Data Structures And Algorithms

***Data Structures:*** Data structures are the way data is organized, stored, and accessed in a computer's memory. They define the relationship between the data elements and enable efficient operations like insertion, deletion, and searching.  
***Algorithms:*** Algorithms are step-by-step procedures or sets of instructions used to solve problems or perform tasks. They specify the sequence of operations to be carried out to achieve a specific goal.

#### Difference between data structures and algorithms

|  |  |  |
| --- | --- | --- |
|  | Data Structures | Algorithms |
| 1. Nature | Data structures focus on organizing and storing data efficiently. | algorithms focus on solving problems or performing tasks efficiently. |
| 1. Purpose | Data structures provide a way to represent and manage data. | algorithms provide methods for manipulating that data to achieve a desired outcome. |
| 1. Examples | arrays, linked lists, trees, and graphs. | sorting algorithms (e.g., bubble sort, quicksort), searching algorithms (e.g., linear search, binary search), and graph traversal algorithms (e.g., depth-first search, breadth-first search). |

Data Structures can be divided into two part, 1> Linear data structure, 2>Non-linear Data structures

* **Linear Data Structure:**

A linear data structure is a type of data organization where elements are arranged sequentially and accessed in a linear order. In other words, each element is connected to its previous and next element in a linear fashion. Elements are stored and accessed sequentially, one after the other. Examples of linear data structures include arrays, linked lists, stacks, and queues.

* **Non-Linear Data Structure:**

A non-linear data structure is a type of data organization where elements are not arranged sequentially like in linear data structures. In non-linear data structures, elements may be interconnected in a more complex manner, forming relationships other than the simple linear order. Common examples of non-linear data structures include trees and graphs. Trees have a hierarchical structure with parent-child relationships, while graphs consist of vertices (nodes) and edges that connect these vertices, allowing for arbitrary relationships between them.

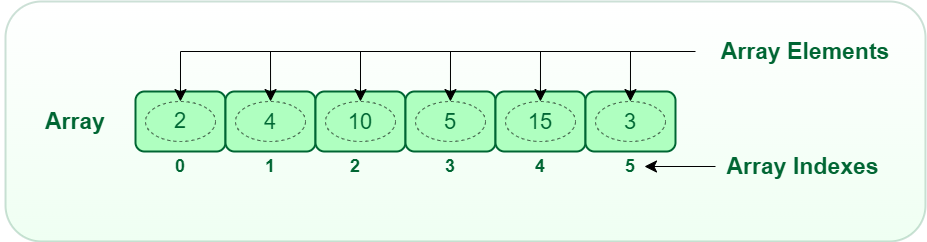
**Linear Data Structures:**

1. Arrays
2. Linked Lists
3. Stacks
4. Queues
5. Hash Tables (though implemented using arrays, they provide constant-time access to elements and are often considered separately due to their unique characteristics)

Non-Linear Data Structures:

1. Trees
   * 1. Binary Tree
     2. Binary Search Tree
     3. AVL Tree
     4. B-Tree
     5. Red-Black Tree
     6. Trie
2. Graphs
   * 1. Directed Graph
     2. Undirected Graph
     3. Weighted Graph
     4. Directed Acyclic Graph (DAG)
     5. Bipartite Graph

Arrays



Arrays are fundamental data structures in Java, used to store collections of elements of the same data type. They offer efficient random access and a variety of operations, making them versatile for many programming tasks.  
  
Definition: An array is a collection of variables of the same data type, accessed using a unique index.

|  |
| --- |
| Java arrays are always dynamically allocated. This means that the memory for an array is allocated on the heap, a region managed by the Java Virtual Machine (JVM). The heap is a flexible memory space that can grow or shrink as needed during program execution.  When you declare an array in Java, you only create a reference to the actual array data. This reference is stored on the stack, a fixed-size memory space used for storing local variables and method parameters.  The size of the array is specified at initialization using the new keyword.   eg: 1> int arr[] = {1,2,3}; // declaration and definition at same line  2> int[] myArray = new int[10]; //fixed sized array  3> int[] myArray; //declaration without initialization and initialization must be done later using new with a size.   1. int[][] jaggedArray = new int[3][]; //jugged Array |
| C arrays use static arrays for fixed-size data with performance benefits,  or dynamic arrays for larger or unknown-size data, but remember to manage memory manually.  Eg:  1> int myArray[5] = {1, 2, 3, 4, 5}; // Declares and defines an array of 5 integers with initial values  2> int anotherArray[10]; // Declares an array of 10 integers, but doesn't initialize it  3> int\* ptr; // Declare a pointer to an integer  ptr=myArray; // assigning base address of an array to the pointer;  4> ptr = (int\*)malloc(sizeof(int) \* 5); |

Common Array Operations:

Traversing: Visit each element sequentially (using loops).

Searching: Find an element with a specific value (e.g., linear search, binary search).

Sorting: Arrange elements in a specific order (e.g., bubble sort, selection sort, insertion sort, merge sort, quicksort).

Inserting: Add an element at a specific position (shifting elements if needed).

Deleting: Remove an element (shifting elements if necessary).

Reversing: Change the order of elements from front to back.

Copying: Create a new array with the same elements.

Filling: Set all elements to a specific value.

***Array Algorithms:***

***Searching Algorithms***

1. **Linear Search:**

**Definition:** Compares the target value with each element in the array sequentially.

Time complexity: O(n) (average and worst case).

Code:

|  |
| --- |
| Java Code public static int linearSearch(int[] arr, int target) {  for (int i = 0; i < arr.length; i++) {  if (arr[i] == target) {  return i; // Found!  }  }  return -1; // Not found  } |
| C code int linearSearch(int arr[], int length, int target) {  for (int i = 0; i < length; i++) {  if (arr[i] == target) {  return i; // Found!  }  }  return -1; // Not found  } |

1. **Binary Search:**

Definition: Requires a sorted array. Divides the array in half, compares the target with the middle element, and recursively searches the appropriate half.

Time complexity: O(log n).

Code:

|  |
| --- |
| Java Code public static int binarySearch(int[] arr, int target) {  int low = 0, high = arr.length - 1;  while (low <= high) {  int mid = (low + high) / 2;  if (arr[mid] == target) {  return mid;  } else if (arr[mid] < target) {  low = mid + 1; // Search right half  } else {  high = mid - 1; // Search left half  }  }  return -1; // Not found  } |
| C code int binarySearch(int arr[], int length, int target) {  int low = 0;  int high = length - 1;  while (low <= high) {  int mid = low + (high - low) / 2; // Prevent integer overflow  if (arr[mid] == target) {  return mid;  } else if (arr[mid] < target) {  low = mid + 1;  } else {  high = mid - 1;  }  }  return -1; // Not found  } |

**Comparison Between Linear Search O(n) and O(logn):**

For small input sizes, the difference between O(n) and O(log n) might be negligible.

As the input size grows, O(log n) algorithms become significantly faster than O(n) algorithms.

|  | A | B | C | D |
| --- | --- | --- | --- | --- |
| 1 | Time Complexity | Growth Rate | Example Algorithm | Faster for Larger Inputs? |
| 2 | O(n) | Linear | Linear search | No |
| 3 | O(log n) | Logarithmic | Binary search | Yes |

* Hence Binary Search is better than linear search for big input size.

# ***Sorting Algorithms***

1. Bubble Sort: Compares adjacent elements, swapping if they're out of order. Repeats until no swaps occur.
2. Selection Sort: Finds the minimum element, swaps it with the first, repeats for remaining elements.
3. Insertion Sort: Inserts each element into its correct position in a sorted sub-array.
4. Merge Sort: Divides the array in half, sorts each half recursively, then merges them.
5. Quicksort: Chooses a pivot element, partitions the array around it, and sorts the sub-arrays recursively.

# Bubble Sort

Algorithm:

1. Initialization: Set up a loop to iterate through the array n-1 times (total elements - 1).

2. Inner Loop: In each iteration of the outer loop, set up another loop to iterate through the array from index 0 to n-i-2 (excluding the last i elements already considered).

3. Comparison & Swap: Compare the current element arr[j] with its next element arr[j+1]. If they are in the wrong order (arr[j] > arr[j+1]), do the following :

Swap their positions using a temporary

variable: temp = arr[j];

arr[j] = arr[j+1];

arr[j+1] = temp;

4. Optimization (Optional): In each inner loop iteration, if no swaps occur (meaning the sub-array is already sorted), you can break out of the inner loop early as the remaining comparisons will be redundant. This optimization reduces unnecessary comparisons in partially sorted arrays.

5. Outer Loop Iteration: After each inner loop, the largest element in the considered sub-array has "bubbled up" to its final position at the end.

6. Termination: Repeat steps 2-5 until the outer loop completes (no swaps occur in the entire array).

7. Sorted Array: At the end, the array arr will be sorted in ascending order (largest elements at the end).

Time Complexity  
The time complexity of Bubble Sort is O(n^2) in both the worst-case and average-case scenarios. Here's why,

Inner Loop: Each iteration of the outer loop involves an inner loop that iterates through the array (except for the last i elements already sorted). In the worst-case and average-case scenarios, this inner loop runs n-1 times (total elements - 1) for each outer loop iteration.

Within the inner loop, comparisons and potential swaps occur, each taking constant time (ignoring the overhead of the swap operation).

Outer Loop : The outer loop iterates n-1 times (total elements - 1) to consider all elements and ensure they reach their sorted positions.  
  
Complexity Breakdown:

Inner loop: (n-1) comparisons/swaps per iteration

Outer loop: n-1 iterations

Total: (n-1) \* (n-1) = n^2 - 2n + 1 ≈ O(n^2) (ignoring constant terms)

Therefore, the time complexity of Bubble Sort is O(n^2), indicating that the number of comparisons and swaps grows quadratically with the input size. This means that the algorithm becomes significantly slower for larger datasets compared to sorting algorithms with lower time complexity like Merge Sort (O(n log n)) or Quick Sort (average O(n log n)).

Code:

|  |
| --- |
| C code  #include <stdio.h>  void bubbleSort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  int main() {  int arr[] = {20, 30, 40, 6, 4, 9, 10};  int n = sizeof(arr) / sizeof(arr[0]);  bubbleSort(arr, n);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Java Code  class BubbleSort {  public static void bubbleSort(int[] arr) {  int n = arr.length;  for (int i = 0; i < n - 1; i++) {  for (int j = 0; j < n - i - 1; j++) {  if (arr[j] > arr[j + 1]) {  int temp = arr[j];  arr[j] = arr[j + 1];  arr[j + 1] = temp;  }  }  }  }  public static void main(String[] args) {  int[] arr = {20, 30, 40, 6, 4, 9, 10};  bubbleSort(arr);  System.out.println("Sorted array: ");  for (int i : arr) {  System.out.print(i + " ");  }  }  } |

# 2> Selection Sort

Algorithm:

1. Initialization: Start with the first element (i = 0).
2. Find minimum: Iterate through the remaining elements (j = i + 1 to n-1).

* Keep track of the index of the minimum element (min\_index = i).
* If the current element (arr[j]) is less than the element at the min\_index, update min\_index to j.

1. Swap: Swap the element at the current index (i) with the element at the min\_index.
2. Increment: Increment i and repeat steps 2-3 until all elements have been considered (i = n-1).

Time Complexity:

* Worst-case and average-case: O(n^2)
* Best-case: O(n)

**Justification:**

* The outer loop iterates n-1 times (total elements - 1).
* In each iteration, the inner loop iterates through the remaining elements (n-i-1 times) to find the minimum.
* This results in a total of (n-1) \* (n-i-1) ≈ n^2 comparisons in the worst-case and average-case scenarios.
* In the best-case scenario (array already sorted), the inner loop only runs once per iteration, resulting in O(n) complexity.

Code:

|  |
| --- |
| C code  #include <stdio.h>  void selectionSort(int arr[], int n) {  for (int i = 0; i < n - 1; i++) {  int min\_index = i;  for (int j = i + 1; j < n; j++) {  if (arr[j] < arr[min\_index]) {  min\_index = j;  }  }  if (min\_index != i) {  int temp = arr[min\_index];  arr[min\_index] = arr[i];  arr[i] = temp;  }  }  }  int main() {  int arr[] = {20, 30, 40, 6, 4, 9, 10};  int n = sizeof(arr) / sizeof(arr[0]);  selectionSort(arr, n);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Java Program  public class SelectionSort {  public static void selectionSort(int[] arr) {  int n = arr.length;  for (int i = 0; i < n - 1; i++) {  int min\_index = i;  for (int j = i + 1; j < n; j++) {  if (arr[j] < arr[min\_index]) {  min\_index = j;  }  }  if (min\_index != i) {  int temp = arr[min\_index];  arr[min\_index] = arr[i];  arr[i] = temp;  }  }  }  public static void main(String[] args) {  int[] arr = {20, 30, 40, 6, 4, 9, 10};  selectionSort(arr);  System.out.println("Sorted array: ");  for (int i : arr) {  System.out.print(i + " ");  }  }  } |

# Insertion Sort

Algorithm:

1. Initialization: Start with the second element (i = 1).
2. Insertion: Iterate through the sorted sub-array (elements before the current index i).
3. Compare the current element (arr[i]) with each element in the sub-array (j = i - 1).
4. If the current element is smaller (arr[i] < arr[j]), shift all elements greater than it one position to the right.
5. Insert the current element at the correct position (arr[j + 1] = arr[i]).
6. Increment: Increment i and repeat steps 2-3 until all elements have been considered (i = n-1).

Time Complexity:

* Worst-case and average-case: O(n^2)
* Best-case: O(n)

Justification:

* The outer loop iterates n-1 times (total elements - 1).
* In the worst-case and average-case scenarios, the inner loop iterates through the entire sorted sub-array (j from i-1 to 0) in each iteration.
* This results in a total of (n-1) \* (i) ≈ n^2 comparisons and shifts in the worst-case and average-case scenarios.
* In the best-case scenario (array already sorted), the inner loop only runs once per iteration, resulting in O(n) complexity.

Code :

|  |
| --- |
| C code  #include <stdio.h>  void insertionSort(int arr[], int n) {  for (int i = 1; i < n; i++) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  }  }  int main() {  int arr[] = {20, 30, 40, 6, 4, 9, 10};  int n = sizeof(arr) / sizeof(arr[0]);  insertionSort(arr, n);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Java Code  public class InsertionSort {  public static void insertionSort(int[] arr) {  int n = arr.length;  for (int i = 1; i < n; i++) {  int key = arr[i];  int j = i - 1;  while (j >= 0 && arr[j] > key) {  arr[j + 1] = arr[j];  j--;  }  arr[j + 1] = key;  }  }  public static void main(String[] args) {  int[] arr = {20, 30, 40, 6, 4, 9, 10};  insertionSort(arr);  System.out.println("Sorted array: ");  for (int i : arr) {  System.out.print(i + " ");  }  }  } |

# **4>Merge Sort:**

Algorithm:

Divide: Recursively divide the array into two halves until each sub-array contains only one element (base case).

Conquer: Conquer by merging the two halves back together in sorted order.

Create a temporary array to store the merged elements.

Compare the elements at the beginning of each sub-array and copy the smaller one to the temporary array.

Repeat until both sub-arrays are empty.

Copy the remaining elements from either sub-array (if any) to the temporary array.

Combine: Copy the sorted elements from the temporary array back to the original array.

Time Complexity:

Worst-case, average-case, and best-case: O(n log n)

Justification:

Each recursive call divides the array in half, leading to a logarithmic relationship between the number of calls and the input size.

The merging process takes linear time (O(n)) in each call, as it involves comparing and copying elements.

Combining these factors results in a total time complexity of O(n log n), indicating efficient performance even for large datasets.

|  |
| --- |
| C code  #include <stdio.h>  void merge(int arr[], int l, int m, int r) {  int i, j, k;  int n1 = m - l + 1;  int n2 = r - m;  int L[n1], R[n2];  for (i = 0; i < n1; i++)  L[i] = arr[l + i];  for (j = 0; j < n2; j++)  R[j] = arr[m + 1 + j];  i = 0;  j = 0;  k = l;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  } else {  arr[k] = R[j];  j++;  }  k++;  }  while (i < n1) {  arr[k] = L[i];  i++;  k++;  }  while (j < n2) {  arr[k] = R[j];  j++;  k++;  }  }  void mergeSort(int arr[], int l, int r) {  if (l < r) {  int m = l + (r - l) / 2;  mergeSort(arr, l, m);  mergeSort(arr, m + 1, r);  merge(arr, l, m, r);  }  }  int main() {  int arr[] = {20, 30, 40, 6, 4, 9, 10};  int n = sizeof(arr) / sizeof(arr[0]);  mergeSort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Java Code  public class MergeSort {  public static void merge(int[] arr, int l, int m, int r) {  int n1 = m - l + 1;  int n2 = r - m;  int[] L = new int[n1];  int[] R = new int[n2];  for (int i = 0; i < n1; i++) {  L[i] = arr[l + i];  }  for (int j = 0; j < n2; j++) {  R[j] = arr[m + 1 + j];  }  // Merge the arrays back together  int i = 0, j = 0, k = l;  while (i < n1 && j < n2) {  if (L[i] <= R[j]) {  arr[k] = L[i];  i++;  } else {  arr[k] = R[j];  j++;  }  k++;  }  // Copy remaining elements from L[]  while (i < n1) {  arr[k] = L[i];  i++;  k++;  }  // Copy remaining elements from R[]  while (j < n2) {  arr[k] = R[j];  j++;  k++;  }  }  public static void mergeSort(int[] arr, int l, int r) {  if (l < r) {  // Same as C code  int m = l + (r - l) / 2;  mergeSort(arr, l, m);  mergeSort(arr, m + 1, r);  merge(arr, l, m, r);  }  }  public static void main(String[] args) {  int[] arr = {20, 30, 40, 6, 4, 9, 10};  mergeSort(arr, 0, arr.length - 1);  System.out.println("Sorted array: ");  for (int i : arr) {  System.out.print(i + " ");  }  }  } |

# **Quick Sort:**

Algorithm:

1. Choose a pivot: Select an element from the array as a pivot (often the last element).
2. Partition: Rearrange the array so that:

* Elements less than the pivot are placed to its left.
* Elements greater than the pivot are placed to its right.
* The pivot is in its final sorted position.

1. Recurse: Recursively apply the quick sort algorithm to the sub-arrays to the left and right of the pivot.

Time Complexity:

* Best-case and average-case: O(n log n)
* Worst-case: O(n^2)
* Justification:
* The average-case time complexity is O(n log n) because each partitioning step divides the array into two smaller sub-arrays, leading to a logarithmic relationship between the number of steps and the input size.
* The worst-case scenario occurs when the pivot consistently divides the array into highly unbalanced sub-arrays (e.g., always choosing the largest or smallest element as pivot). This leads to a quadratic number of comparisons.

Code:

|  |
| --- |
| C code  #include <stdio.h>  int partition(int arr[], int low, int high) {  int pivot = arr[high];  int i = (low - 1);  for (int j = low; j < high; j++) {  if (arr[j] <= pivot) {  i++;  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }  }  int temp = arr[i + 1];  arr[i + 1] = arr[high];  arr[high] = temp;  return (i + 1);  }  void quickSort(int arr[], int low, int high) {  if (low < high) {  int pi = partition(arr, low, high);  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }  int main() {  int arr[] = {20, 30, 40, 6, 4, 9, 10};  int n = sizeof(arr) / sizeof(arr[0]);  quickSort(arr, 0, n - 1);  printf("Sorted array: ");  for (int i = 0; i < n; i++) {  printf("%d ", arr[i]);  }  printf("\n");  return 0;  } |
| Java Code  public class QuickSort {    public static void quickSort(int[] arr, int low, int high) {  if (low < high) {  // Partition the array around the pivot  int pi = partition(arr, low, high);  // Recursively sort elements before and after partition  quickSort(arr, low, pi - 1);  quickSort(arr, pi + 1, high);  }  }    public static int partition(int[] arr, int low, int high) {  int pivot = arr[high]; // Choose the last element as the pivot  int i = (low - 1);  for (int j = low; j <= high - 1; j++) {  // Move elements smaller than pivot to the left  if (arr[j] <= pivot) {  i++;  swap(arr, i, j);  }  }  swap(arr, i + 1, high);  return (i + 1);  }    public static void swap(int[] arr, int i, int j) {  int temp = arr[i];  arr[i] = arr[j];  arr[j] = temp;  }    public static void main(String[] args) {  int[] arr = {20, 30, 40, 6, 4, 9, 10};  quickSort(arr, 0, arr.length - 1);  System.out.println("Sorted array: ");  for (int i : arr) {  System.out.print(i + " ");  }  }  } |

# Time Complexity Comparison:

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Algorithm | Best-Case Time Complexity | Average-Case Time Complexity | Worst-Case Time Complexity |
| 1 | Bubble Sort | O(n) | O(n^2) | O(n^2) |
| 2 | Insertion Sort | O(n) | O(n^2) | O(n^2) |
| 3 | Selection Sort | O(n^2) | O(n^2) | O(n^2) |
| 4 | Merge Sort | O(n log n) | O(n log n) | O(n log n) |
| 5 | Quick Sort | O(n log n) | O(n log n) | O(n^2) |

Key factors influencing performance:

* Input size: For small arrays (up to a few hundred elements), Insertion Sort and Selection Sort might be faster than Merge Sort and Quick Sort due to their lower overhead. However, as the input size increases, Merge Sort and Quick Sort dominate due to their asymptotic growth.
* Data distribution: Quick Sort's performance is particularly sensitive to the pivot element's choice. If the pivot consistently divides the array into highly unbalanced sub-arrays, it can degrade to O(n^2) complexity. Merge Sort is stable and generally performs well regardless of the input data distribution.

Choosing the right algorithm:

* For small datasets and simplicity, Insertion Sort or Selection Sort might be suitable.
* For larger datasets and efficiency, Merge Sort or Quick Sort are generally preferred. However, consider the following factors for specific scenarios:
* Guaranteed stability: If maintaining the relative order of equal elements is crucial, use Merge Sort.
* Parallel processing: Merge Sort's divide-and-conquer approach is well-suited for parallel architectures.
* Worst-case performance: If avoiding the worst-case O(n^2) time complexity is critical, Merge Sort is safer than Quick Sort.
* Merge Sort is the most optimized in every case of time complexity.

***Practice Questions:***

1. ***Fibonacci number Series:*** The Fibonacci series starts with 0 and 1, where each subsequent number is the sum of the two preceding numbers.

Here are the first 10 Fibonacci numbers:0, 1, 1, 2, 3, 5, 8, 13, 21, 34

Explanation: The series starts with 0 and 1 because there's no "previous" number for them. Each number is calculated by adding the two numbers before it. So, the third number is 1 + 1 = 2, the fourth is 2 + 1 = 3, and so on.

|  |
| --- |
| C code  #include <stdio.h>  int fibonacci(int n, int \*arr) {  if (n <= 1) {  return n;  }  if (arr[n] != -1) {  return arr[n];  } else {  arr[n] = fibonacci(n - 1, arr) + fibonacci(n - 2, arr);  return arr[n];  }  }  int main() {  int n = 10; // Example input  int arr[n + 1];  for (int i = 0; i <= n; i++) {  arr[i] = -1; // Initialize array with -1 to indicate uncalculated values  }  int result = fibonacci(n, arr);  printf("The %dth Fibonacci number is %d\n", n, result);  return 0;  } |
| Java Code  public class FibonacciOptimized {  static int[] arr = new int[100]; // Array for storing Fibonacci numbers (adjust size as needed)  static int fibonacci(int n) {  if (n <= 1) {  return n;  }  if (arr[n] != -1) {  return arr[n];  } else {  arr[n] = fibonacci(n - 1) + fibonacci(n - 2);  return arr[n];  }  }  public static void main(String[] args) {  int n = 40; // Example input  for (int i = 0; i <= n; i++) {  arr[i] = -1; // Initialize array with -1  }  int result = fibonacci(n);  System.out.println("The " + n + "th Fibonacci number is " + result);  }  } |

1. ***Binomial Coefficient:***

The binomial coefficient, denoted by nCk or n choose k, represents the number of ways to choose k elements without order from a set of n distinct elements.

Concept:

n: total number of elements

k: number of elements chosen

nCk: number of ways to choose k elements from n without order

Factorial Formula for nCk:

nCk = n! / (k! \* (n-k)!)

Code:

|  |
| --- |
| C code  #include <stdio.h>  int binomialCoefficient(int n, int k) {  if (k == 0 || k == n) {  return 1;  } else if (k < 0 || k > n) {  return 0;  } else {  // Use a 2D array to store precomputed values for efficiency  int dp[n + 1][k + 1];  // Base cases  for (int i = 0; i <= n; i++) {  dp[i][0] = 1;  }  for (int j = 0; j <= k; j++) {  dp[0][j] = 0;  }  // Fill the array using the recurrence relation  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= min(i, k); j++) {  dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1];  }  }  return dp[n][k];  }  }  int main() {  int n, k;  printf("Enter values for n and k: ");  scanf("%d %d", &n, &k);  int result = binomialCoefficient(n, k);  printf("The binomial coefficient C(%d, %d) is: %d\n", n, k, result);  return 0;  } |
| Java Code  public class BinomialCoefficient {  public static int binomialCoefficient(int n, int k) {  if (k == 0 || k == n) {  return 1;  } else if (k < 0 || k > n) {  return 0;  } else {  // Use a 2D array to store precomputed values for efficiency  int[][] dp = new int[n + 1][k + 1];  // Base cases  for (int i = 0; i <= n; i++) {  dp[i][0] = 1;  }  for (int j = 0; j <= k; j++) {  dp[0][j] = 0;  }  // Fill the array using the recurrence relation  for (int i = 1; i <= n; i++) {  for (int j = 1; j <= min(i, k); j++) {  dp[i][j] = dp[i - 1][j] + dp[i - 1][j - 1];  }  }  return dp[n][k];  }  }  public static void main(String[] args) {  int n, k;  System.out.print("Enter values for n and k: ");  try {  Scanner scanner = new Scanner(System.in);  n = scanner.nextInt();  k = scanner.nextInt();  } catch (InputMismatchException e) {  System.out.println("Invalid input. Please enter integers.");  return;  }  int result = binomialCoefficient(n, k);  System.out.println("The binomial coefficient C(" + n + ", " + k + ") is: " + result);  }  } |

3>Coin Change Problem:

Given a value N, if we want to make change for N cents, and we have infinite supply of each of S = { S1, S2, .. , Sm} valued coins, how many ways can we make the change? The order of coins doesn’t matter.

For example, for N = 4 and S = {1,2,3}, there are four solutions: {1,1,1,1},{1,1,2},{2,2},{1,3}. So output should be 4. For N = 10 and S = {2, 5, 3, 6}, there are five solutions: {2,2,2,2,2}, {2,2,3,3}, {2,2,6}, {2,3,5} and {5,5}. So the output should be 5.

|  |
| --- |
| C code  // C program for coin change problem.  #include<stdio.h>  int count( int S[], int m, int n )  {  int i, j, x, y;  // We need n+1 rows as the table is constructed  // in bottom up manner using the base case 0  // value case (n = 0)  int table[n+1][m];    // Fill the entries for 0 value case (n = 0)  for (i=0; i<m; i++)  table[0][i] = 1;  // Fill rest of the table entries in bottom  // up manner  for (i = 1; i < n+1; i++)  {  for (j = 0; j < m; j++)  {  // Count of solutions including S[j]  x = (i-S[j] >= 0)? table[i - S[j]][j]: 0;  // Count of solutions excluding S[j]  y = (j >= 1)? table[i][j-1]: 0;  // total count  table[i][j] = x + y;  }  }  return table[n][m-1];  }  // Driver program to test above function  int main()  {  int arr[] = {1, 2, 3};  int m = sizeof(arr)/sizeof(arr[0]);  int n = 4;  printf(" %d ", count(arr, m, n));  return 0;  }  //Time Complexity: O(m\*n) |

4> Subset Sum Problem:  
  
Write a program for a given set of non-negative integers and a value sum, the task is to check if there is a subset of the given set whose sum is equal to the given sum.  
  
Examples:

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 9

Output: True

Explanation: There is a subset (4, 5) with sum 9.

Input: set[] = {3, 34, 4, 12, 5, 2}, sum = 30

Output: False

Explanation: There is no subset that adds up to 30.  
  
Discussion:

For the recursive approach, there will be two cases.

1. Consider the ‘last’ element to be a part of the subset. Now the new required sum = required sum – value of ‘last’ element.
2. Don’t include the ‘last’ element in the subset. Then the new required sum = old required sum.

In both cases, the number of available elements decreases by 1.  
  
code:

|  |
| --- |
| C code  // A recursive solution for subset sum problem  #include <stdio.h>  #include <stdbool.h>  // Returns true if there is a subset  // of set[] with sum equal to given sum  bool isSubsetSum(int set[], int n, int sum)  {  // Base Cases  if (sum == 0)  return true;  if (n == 0)  return false;  // If last element is greater than sum,  // then ignore it  if (set[n - 1] > sum)  return isSubsetSum(set, n - 1, sum);  // Else, check if sum can be obtained by any  // of the following:  // (a) including the last element  // (b) excluding the last element  return isSubsetSum(set, n - 1, sum)  || isSubsetSum(set, n - 1, sum - set[n - 1]);  }  // Driver code  int main()  {  int set[] = { 3, 34, 4, 12, 5, 2 };  int sum = 9;  int n = sizeof(set) / sizeof(set[0]);  if (isSubsetSum(set, n, sum) == true)  printf("Found a subset with given sum");  else  printf("No subset with given sum");  return 0;  }  // Time Complexity: O(sum\*n)  //Auxiliary space: O(n) |

***5> Compute nCr % p***:

***Using Lucas Theorem for nCr % p:*** Lucas theorem basically suggests that the value of nCr can be computed by multiplying results of niCri where ni and ri are individual same-positioned digits in base p representations of n and r respectively.

|  |
| --- |
| Java Code  // A Lucas Theorem based solution to compute nCr % p  class Main{  // Returns nCr % p. In this Lucas Theorem based program,  // this function is only called for n < p and r < p.  static int nCrModpDP(int n, int r, int p)  {  // The array C is going to store last row of  // pascal triangle at the end. And last entry  // of last row is nCr  int[] C=new int[r+1];  C[0] = 1; // Top row of Pascal Triangle  // One by constructs remaining rows of Pascal  // Triangle from top to bottom  for (int i = 1; i <= n; i++)  {  // Fill entries of current row using previous  // row values  for (int j = Math.min(i, r); j > 0; j--)  // nCj = (n-1)Cj + (n-1)C(j-1);  C[j] = (C[j] + C[j-1])%p;  }  return C[r];  }  // Lucas Theorem based function that returns nCr % p  // This function works like decimal to binary conversion  // recursive function. First we compute last digits of  // n and r in base p, then recur for remaining digits  static int nCrModpLucas(int n, int r, int p)  {  // Base case  if (r==0)  return 1;  // Compute last digits of n and r in base p  int ni = n%p;  int ri = r%p;  // Compute result for last digits computed above, and  // for remaining digits. Multiply the two results and  // compute the result of multiplication in modulo p.  return (nCrModpLucas(n/p, r/p, p) \*  nCrModpDP(ni, ri, p)) % p; // Last digits of n and r  // Remaining digits  }  // Driver program  public static void main(String[] args)  {  int n = 1000, r = 900, p = 13;  System.out.println("Value of nCr % p is "+nCrModpLucas(n, r, p));  }  } |

# ***6>Cutting a Rod:***

Given a rod of length n inches and an array of prices that contains prices of all pieces of size smaller than n. Determine the maximum value obtainable by cutting up the rod and selling the pieces. For example,

* if length of the rod is 8 and the values of different pieces are given as following, then the maximum obtainable value is 22 (by cutting in two pieces of lengths 2 and 6)

length | 1 2 3 4 5 6 7 8

--------------------------------------------

price | 1 5 8 9 10 17 17 20

* And if the prices are as following, then the maximum obtainable value is 24 (by cutting in eight pieces of length 1)

length | 1 2 3 4 5 6 7 8

--------------------------------------------

price | 3 5 8 9 10 17 17 20

|  |
| --- |
| C code  // A Naive recursive solution for Rod cutting problem  #include <limits.h>  #include <stdio.h>  // A utility function to get the maximum of two integers  int max(int a, int b) { return (a > b) ? a : b; }  /\* Returns the best obtainable price for a rod of length n and  price[] as prices of different pieces \*/  int cutRod(int price[], int n)  {  if (n <= 0)  return 0;  int max\_val = INT\_MIN;  // Recursively cut the rod in different pieces and compare different  // configurations  for (int i = 0; i < n; i++)  max\_val = max(max\_val, price[i] + cutRod(price, n - i - 1));  return max\_val;  }  /\* Driver program to test above functions \*/  int main()  {  int arr[] = { 1, 5, 8, 9, 10, 17, 17, 20 };  int size = sizeof(arr) / sizeof(arr[0]);  printf("Maximum Obtainable Value is %d", cutRod(arr, size));  getchar();  return 0;  } |

# ***7>Maximum size square sub-matrix with all 1s:***

program for a given binary matrix, the task is to find out the maximum size square sub-matrix with all 1s.

***Approach:***

Let the given binary matrix be M[R][C]. The idea of the algorithm is to construct an auxiliary size matrix S[][] in which each entry S[i][j] represents the size of the square sub-matrix with all 1s including M[i][j] where M[i][j] is the rightmost and bottom-most entry in sub-matrix.

***Step-by-step approach:***

* Construct a sum matrix S[R][C] for the given M[R][C].
  + Copy first row and first columns as it is from M[][] to S[][]
  + For other entries, use the following expressions to construct S[][]
    - If M[i][j] is 1 then
      * S[i][j] = min(S[i][j-1], S[i-1][j], S[i-1][j-1]) + 1
    - Else If M[i][j] is 0 then
      * S[i][j] = 0
* Find the maximum entry in S[R][C]
* Using the value and coordinates of maximum entry in S[i], print sub-matrix of M[][]

Code:

|  |
| --- |
| C code  // C code for Maximum size square  // sub-matrix with all 1s  #include <stdio.h>  #define bool int  #define R 6  #define C 5  void printMaxSubSquare(bool M[R][C])  {  int i, j;  int S[R][C];  int max\_of\_s, max\_i, max\_j;  /\* Set first column of S[][]\*/  for (i = 0; i < R; i++)  S[i][0] = M[i][0];  /\* Set first row of S[][]\*/  for (j = 0; j < C; j++)  S[0][j] = M[0][j];  /\* Construct other entries of S[][]\*/  for (i = 1; i < R; i++) {  for (j = 1; j < C; j++) {  if (M[i][j] == 1)  S[i][j] = min(S[i][j - 1], S[i - 1][j],  S[i - 1][j - 1])  + 1;  else  S[i][j] = 0;  }  }  /\* Find the maximum entry, and indexes of maximum entry  in S[][] \*/  max\_of\_s = S[0][0];  max\_i = 0;  max\_j = 0;  for (i = 0; i < R; i++) {  for (j = 0; j < C; j++) {  if (max\_of\_s < S[i][j]) {  max\_of\_s = S[i][j];  max\_i = i;  max\_j = j;  }  }  }  printf("Maximum size sub-matrix is: \n");  for (i = max\_i; i > max\_i - max\_of\_s; i--) {  for (j = max\_j; j > max\_j - max\_of\_s; j--) {  printf("%d ", M[i][j]);  }  printf("\n");  }  }  /\* UTILITY FUNCTIONS \*/  /\* Function to get minimum of three values \*/  int min(int a, int b, int c)  {  int m = a;  if (m > b)  m = b;  if (m > c)  m = c;  return m;  }  /\* Driver function to test above functions \*/  int main()  {  bool M[R][C] = { { 0, 1, 1, 0, 1 }, { 1, 1, 0, 1, 0 },  { 0, 1, 1, 1, 0 }, { 1, 1, 1, 1, 0 },  { 1, 1, 1, 1, 1 }, { 0, 0, 0, 0, 0 } };  printMaxSubSquare(M);  getchar();  } |

# ***Longest Common Substring (Space optimized DP solution):***

Given two strings ‘X’ and ‘Y’, find the length of longest common substring. Expected space complexity is linear.

Examples :

Input : X = "abcdxyz", Y = "xyzabcd"

Output : 4

The longest common substring is "abcd" and is of

length 4.  
  
both xyz and abcd are substring to the 1st string variable X of 2nd string variable Y but we will take abcd because it is the largest.  
***code:***

|  |
| --- |
| // Space optimized CPP implementation of  // longest common substring.  import java.io.\*;  import java.util.\*;  public class Main{    // Function to find longest  // common substring.  static int LCSubStr(String X, String Y)  {    // Find length of both the strings.  int m = X.length();  int n = Y.length();    // Variable to store length of longest  // common substring.  int result = 0;    // Matrix to store result of two  // consecutive rows at a time.  int [][]len = new int[2][n];    // Variable to represent which row of  // matrix is current row.  int currRow = 0;    // For a particular value of  // i and j, len[currRow][j]  // stores length of longest  // common substring in string  // X[0..i] and Y[0..j].  for (int i = 0; i < m; i++) {  for (int j = 0; j < n; j++) {  if (i == 0 || j == 0) {  len[currRow][j] = 0;  }  else if (X.charAt(i - 1) ==  Y.charAt(j - 1))  {  len[currRow][j] =  len[(1 - currRow)][(j - 1)]  + 1;  result = Math.max(result,  len[currRow][j]);  }  else  {  len[currRow][j] = 0;  }  }    // Make current row as previous  // row and previous row as  // new current row.  currRow = 1 - currRow;  }    return result;  }    // Driver Code  public static void main(String args[])  {  String X = "abcdxyz";  String Y = "xyzabcd";    System.out.print(LCSubStr(X, Y));  }  }  // Time Complexity: O(m\*n)  //Auxiliary Space: O(n) |

# ***Count ways to reach the nth stair using step 1, 2 or 3:***

A child is running up a staircase with n steps and can hop either 1 step, 2 steps, or 3 steps at a time. Implement a method to count how many possible ways the child can run up the stairs.

Examples:

Input : 4

Output : 7

Explanation:

Below are the seven ways

1 step + 1 step + 1 step + 1 step

1 step + 2 step + 1 step

2 step + 1 step + 1 step

1 step + 1 step + 2 step

2 step + 2 step

3 step + 1 step

1 step + 3 step

Input : 3

Output : 4

Explanation:

Below are the four ways

1 step + 1 step + 1 step

1 step + 2 step

2 step + 1 step

3 step  
  
***code:***

|  |
| --- |
| Java Code  // Program to find n-th stair  // using step size 1 or 2 or 3.  import java.lang.\*;  import java.util.\*;  public class Main {  // A recursive function used by countWays  public static int countWays(int n)  {  int[] res = new int[n + 1];  res[0] = 1;  res[1] = 1;  res[2] = 2;  for (int i = 3; i <= n; i++)  res[i] = res[i - 1] + res[i - 2] + res[i - 3];  return res[n];  }  // Driver function  public static void main(String argc[])  {  int n = 4;  System.out.println(countWays(n));  }  } |

|  |
| --- |
| Output  7 |

|  |
| --- |
| Working:  1 -> 1 -> 1 -> 1  1 -> 1 -> 2  1 -> 2 -> 1  1 -> 3  2 -> 1 -> 1  2 -> 2  3 -> 1  So Total ways: 7 |

|  |
| --- |
| Complexity Analysis:  Time Complexity: O(n).  Only one traversal of the array is needed. So Time Complexity is O(n).  Space Complexity: O(n). |

# ***Count all possible paths from top left to bottom right of a mXn matrix:***

Given a 2D matrix of dimension m✕n, the task is to print all the possible paths from the top left corner to the bottom right corner in a 2D matrix with the constraints that from each cell you can either move to right or down only.

Examples :

Input: [[1,2,3],

[4,5,6]]

Output: [[1,4,5,6],

[1,2,5,6],

[1,2,3,6]]

Input: [[1,2],

[3,4]]

Output: [[1,2,4],

[1,3,4]]

***Code:***

|  |
| --- |
| ***Java Code***  import java.util.ArrayList;  import java.util.List;  public class MatrixPaths {  // To store the matrix dimensions  static int M, N;  // Function to print the path taken to reach the destination  static void printPath(List<Integer> path) {  for (int i : path) {  System.out.print(i + ", ");  }  System.out.println();  }  // Function to find all possible paths in the matrix from the top-left cell to the bottom-right cell  static void findPaths(int[][] arr, List<Integer> path, int i, int j) {  // If it's the bottom-right cell, print the path  if (i == M - 1 && j == N - 1) {  path.add(arr[i][j]);  printPath(path);  path.remove(path.size() - 1);  return;  }  // Boundary cases: Check if we are out of the matrix  if (i < 0 || i >= M || j < 0 || j >= N) {  return;  }  // Include the current cell in the path  path.add(arr[i][j]);  // Move right in the matrix  if (j + 1 < N) {  findPaths(arr, path, i, j + 1);  }  // Move down in the matrix  if (i + 1 < M) {  findPaths(arr, path, i + 1, j);  }  // Backtrack: Remove the current cell from the current path  path.remove(path.size() - 1);  }  // Driver code  public static void main(String[] args) {  // Input matrix  int[][] arr = {  {1, 2, 3},  {4, 5, 6},  {7, 8, 9}  };  // To store the path  List<Integer> path = new ArrayList<>();  // Starting cell (0, 0)  int i = 0, j = 0;  M = arr.length;  N = arr[0].length;  // Function call  findPaths(arr, path, i, j);  }  } |

|  |
| --- |
| Output  1, 2, 3, 6, 9,  1, 2, 5, 6, 9,  1, 2, 5, 8, 9,  1, 4, 5, 6, 9,  1, 4, 5, 8, 9,  1, 4, 7, 8, 9, |

Time Complexity : O(2^(N\*M))

Auxiliary space : O(N + M), where M and N are dimension of matrix.

# ***Unique paths in a Grid with Obstacles:***

Given a grid of size m \* n, let us assume you are starting at (1, 1) and your goal is to reach (m, n). At any instance, if you are on (x, y), you can either go to (x, y + 1) or (x + 1, y).

Now consider if some obstacles are added to the grids. How many unique paths would there be?

An obstacle and space are marked as 1 and 0 respectively in the grid.

Examples:

Input:

[[0, 0, 0],

[0, 1, 0],

[0, 0, 0]]

Output: 2

There is only one obstacle in the middle.

As Per Problem tell us that we can move in two ways can either go to (x, y + 1) or (x + 1, y). So we just calculate all possible outcome in both ways and store in 2d dp vector and return the dp[0][0] i.e all possible ways that takes you from (0,0) to (n-1,m-1);

***Code:***

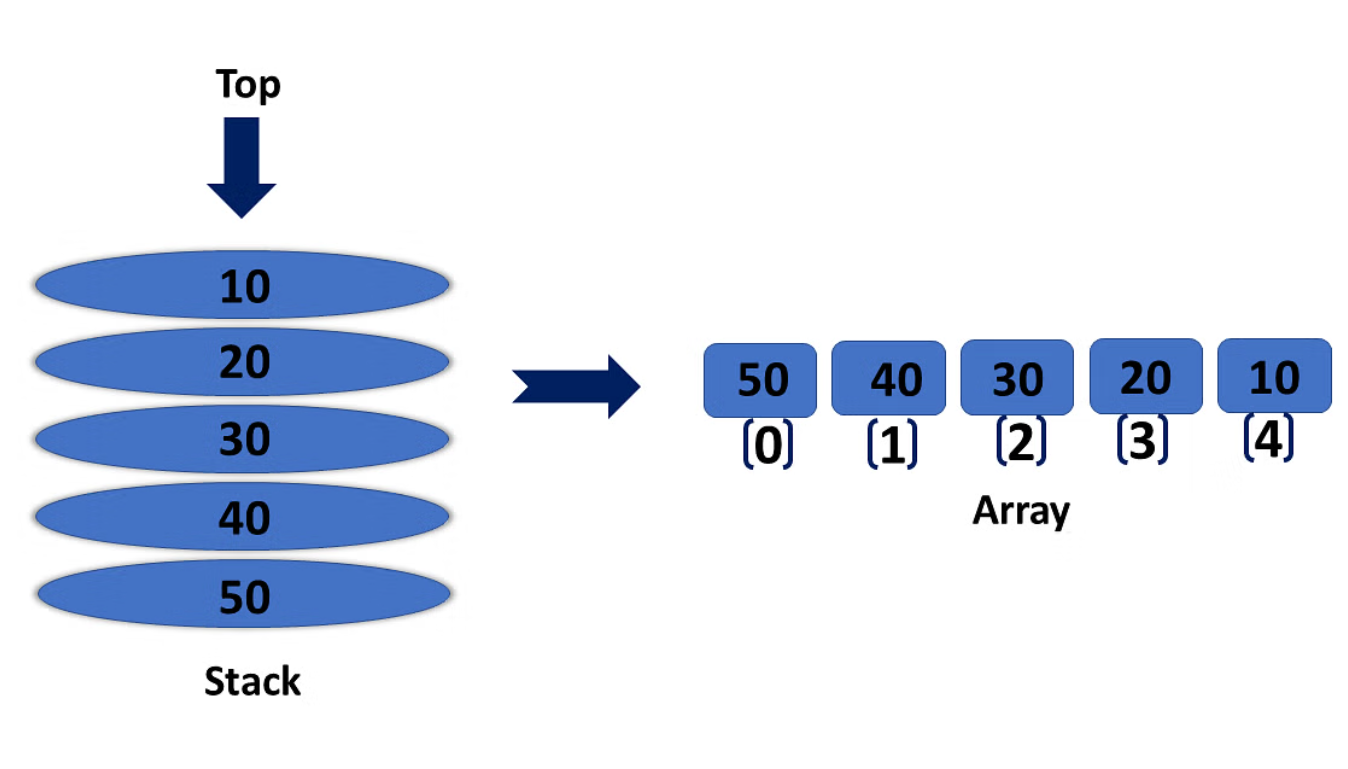
|  |
| --- |
| Java Code  import java.util.\*;  class Main {  static int n, m;  static int path(int[][] dp, int[][] grid, int i, int j)  {  if (i < n && j < m && grid[i][j] == 1)  return 0;  if (i == n - 1 && j == m - 1)  return 1;  if (i >= n || j >= m)  return 0;  if (dp[i][j] != -1)  return dp[i][j];  int left = path(dp, grid, i + 1, j);  int right = path(dp, grid, i, j + 1);  return dp[i][j] = left + right;  }  static int uniquePathsWithObstacles(int[][] grid)  {  n = grid.length;  m = grid[0].length;  if (n == 1 && m == 1 && grid[0][0] == 0)  return 1;  if (n == 1 && m == 1 && grid[0][0] == 1)  return 0;  int[][] dp = new int[n][m];  for (int i = 0; i < n; i++) {  for (int j = 0; j < m; j++) {  dp[i][j] = -1;  }  }  path(dp, grid, 0, 0);  if (dp[0][0] == -1)  return 0;  return dp[0][0];  }  public static void main(String[] args)  {  int[][] v  = { { 0, 0, 0 }, { 0, 1, 0 }, { 0, 0, 0 } };  System.out.println(uniquePathsWithObstacles(v));  }  } |

|  |
| --- |
| Output  2 |

Time Complexity: O(M\*N),For traversing all possible ways.

Auxiliary Space: O(M\*N),For storing in 2D Dp Vector.

# ***Stack Data Structures***



Stack Data Structure: A Powerful Tool for Sequential Access

A stack, in computer science, is a linear data structure that follows the “**Last In, First Out (LIFO)**” or “**First In, Last Out (FILO)**” principle. It operates much like a stack of plates in a cafeteria, where you can only add or remove plates from the top. This principle translates to the following operations:

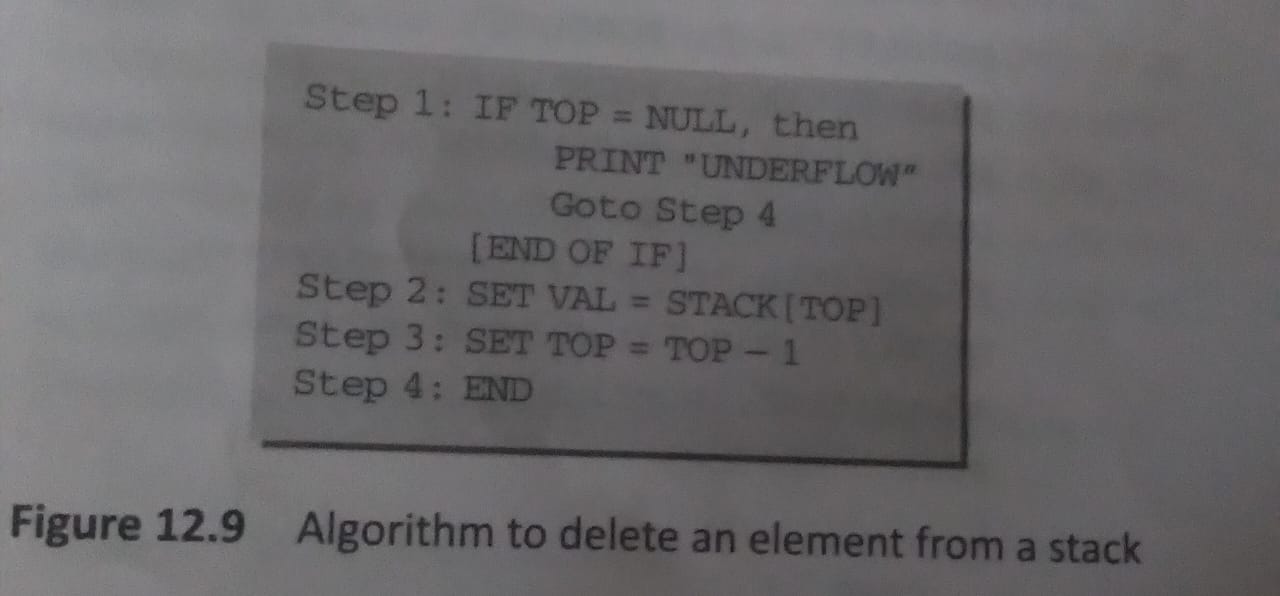
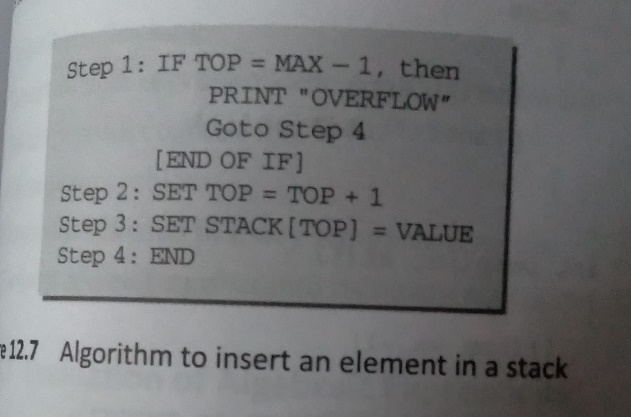
* Push: Adds an element to the top of the stack.
* Pop: Removes the element from the top of the stack and returns it.
* Peek: Returns the element at the top of the stack without removing it.
* IsEmpty: Checks if the stack is empty.
* IsFull: Checks if the stack is full (applicable only for fixed-size stacks).
* **Why Use a Stack? When Arrays Aren't Enough:**

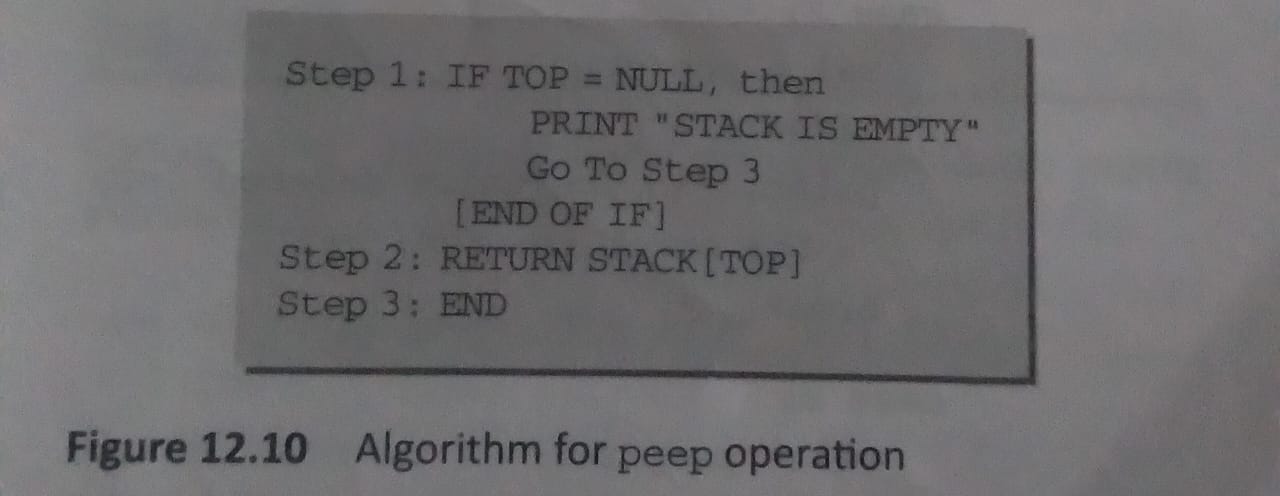
While arrays provide essential data storage capabilities, they lack the LIFO functionality offered by stacks. Here's where stacks excel:

* **Undo/Redo Operations**: Stacks enable easy implementation of undo/redo functionality in applications like text editors, allowing users to reverse actions.
* **Function Calls and Expression Evaluation**: Stacks play a crucial role in managing function calls in compilers and interpreters. They hold arguments and local variables, ensuring proper execution and memory management.
* **Backtracking Algorithms**: Backtracking algorithms, used in maze solving and problem-solving, often benefit from stacks to retrace steps and explore alternative paths.
* **Expression Conversion:** Stacks are used to convert infix expressions (e.g., (a + b) \* c) into postfix (e.g., a b + c \*) or prefix (e.g., \* + a b c) notations, facilitating efficient evaluation.
* **Syntax Analysis**: Stacks are used in parsing algorithms for languages and expressions, verifying their grammatical correctness.

***Anatomy of a Stack (Top, Bottom, and Code Implementations):***

* **Top:** The top of the stack refers to the element that will be accessed for push, pop, or peek operations. It's the most recently added element.
* **Bottom:** The bottom represents the base of the stack, conceptualized as the "bottom plate" in the stack analogy. However, it's not directly accessible in most implementation.





* **Code Implementations in C (CRUD Operations, Time Complexities):**

|  |
| --- |
| #include <stdio.h>  #include <stdbool.h>  #define MAX\_SIZE 100  int stack[MAX\_SIZE];  int top = -1;  bool isEmpty() {  return top == -1;  }  bool isFull() {  return top == MAX\_SIZE - 1;  }  void push(int data) {  if (isFull()) {  printf("Stack Overflow\n");  return;  }  stack[++top] = data;  }  int pop() {  if (isEmpty()) {  printf("Stack Underflow\n");  return -1; // Or any default value  }  return stack[top--];  }  int peek() {  if (isEmpty()) {  printf("Stack is Empty\n");  return -1; // Or any default value  }  return stack[top];  }  int main() {  push(10);  push(20);  push(30);  printf("%d popped\n", pop()); // Output: 30  printf("Top element: %d\n", peek()); // Output: 20  return 0;  } |

* **Time Complexity:** Push, pop, peek, isEmpty, isFull : O(1) (constant time), efficient due to direct access using the top pointer.
* **Code Implementations in Java (CRUD Operations, Time Complexities):**

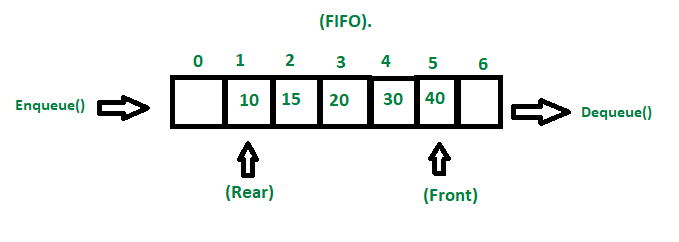
|  |
| --- |
| public class ArrayStack {  private int[] stack;  private int top; // Index of the top element  public ArrayStack(int capacity) {  this.stack = new int[capacity];  this.top = -1; // Initially empty  }  public boolean isEmpty() {  return top == -1;  }  public boolean isFull() {  return top == stack.length - 1;  }  public void push(int data) {  if (isFull()) {  throw new RuntimeException("Stack overflow"); // Or implement resize logic  }  stack[++top] = data; // Increment top first  }  public int pop() {  if (isEmpty()) {  throw new RuntimeException("Stack underflow"); // Or handle appropriately  }  return stack[top--]; // Decrement top after returning data  }  public int peek() {  if (isEmpty()) {  throw new RuntimeException("Stack is empty"); // Or handle appropriately  }  return stack[top]; // Access data without modifying top  }  public static void main(String[] args) {  ArrayStack stack = new ArrayStack(5);  stack.push(10);  stack.push(20);  stack.push(30);  System.out.println(stack.pop()); // Output: 30  System.out.println(stack.peek()); // Output: 20  }  } |

* **Time Complexity:**

isEmpty(), isFull(), push(), pop(), peek(): O(1) (constant time)

* ***Explanation:***
* **ArrayStack class:** This class represents the stack data structure.
* **capacity:** This constructor parameter specifies the initial capacity of the stack.
* **stack:** This array stores the elements of the stack.
* **top:** This variable keeps track of the index of the top element in the stack.
* main method: Demonstrates the usage of the ArrayStack class.
* **Methods:**
  + **isEmpty():** Checks if the stack is empty.
  + **isFull():** Checks if the stack is full.
  + **push(data):** Adds an element to the top of the stack. Throws an exception if full.
  + **pop():** Removes and returns the top element. Throws an exception if empty.
  + **peek():** Returns the top element without removing it. Throws an exception if empty.

# ***Queue Data Structure***



Queues: First-In, First-Out Data Structure

A queue is a linear data structure that adheres to the First-In, First-Out (FIFO) principle. It operates much like a waiting line, where the first person who joins the line (enqueues) is also the first to be served (dequeued).

* ***Why Use Queues? While Arrays and Stacks Have Their Roles:***

Stacks cannot maintain FIFO (First In, First Out) order, and arrays also lack FIFO methods until they are modified to function as queues.

**Now, why would we need to modify arrays to function as queues?**

* **Efficiency:** Arrays offer fast access to elements based on their indices. By modifying arrays to behave like queues, we can leverage this efficiency while also maintaining the FIFO order required by queues.
* **Flexibility:** Sometimes, using arrays as the underlying data structure for queues can offer more flexibility in terms of memory management and performance, especially in scenarios where we have a predefined size of elements to process.

Usage Of Queues:

* **Processing tasks sequentially:** Operating systems use queues to manage processes waiting for CPU time, ensuring fair and orderly execution.
* **Buffering data:** Queues are used to buffer data streams in networking, file processing, and other I/O (Input/Output) operations, smoothing out flow variations.

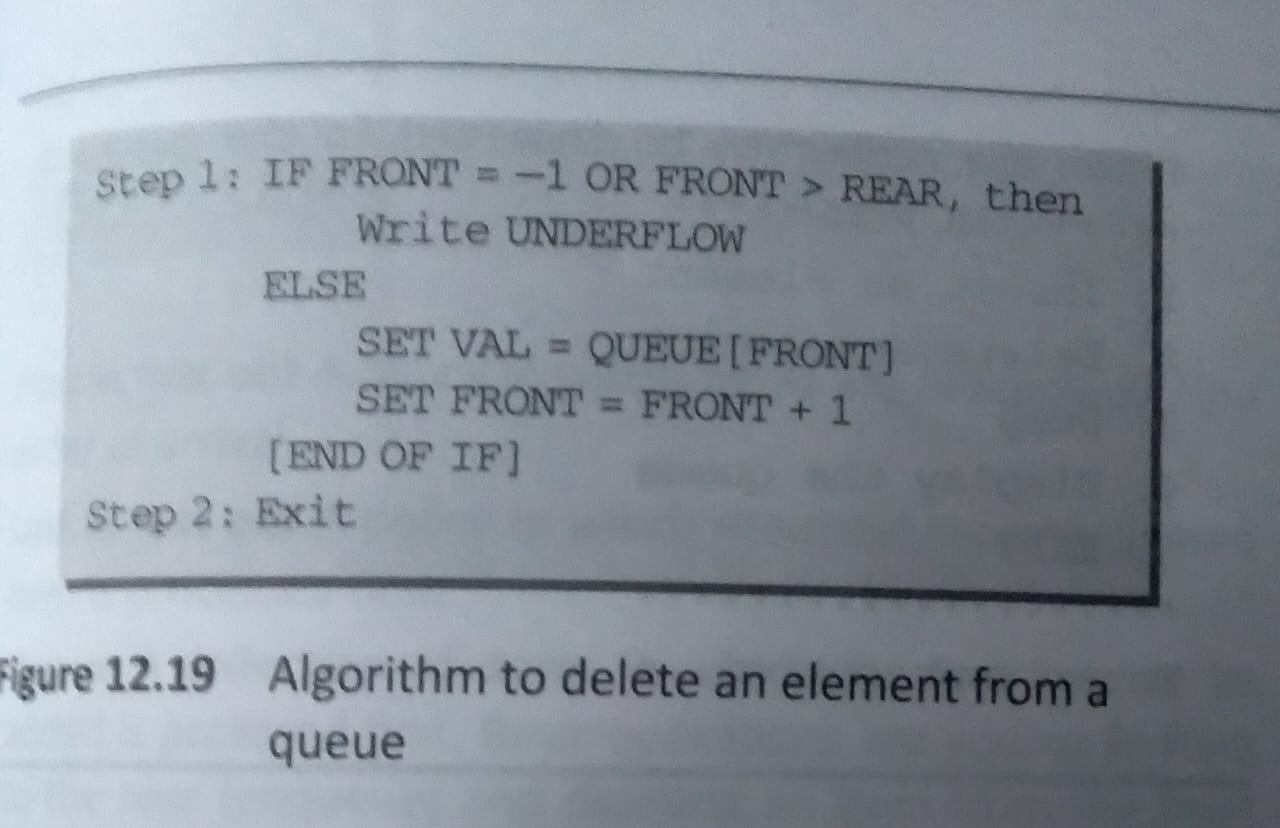
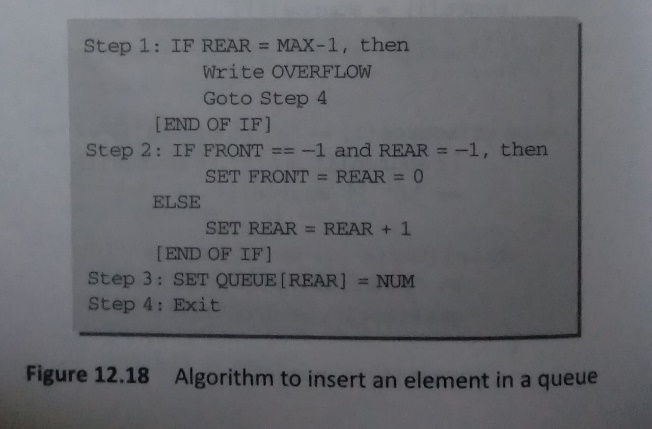
Anatomy of a Queue (Front, Rear):

* **Front:** The front of the queue refers to the element that will be removed first.
* **Rear:** The rear of the queue is where new elements are added.

# Operations:

* **Enqueue:** Adding an element to the rear of the queue.
* **Dequeue:** Removing an element from the front of the queue.

# Code:



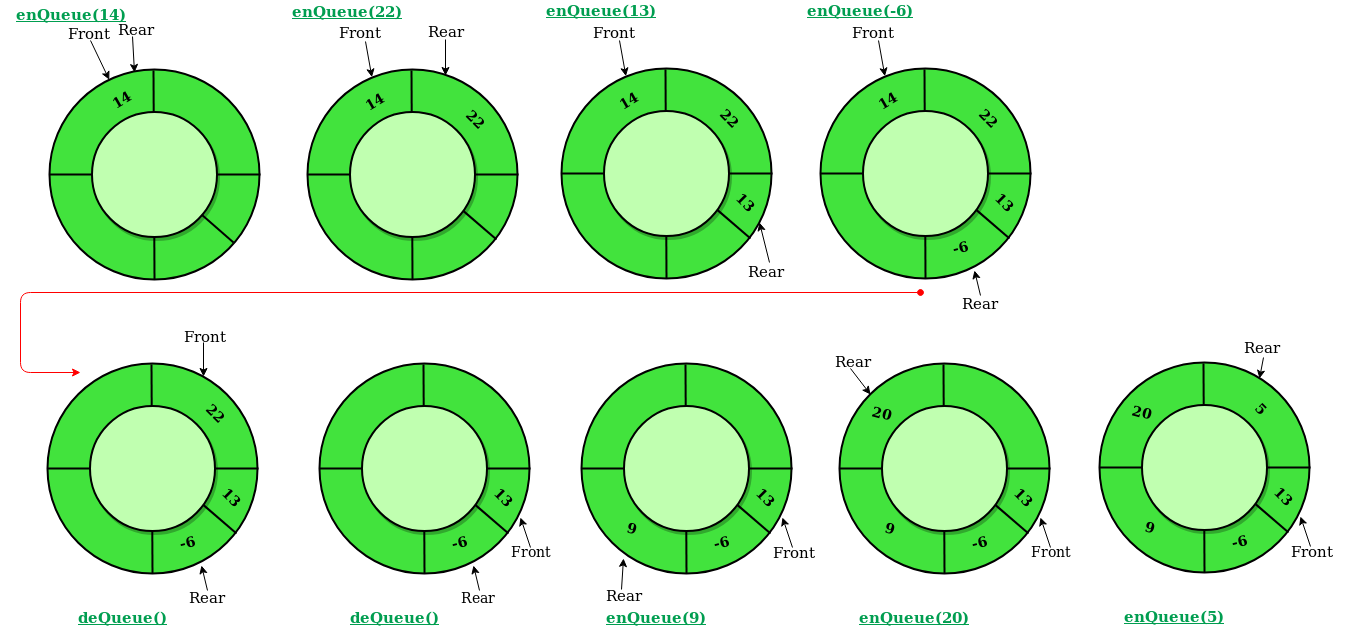
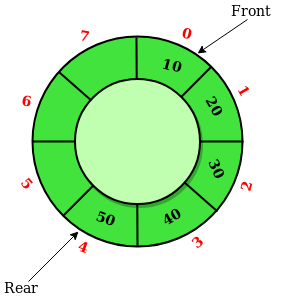
|  |
| --- |
| C code of creating Queue using Array  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100  int queue[MAX\_SIZE];  int front = -1, rear = -1;  bool isEmpty() {  return front == -1;  }  bool isFull() {  return rear == MAX\_SIZE - 1;  }  void enqueue(int data) {  if (isFull()) {  printf("Queue Overflow\n");  return;  }  if (isEmpty()) {  front = 0; // Handle first element case  }  queue[++rear] = data;  }  int dequeue() {  if (isEmpty()) {  printf("Queue Underflow\n");  return -1; // Or any default value  }  int data = queue[front];  if (front == rear) { // Handle single element case  front = rear = -1;  } else {  front++;  }  return data;  }  int peek() {  if (isEmpty()) {  printf("Queue is Empty\n");  return -1; // Or any default value  }  return queue[front];  }  int main() {  enqueue(10);  enqueue(20);  enqueue(30);  printf("%d dequeued\n", dequeue()); // Output: 10  printf("Front element: %d\n", peek()); // Output: 20  return 0;  } |
| Java code of creating Queue using Array  import java.io.\*;  /\*\*  \* This class defines and demonstrates the usage of a queue data structure in Java.  \* A queue follows the First-In, First-Out (FIFO) principle, meaning elements are  \* added at the back (rear) and removed from the front.  \*/  class Queue {  /\*\*  \* Represents the front index of the queue, pointing to the first element to remove.  \*/  int front;  /\*\*  \* Represents the rear index of the queue, pointing to the last added element (rear - 1).  \*/  int rear;  /\*\*  \* Represents the current size of the queue (number of elements).  \*/  int size;  /\*\*  \* Represents the maximum capacity of the queue, defining the maximum number of elements it can hold.  \*/  int cap;  /\*\*  \* The array that stores the queue elements.  \*/  int[] arr;  /\*\*  \* Constructor to create a queue of a given capacity.  \*  \* @param cap: The maximum capacity of the queue.  \*/  Queue(int cap) {  this.cap = cap;  this.front = this.size = 0;  this.rear = cap - 1; // Initialize rear to capacity - 1 for correct indexing  this.arr = new int[cap];  }  /\*\*  \* Checks if the queue is empty.  \*  \* @return True if the queue is empty, False otherwise.  \*/  boolean isEmpty() {  return size == 0;  }  /\*\*  \* Checks if the queue is full.  \*  \* @return True if the queue is full, False otherwise.  \*/  boolean isFull() {  return size == cap;  }  /\*\*  \* Adds an element to the rear of the queue.  \*  \* @param data: The element to be added.  \* @throws IllegalStateException if the queue is full.  \*/  void enqueue(int data) {  if (isFull()) {  throw new IllegalStateException("Queue overflow");  }  arr[rear++] = data;  size++;  rear %= cap; // Handle wrap-around for circular queue  }  /\*\*  \* Removes and returns the element from the front of the queue.  \*  \* @return The element removed from the front.  \* @throws IllegalStateException if the queue is empty.  \*/  int dequeue() {  if (isEmpty()) {  throw new IllegalStateException("Queue underflow");  }  int data = arr[front];  front = (front + 1) % cap;  size--;  return data;  }  /\*\*  \* Returns the element at the front of the queue without removing it.  \*  \* @return The element at the front of the queue.  \* @throws IllegalStateException if the queue is empty.  \*/  int peek() {  if (isEmpty()) {  throw new IllegalStateException("Queue is empty");  }  return arr[front];  }  /\*\*  \* Main function to demonstrate the usage of the Queue class.  \*  \* @param args: Command-line arguments (unused in this example).  \*/  public static void main(String[] args) {  Queue queue = new Queue(5); // Create a queue with capacity 5  queue.enqueue(10);  queue.enqueue(20);  queue.enqueue(30);  System.out.println(queue.dequeue() + " dequeued"); // Output: 10 dequeued  System.out.println("Front element: " + queue.peek()); // Output: Front element: 20  }  } |

* **Time Complexity:**
* **enqueue, dequeue, peek, isEmpty, isFull:** O(1) (constant time), efficient due to direct access using front and rear pointers.
* ***Types of Queue:***

There are different types of queues:

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

# **Circular Queue**



* ***What is a Circular Queue?***

A Circular Queue is an extended version of a normal queue where the last element of the queue is connected to the first element of the queue forming a circle.

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

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

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

Check whether the queue is full – [i.e., the rear end is in just before the front end in a circular manner].

If it is full then display Queue is full.

If the queue is not full then, insert an element at the end of the queue.

* **deQueue()** This function is used to delete an element from the circular queue. In a circular queue, the element is always deleted from the front position.

Check whether the queue is Empty.

If it is empty then display Queue is empty.

If the queue is not empty, then get the last element and remove it from the queue.

**Implement Circular Queue using Array:**

* + - 1. Initialize an array queue of size n, where n is the maximum number of elements that the queue can hold.
      2. Initialize two variables front and rear to -1.
      3. Enqueue: To enqueue an element x into the queue, do the following:
  + Increment rear by 1.
    - If rear is equal to n, set rear to 0.
  + If front is -1, set front to 0.
  + Set queue[rear] to x.
    - 1. Dequeue: To dequeue an element from the queue, do the following:
* Check if the queue is empty by checking if front is -1.
  + - If it is, return an error message indicating that the queue is empty.
* Set x to queue[front].
* If front is equal to rear, set front and rear to -1.
* Otherwise, increment front by 1 and if front is equal to n, set front to 0.
* Return x.
* **Code:**

|  |
| --- |
| * **Imp formula:** * front = new\_front ; * rear = (rear + 1) % MAX\_SIZE; |

|  |
| --- |
| C code  #include <stdio.h>  #include <stdlib.h>  #define MAX\_SIZE 100 // Maximum size of the queue  int queue[MAX\_SIZE];  int front = -1, rear = -1;  // Function prototypes  int isEmpty();  int isFull();  void enqueue(int data);  int dequeue();  int main() {  enqueue(10);  enqueue(20);  enqueue(30);  printf("Dequeued element: %d\n", dequeue());  printf("Dequeued element: %d\n", dequeue());  enqueue(40);  enqueue(50); // Queue becomes full  printf("Enqueueing when full at front: ");  if (enqueueAtFront(60)) { // Special function for front enqueueing  printf("Success\n");  } else {  printf("Failed\n");  }  printf("Dequeued element: %d\n", dequeue()); // Dequeue original elements  printf("Dequeued element: %d\n", dequeue());  printf("Dequeued element: %d\n", dequeue()); // Now dequeues 60  return 0;  }  // Check if the queue is empty  int isEmpty() {  return front == -1;  }  // Check if the queue is full  int isFull() {  return (rear + 1) % MAX\_SIZE == front;  }  // Enqueue an element (standard FIFO operation)  void enqueue(int data) {  if (isFull()) {  printf("Queue overflow\n");  return;  }  if (front == -1) { // Initialize the queue if empty  front = 0;  }  rear = (rear + 1) % MAX\_SIZE;  queue[rear] = data;  printf("Enqueued element: %d\n", data);  }  // Dequeue an element (standard FIFO operation)  int dequeue() {  if (isEmpty()) {  printf("Queue underflow\n");  return -1;  }  int data = queue[front];  if (front == rear) { // Reset pointers if only one element is left  front = rear = -1;  } else {  front = (front + 1) % MAX\_SIZE;  }  return data;  }  // Enqueue an element at the front of the queue (special case handling)  int enqueueAtFront(int data) {  if (isFull()) {  return 0; // Indicate failure if full and cannot enqueue at front  }  // Handle two cases:  if (front == -1 || front == 0) { // Empty queue or front at beginning  enqueue(data); // Use standard enqueue for efficiency  } else { // Special case: front in the middle, need to shift elements  int i, temp;  // Create space at the beginning by shifting elements one position to the right  for (i = rear; i >= front; i--) {  queue[(i + 1) % MAX\_SIZE] = queue[i];  }  // Insert the new element at the front  queue[front] = data;  front = (front - 1 + MAX\_SIZE) % MAX\_SIZE;  rear = (rear + 1) % MAX\_SIZE;  }  return 1; // Indicate success  } |
| Java Code  public class CircularQueue {  private static final int MAX\_SIZE = 100;  private int[] queue = new int[MAX\_SIZE];  private int front = -1, rear = -1;  // Check if the queue is empty  public boolean isEmpty() {  return front == -1;  }  // Check if the queue is full  public boolean isFull() {  return (rear + 1) % MAX\_SIZE == front;  }  // Enqueue an element (standard FIFO operation)  public void enqueue(int data) {  if (isFull()) {  System.out.println("Queue overflow");  return;  }  if (front == -1) { // Initialize the queue if empty  front = 0;  }  rear = (rear + 1) % MAX\_SIZE;  queue[rear] = data;  System.out.println("Enqueued element: " + data);  }  // Dequeue an element (standard FIFO operation)  public int dequeue() {  if (isEmpty()) {  System.out.println("Queue underflow");  return -1;  }  int data = queue[front];  if (front == rear) { // Reset pointers if only one element is left  front = rear = -1;  } else {  front = (front + 1) % MAX\_SIZE;  }  return data;  }  // Enqueue an element at the front of the queue (special case handling)  public boolean enqueueAtFront(int data) {  if (isFull()) {  return false; // Indicate failure if full and cannot enqueue at front  }  // Handle two cases:  if (front == -1 || front == 0) { // Empty queue or front at beginning  enqueue(data); // Use standard enqueue for efficiency  return true;  } else { // Special case: front in the middle, need to shift elements  // Calculate the new position for the front after shifting  int newFront = (front - 1 + MAX\_SIZE) % MAX\_SIZE;  // Check if there's enough space to shift without overwriting existing elements  if ((newFront + 1) % MAX\_SIZE == rear) {  System.out.println("Queue overflow - cannot enqueue at front");  return false;  }  // Create space at the beginning by shifting elements one position to the right  for (int i = rear; i >= front; i--) {  queue[(i + 1) % MAX\_SIZE] = queue[i];  }  // Insert the new element at the front  queue[newFront] = data;  front = newFront;  rear = (rear + 1) % MAX\_SIZE;  }  return true; // Indicate success  }  } |

# **Linked- List**

Linked Lists: A Flexible Alternative to Arrays

**Linked Lists vs. Arrays: Why Use Both?**

While arrays are a fundamental data structure, they have limitations. Let's explore these limitations and how linked lists offer an alternative solution:

* **Array Limitations:**
* Fixed Size: Arrays require specifying the size upfront, which can be problematic if the data size is unknown or needs to grow dynamically.
* Inefficient Insertion/Deletion: Inserting or deleting elements in the middle of an array is expensive, often requiring data shifting and potentially reallocating the entire array.
* Memory Waste: If the array is not fully utilized, wasted memory cannot be reclaimed until the entire array is reallocated.
* Linked List Advantages:
* Dynamic Size: Linked lists don't require a predefined size, allowing them to grow or shrink as needed at runtime.
* Efficient Insertion/Deletion: Inserting or deleting elements in the middle of a linked list is efficient, as only the pointers need to be adjusted, not the data itself.
* No Memory Waste: Linked lists only occupy memory for the actual data, eliminating wasted space.

**Therefore, while arrays excel in random access and space efficiency for fixed-size data, linked lists become indispensable when:**

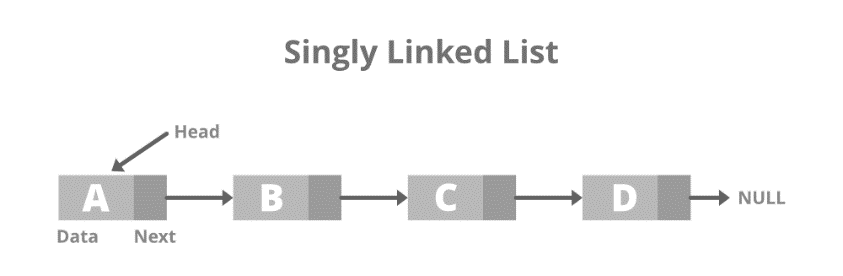
* Data size is unknown or needs to grow/shrink dynamically.
* Frequent insertions/deletions are anticipated at the middle of the data.
* Memory usage optimization is crucial.

**Singly Linked List:**

A singly linked list is a linear data structure consisting of elements called nodes. Each node stores two pieces of information:

Data: The actual value or object being held.

Next Pointer: A reference to the next node in the sequence.



* Imagine a train: each train car is a node, carrying passengers (data) and connected to the next car (next pointer) via a coupling. The first car has no "previous", and the last car has no "next", marking the list's beginning and end.

Singly linked lists offer efficient insertion and deletion operations, making them well-suited for scenarios where modifying the data order is frequent. However, accessing a specific element by its index is slower than with arrays, as it involves traversing the list from the beginning until the desired element is found.

* Creation of Node In Link-List

Code Section

|  |
| --- |
| C Section Of Node  typedef struct Node{  int data ; // Data value stored in the node  struct Node\* next ; // Pointer to the next node in the list  } |
| Java Section Of Node  public class Node {  int data; // Data value stored in the node  Node next; // Reference to the next node in the list  } |

Explanation:

* Each node in the linked list is a distinct structure containing two fields:
  + **data:** Stores the actual data value.
  + **next:** A pointer (C) or reference (Java) to the next node in the sequence, forming the chain-like structure.
* The head pointer (C) or reference (Java) points to the first node in the list. A NULL value in head indicates an empty list.

## **Insertion**

Insertion :-

* at beginning,
* at end,
* at position n

Algorithms:

|  |
| --- |
| * **Adding data (Insertion):**   ***Time Complexity:***  Best case: O(1) (if adding at the beginning)  Average case: O(n) (requires traversal to find the insertion point)  Worst case: O(n) (if adding at the end) |

|  |
| --- |
| C code  // Structure for a node  struct Node {  int data;  struct Node\* next;  };  // Function prototypes (assuming Node is declared globally)  void insertAtBeginning(struct Node\*\* head\_ref, int new\_data);  void insertAtEnd(struct Node\*\* head\_ref, int new\_data);  void insertAfter(struct Node\* prev\_node, int new\_data);  // Insertion at beginning  void insertAtBeginning(struct Node\*\* head\_ref, int new\_data) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node)); // Allocate memory for new node  new\_node->data = new\_data;  new\_node->next = \*head\_ref; // Point the new node's next to the current head  \*head\_ref = new\_node; // Update the head pointer to point to the new node  }  // Insertion at end  void insertAtEnd(struct Node\*\* head\_ref, int new\_data) {  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node)); // Allocate memory for new node  struct Node\* last = \*head\_ref;  new\_node->data = new\_data;  new\_node->next = NULL; // Mark the last node  if (\*head\_ref == NULL) { // Handle empty list case  \*head\_ref = new\_node;  return;  }  while (last->next != NULL) { // Traverse to the last node  last = last->next;  }  last->next = new\_node; // Link the last node to the new node  }  // Insertion after a specific node  void insertAfter(struct Node\* prev\_node, int new\_data) {  if (prev\_node == NULL) { // Handle invalid previous node  printf("The given previous node cannot be NULL\n");  return;  }  struct Node\* new\_node = (struct Node\*)malloc(sizeof(struct Node)); // Allocate memory for new node  new\_node->data = new\_data;  new\_node->next = prev\_node->next; // Link the new node's next to the node after prev\_node  prev\_node->next = new\_node; // Link prev\_node to the new node  } |
| Java Code  // Node class  public class Node {  int data;  Node next;  // Constructor  public Node(int data) {  this.data = data;  }  }  // Function prototypes (assuming Node class is defined)  void insertAtBeginning(Node head, int newData);  void insertAtEnd(Node head, int newData);  void insertAfter(Node prevNode, int newData);  // Insertion at beginning  public void insertAtBeginning(Node head, int newData) {  Node newNode = new Node(newData); // Create a new node  newNode.next = head; // Point the new node's next to the current head  head = newNode; // Update the head reference to point to the new node  }  // Insertion at end  public void insertAtEnd(Node head, int newData) {  Node newNode = new Node(newData); // Create a new node  Node last = head;  if (head == null) { // Handle empty list case  head = newNode;  return;  }  while (last.next != null) { // Traverse to the last node  last = last.next;  }  last.next = newNode; // Link the last node to the new node  }  // Insertion after a specific node  public void insertAfter(Node prevNode, int newData) {  if (prevNode == null) { // Handle invalid previous node  System.out.println("The given previous node cannot be NULL");  return;  }  Node newNode = new Node(newData); // Create a new node  newNode.next = prevNode.next; // Link the new node's next to the node after prevNode  prevNode.next = newNode; // Link prevNode to the new node  } |

## **Deletion**

Deletion:-

* at beginning,
* at end,
* at position n

|  |
| --- |
| C code  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  struct Node\* head = NULL; // Assuming a global head pointer  int deleteFirst() {  if (head == NULL) {  printf("List is empty\n");  return -1; // Indicate error  }  struct Node\* temp = head;  int data = temp->data;  head = head->next; // Update head to point to the next node  free(temp); // Free memory of the deleted node  return data;  }  int deleteLast() {  if (head == NULL) {  printf("List is empty\n");  return -1;  }  if (head->next == NULL) { // If only one node exists  int data = head->data;  free(head);  head = NULL; // Mark the list as empty  return data;  }  struct Node\* temp = head;  while (temp->next->next != NULL) { // Traverse to the second last node  temp = temp->next;  }  int data = temp->next->data;  free(temp->next); // Free memory of the last node  temp->next = NULL; // Mark the end of the list  return data;  }  int deleteAtPosition(int n) {  if (head == NULL) {  printf("List is empty\n");  return -1;  }  if (n == 1) { // Deleting the first node  return deleteFirst();  }  struct Node\* temp = head;  for (int i = 1; i < n - 1 && temp != NULL; i++) { // Traverse to the node before the one to be deleted  temp = temp->next;  }  if (temp == NULL || temp->next == NULL) { // If n is invalid or beyond the last node  printf("Invalid position\n");  return -1;  }  struct Node\* nodeToDelete = temp->next;  int data = nodeToDelete->data;  temp->next = nodeToDelete->next; // Skip over the node to be deleted  free(nodeToDelete); // Free memory  return data;  } |
| Java Code  class Node {  int data;  Node next;  Node(int data) {  this.data = data;  }  }  Node head = null; // Assuming a global head reference  int deleteFirst() {  if (head == null) {  System.out.println("List is empty");  return -1; // Indicate error  }  int data = head.data;  head = head.next; // Update head to point to the next node  return data;  }  int deleteLast() {  if (head == null) {  System.out.println("List is empty");  return -1;  }  if (head.next == null) { // If only one node exists  int data = head.data;  head = null; // Mark the list as empty  return data;  }  Node temp = head;  while (temp.next.next != null) { // Traverse to the second last node  temp = temp.next;  }  int data = temp.next.data;  temp.next = null; // Mark the end of the list  return data;  }  int deleteAtPosition(int n) {  if (head == null) {  System.out.println("List is empty");  return -1;  }  if (n == 1) { // Deleting the first node  return deleteFirst();  }  Node temp = head;  for (int i = 1; i < n - 1 && temp != null; i++) { // Traverse to the node before the one to be deleted  temp = temp.next;  }  if (temp == null || temp.next == null) { // If n is invalid or beyond the last node  System.out.println("Invalid position");  return -1;  }  Node nodeToDelete = temp.next;  int data = nodeToDelete.data;  temp.next = nodeToDelete.next; // Skip over the node to be deleted  return data;  } |

Explanation (C and Java):

1. Deletion at the Beginning (deleteFirst)

* Logic: Update the head pointer to point to the second node in the linked list, effectively removing the first node.
* Time Complexity: O(1) (constant time)

2. Deletion at the End (deleteLast)

* Logic: Traverse the list to find the second-to-last node. Update its next pointer to NULL to mark the end of the list.
* Time Complexity: O(n) (requires traversal to reach the end)

3. Deletion at a Given Position (deleteAtPosition)

* Logic:
  + Handle empty list and special case of deleting the first node.
  + Traverse the list to find the node just before the one to be deleted (n-1 position).
  + Update the next pointer of the previous node to skip over the node to be deleted.
* Time Complexity: O(n) (traverses the list until the desired position)

## **Searching:-**

* By value
* at position

code

|  |
| --- |
| C Code  #include <stdio.h>  #include <stdlib.h>  struct Node {  int data;  struct Node\* next;  };  // ... (Assuming head pointer and functions to create linked list from array)  int searchByValue(int value) {  int position = 1;  struct Node\* current = head;  while (current != NULL) {  if (current->data == value) {  return position; // Value found  }  current = current->next;  position++;  }  return -1; // Value not found  }  int searchByPosition(int n) {  if (n < 1) {  printf("Invalid position\n");  return -1;  }  int position = 1;  struct Node\* current = head;  while (current != NULL && position < n) {  current = current->next;  position++;  }  if (current == NULL) {  printf("Position out of bounds\n");  return -1;  }  return current->data; // Data at the specified position  } |
| Java Code  class Node {  int data;  Node next;  // ... (constructor)  }  // ... (Assuming head reference and functions to create linked list from array)  int searchByValue(int value) {  int position = 1;  Node current = head;  while (current != null) {  if (current.data == value) {  return position; // Value found  }  current = current.next;  position++;  }  return -1; // Value not found  }  int searchByPosition(int n) {  if (n < 1) {  System.out.println("Invalid position");  return -1;  }  int position = 1;  Node current = head;  while (current != null && position < n) {  current = current.next;  position++;  }  if (current == null) {  System.out.println("Position out of bounds");  return -1;  }  return current.data; // Data at the specified position  } |

# Menu Driven Code For Creation a link-list and add, delete, search and delete

|  |
| --- |
| C Code  #include <stdio.h>  #include <stdlib.h>  // Node structure  struct Node {  int data;  struct Node\* next;  };  // Global variable (head pointer)  struct Node\* head = NULL;  // Function prototypes (already defined)  // Function to search by value  int searchByValue(int value) {  if (head == NULL) {  printf("List is empty\n");  return -1;  }  int position = 1;  struct Node\* current = head;  while (current != NULL) {  if (current->data == value) {  return position;  }  current = current->next;  position++;  }  return -1; // Value not found  }  // Function to search by position  int searchByPosition(int position) {  if (head == NULL) {  printf("List is empty\n");  return -1;  }  if (position < 1) {  printf("Invalid position\n");  return -1;  }  int current\_position = 1;  struct Node\* current = head;  while (current != NULL && current\_position < position) {  current = current->next;  current\_position++;  }  if (current == NULL) {  printf("Position out of bounds\n");  return -1;  }  return current->data;  }  // Function to print the list  void printList() {  if (head == NULL) {  printf("List is empty\n");  return;  }  struct Node\* temp = head;  printf("List: ");  while (temp != NULL) {  printf("%d ", temp->data);  temp = temp->next;  }  printf("\n");  }  // Function to delete the entire list  void deleteList() {  while (head != NULL) {  deleteFirst();  }  printf("List deleted\n");  }  int main() {  int choice, data, position;  while (1) {  printf("\nMenu:\n");  printf("1. Add\n");  printf("2. Print\n");  printf("3. Delete\n");  printf("4. Search\n");  printf("5. End Program\n");  printf("Enter your choice: ");  scanf("%d", &choice);  switch (choice) {  case 1: // Add  // ... (existing code for adding a node)  break;  case 2: // Print  printList();  break;  case 3: // Delete  // ... (existing code for deleting a node)  break;  case 4: // Search  printf("Search by value (1) or position (2): ");  scanf("%d", &position);  if (position == 1) {  // ... (existing code for searching by value)  } else if (position == 2) {  printf("Enter position to search: ");  scanf("%d", &data);  int search\_result = searchByPosition(data);  if (search\_result == -1) {  printf("Invalid position or position out of bounds\n");  } else {  printf("Value at position %d: %d\n", data, search\_result);  }  } else {  printf("Invalid choice\n");  }  break;  case 5: // End Program  deleteList(); // Delete the entire list before ending  printf("Linked list deleted\n");  exit(0);  default:  printf("Invalid choice\n");  }  }  return 0;  } |
| Java Code  public class SLL {  Node head;  class Node{  String data;  Node next; // next ek agle node ko point krrha hai to uska type Node hoga    Node(String d){  data = d;  next = null; //shuruwat me single node create hoga koi list nhi create nhi hogi. isiliye shuruwat me node ke next ko null krdenge  }  }  // Adding - First  public void addFirst(String data) {  Node newNode = new Node(data);  if(head == null){  head = newNode;  return;  }  newNode.next = head;  head = newNode;  }  // Adding - Last  public void addLast(String data){  Node newNode = new Node(data);    if(head == null){  head = newNode;  return;  }  // traverse  Node currNode = head;  while(currNode.next != null){  currNode = currNode.next;  }  currNode.next = newNode;  }  // Adding - Middle  public void addInMiddle(int pos, String data){  if(pos > getSize() || pos < 0){  System.out.println("Invalid Position Value");  return;  }  Node newNode = new Node(data);    if(head == null || pos == 0) {  newNode.next = head;  head = newNode;  return;  }  Node currNode = head;  int cnt = 1;  while(cnt < pos){  currNode = currNode.next;  cnt++;  }  newNode.next = currNode.next;  currNode.next = newNode;  }  // Print  public void printList(){  if(head == null){  System.out.println("List is Empty");  return;  }    Node currNode = head;  while(currNode != null){  System.out.print(currNode.data + " -> ");  currNode = currNode.next;  }  System.out.println("NULL");  }    // Delete - First  public void deleteFirst(){  if(head == null){  System.out.println("List is Empty");  return;  }  head = head.next;  }  // Delete - Last  public void deleteLast(){  if(head == null){  System.out.println("List is Empty");  return;  }  if(head.next == null){  head = null;  return;  }  Node secondLast = head;  Node lastNode = head.next; // head.next = null -> lastNode = null  while(lastNode.next != null){ // null.next  lastNode = lastNode.next;  secondLast = secondLast.next;  }  secondLast.next = null;  }  // Delete - Middle  public void deleteInMiddle(int pos){  Node currNode = head;  Node prevNode = null;  int cnt = 1;  while(cnt <= pos){  prevNode = currNode;  currNode = currNode.next;  cnt++;  }  prevNode.next = currNode.next;  currNode.next = null;  }  public int getSize() {    Node temp = head;  int count = 0;  while(temp != null){  count++;  temp = temp.next;  }  return count;  }    public static void main(String[] args){  SLL list = new SLL();  list.addInMiddle(0, "This");  list.addInMiddle(1, "is");  list.addInMiddle(2, "a");  list.addInMiddle(3, "Linked");  list.addInMiddle(4, "List");  list.addInMiddle(2, "definitely");  list.printList();  list.deleteInMiddle(2);  list.printList();  list.deleteFirst();  list.printList();  list.deleteLast();  list.printList();  list.deleteFirst();  list.printList();  System.out.println(list.getSize());    }  } |

Double