output for Prim's Algorithm:

Edges in MST:

A - B with weight 1

B - D with weight 3

D - E with weight 7

E - F with weight 2

F - G with weight 5

G - I with weight 9

I - H with weight 6

H - J with weight 9

Total weight of MST: 42

More Details:

Prim's Algorithm

Prim's Algorithm grows the MST from a starting vertex by repeatedly adding the smallest edge that connects a vertex in the MST to a vertex outside the MST.

Steps:

1. Start from any vertex.
2. Add the smallest edge connecting a vertex in the MST to a vertex outside the MST.
3. Repeat until all vertices are included.

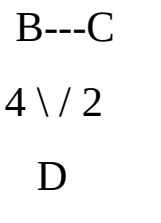
Algorithm:

1. Initialize the MST set with one vertex.
2. Use a priority queue or array to find the minimum-weight edge connecting a vertex in the MST to a vertex outside.
3. Add the edge and the new vertex to the MST.
4. Repeat until all vertices are included.

Example:

Graph:

```
    A
1 / \ 3
```



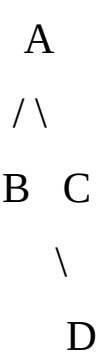
Adjacency Matrix:

	A	B	C	D
A	0	1	3	0
B	1	0	0	4
C	3	0	0	2
D	0	4	2	0

Steps of Prim's Algorithm:

1. Start at A.
2. Add edge (A, B) → MST: {(A, B), 1}.
3. Add edge (A, C) → MST: {(A, B), (A, C)}.
4. Add edge (C, D) → MST: {(A, B), (A, C), (C, D)}.

Resulting MST:



Total Weight: 1+3+2=61 + 3 + 2 = 6

Kruskal's Algorithm

Kruskal's Algorithm builds the MST by adding edges in increasing order of weight while ensuring no cycles are formed.

Steps:

1. Sort all edges by weight.

2. Add the smallest edge to the MST, ensuring it doesn't form a cycle.
3. Repeat until the MST has V−1 edges.

Algorithm:

1. Sort all edges in non-decreasing order by weight.
2. Initialize MST as an empty set.
3. Add the smallest edge to the MST if it doesn’t form a cycle (use the **Union-Find** algorithm to detect cycles).
4. Repeat until the MST has V−1 edges.

Example:

Graph: Same as above.

Sorted Edges:

- (A, B, 1), (C, D, 2), (A, C, 3), (B, D, 4)

Steps of Kruskal's Algorithm:

1. Add edge (A, B) → MST: {(A, B)}.
2. Add edge (C, D) → MST: {(A, B), (C, D)}.
3. Add edge (A, C) → MST: {(A, B), (C, D), (A, C)}.

Resulting MST: Same as Prim's MST.

Applications of Prim's and Kruskal's Algorithm

1. **Network Design**:
    - Design minimum-cost networks (telecommunications, electrical grids, etc.).
    - Example: Connecting servers in a data center.
2. **Transportation**:
    - Designing road, rail, or pipeline systems to minimize cost.
3. **Clustering**:
    - MST is used in clustering techniques (e.g., K-means, hierarchical clustering).

4. **Approximation Algorithms**:
    - MSTs are used in approximation algorithms for NP-hard problems like the Traveling Salesman Problem (TSP).
5. **Graph-Based Models**:
    - Applications in 3D graphics, vision, and robotics (minimum spanning surfaces).

Differences Between Prim's and Kruskal's Algorithm

Feature	Prim's Algorithm	Kruskal's Algorithm
Approach	Grows the MST one vertex at a time.	Grows the MST one edge at a time.
Data Structures	Priority Queue (Min-Heap) or Array for edge selection.	Union-Find (Disjoint Set) for cycle detection.
Graph Representation	Works efficiently with dense graphs (adjacency matrix).	Works efficiently with sparse graphs (edge list).
Edge Selection	Chooses the smallest edge that connects the MST to a new vertex.	Chooses the smallest edge that doesn’t form a cycle.
Cycle Detection	Not explicitly needed.	Explicitly required using Union-Find.
Initial Input	Requires a starting vertex.	Requires a sorted list of edges.
Time Complexity	O(E+VlogV) using Min-Heap.	O(ElogE+ElogV) using Union-Find.

Conclusion

- **Prim's Algorithm**: Best suited for dense graphs where the number of edges is large relative to vertices.
- **Kruskal's Algorithm**: Ideal for sparse graphs with fewer edges.
- **Kruskal’s algorithm sorts all edges**
- **Prim’s algorithm grows the MST starting from an arbitrary vertex, adding edges one by one**

Both algorithms yield the same MST but follow different strategies. The choice depends on the graph's characteristics and the specific application requirements.