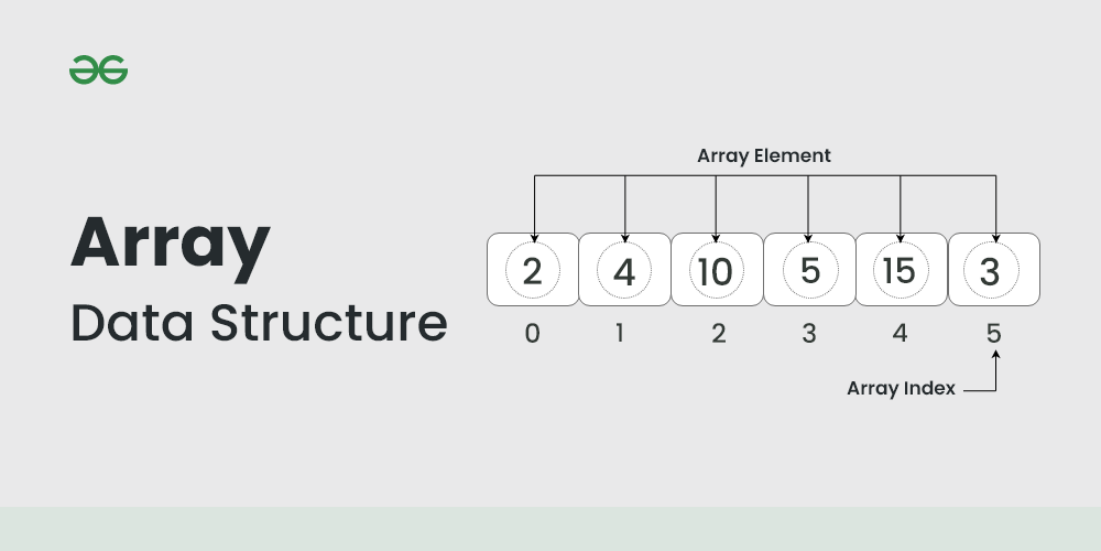
Array is a linear data structure that is a collection of data elements of same types. Arrays are stored in contiguous memory locations. It is a static data structure with a fixed size.



Applications of Array Data Structure:

Arrays mainly have advantages like random access and cache friendliness over other data structures that make them useful.
Below are some applications of arrays.

- **Storing and accessing data**: Arrays store elements in a specific order and allow constant-time O(1) access to any element.
- **Searching**: If data in array is sorted, we can search an item in O(log n) time. We can also find floor(), ceiling(), kth smallest, kth largest, etc efficiently.
- **Matrices**: Two-dimensional arrays are used for matrices in computations like graph algorithms and image processing.
- **Implementing other data structures:** Arrays are used as the underlying data structure for implementing stacks and queues.
- **Dynamic programming**: Dynamic programming algorithms often use arrays to store intermediate results of subproblems in order to solve a larger problem.
- **Data Buffers:** Arrays serve as data buffers and queues, temporarily storing incoming data like network packets, file streams, and database results before processing.

Advantages of Array Data Structure:

- **Efficient and Fast Access:** Arrays allow direct and efficient access to any element in the collection with constant access time, as the data is stored in contiguous memory locations.
- **Memory Efficiency:** Arrays store elements in contiguous memory, allowing efficient allocation in a single block and reducing memory fragmentation.

- **Versatility:** Arrays can be used to store a wide range of data types, including integers, floating-point numbers, characters, and even complex data structures such as objects and pointers.
- **Compatibility with hardware:** The array data structure is compatible with most hardware architectures, making it a versatile tool for programming in a wide range of environments.

Disadvantages of Array Data Structure:

- **Fixed Size:** Arrays have a fixed size set at creation. Expanding an array requires creating a new one and copying elements, which is time-consuming and memory-intensive.
- **Memory Allocation Issues:** Allocating large arrays can cause memory exhaustion, leading to crashes, especially on systems with limited resources.
- **Insertion and Deletion Challenges:** Adding or removing elements requires shifting subsequent elements, making these operations inefficient.
- **Limited Data Type Support:** Arrays support only elements of the same type, limiting their use with complex data types.
- **Lack of Flexibility:** Fixed size and limited type support make arrays less adaptable than structures like linked lists or trees.

Calculating the address of any element In the 1-D array:

A 1-dimensional array (or single-dimension array) is a type of linear array. Accessing its elements involves a single subscript that can either represent a row or column index.

Example:

1-D array

To find the address of an element in an array the followingformula is used-

Address of A[Index] = B + W * (Index – LB)

Where:

- ***Index** = The index of the element whose address is to be found (not the value of the element).*
- ***B** = Base address of the array.*
- ***W** = Storage size of one element in bytes.*
- ***LB** = Lower bound of the index (if not specified, assume zero).*

Example: Given the base address of an array **A[1300 ……. 1900]** as **1020** and the size of each element is 2 bytes in the memory, find the address of **A[1700].**

Solution:

Given:

- *Base address (B) = 1020*
- *Lower bound (LB) = 1300*
- *Size of each element (W) = 2 bytes*
- *Index of element (not value) = 1700*

Formula used:

*Address of A[Index] = B + W * (Index – LB)*

*Address of A[1700] = 1020 + 2 * (1700 – 1300)*

*= 1020 + 2 * (400)*

= 1020 + 800
Address of A[1700] = 1820

Calculate the address of any element in the 2-D array:

The 2-dimensional array can be defined as an array of arrays. The 2-Dimensional arrays are organized as matrices which can be represented as the collection of rows and columns as array[M][N] where M is the number of rows and N is the number of columns.

Example:

Columns

2D Array	0	1	2
0	a[0][0]	a[0][1]	a[0][2]
1	a[1][0]	a[1][1]	a[1][2]
2	a[2][0]	a[2][1]	a[2][2]

Rows

2-D array

To find the address of any element in a **2-Dimensional** array there are the following two ways-

1. Row Major Order
2. Column Major Order

1. Row Major Order:

Row major ordering assigns successive elements, moving across the rows and then down the next row, to successive memory locations. In simple language, the elements of an array are stored in a Row-Wise fashion.

To find the address of the element using row-major order uses the following formula:

Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))

I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in an array(in byte),
LR = Lower Limit of row/start row index of the matrix(If not given assume it as zero),

LC = Lower Limit of column/start column index of the matrix(If not given assume it as zero),
N = Number of column given in the matrix.

Example: Given an array, **arr[1……10][1……15]** with base value **100** and the size of each element is **1 Byte** in memory. Find the address of **arr[8][6]** with the help of row-major order.
Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of column given in the matrix N = Upper Bound – Lower Bound + 1
= 15 – 1 + 1
= 15
Formula:
*Address of A[I][J] = B + W * ((I – LR) * N + (J – LC))*
Solution:
*Address of A[8][6] = 100 + 1 * ((8 – 1) * 15 + (6 – 1))*
*= 100 + 1 * ((7) * 15 + (5))*
*= 100 + 1 * (110)*
Address of A[I][J] = 210

2. Column Major Order:

If elements of an array are stored in a column-major fashion means moving across the column and then to the next column then it's in column-major order. To find the address of the element using column-major order use the following formula:
Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))
I = Row Subset of an element whose address to be found,
J = Column Subset of an element whose address to be found,
B = Base address,
W = Storage size of one element store in any array(in byte),
LR = Lower Limit of row/start row index of matrix(If not given assume it as zero),
LC = Lower Limit of column/start column index of matrix(If not given assume it as zero),
M = Number of rows given in the matrix.

Example: Given an array **arr[1……10][1……15]** with a base value of **100** and the size of each element is **1 Byte** in memory find the address of arr[8][6] with the help of column-major order.

Solution:
Given:
Base address B = 100
Storage size of one element store in any array W = 1 Bytes
Row Subset of an element whose address to be found I = 8
Column Subset of an element whose address to be found J = 6
Lower Limit of row/start row index of matrix LR = 1
Lower Limit of column/start column index of matrix = 1
Number of Rows given in the matrix M = Upper Bound – Lower Bound + 1
= 10 – 1 + 1
= 10
Formula: used
*Address of A[I][J] = B + W * ((J – LC) * M + (I – LR))*
*Address of A[8][6] = 100 + 1 * ((6 – 1) * 10 + (8 – 1))*
*= 100 + 1 * ((5) * 10 + (7))*
*= 100 + 1 * (57)*
Address of A[I][J] = 157

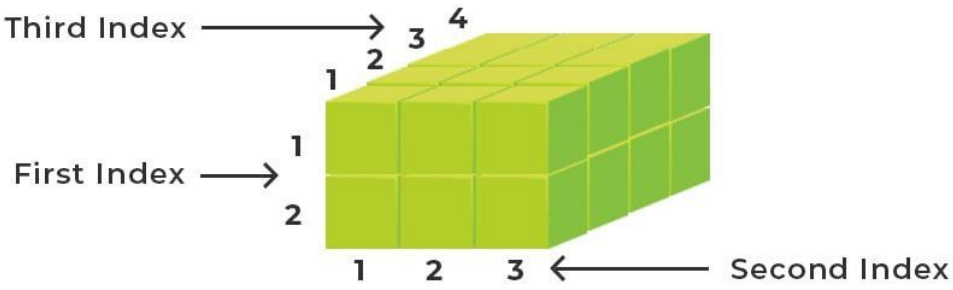
From the above examples, it can be observed that for the same position two different address locations are obtained that's because in row-major order movement is done across the rows and then down to the next row, and in column-major order, first move down to the first column and then next column. So both the answers are right.
So it's all based on the position of the element whose address is to be found for some cases the same answers is also obtained with row-major order and column-major order and for some cases, different answers are obtained.

Calculate the address of any element in the 3-D Array:

A **3-Dimensional** array is a collection of 2-Dimensional arrays. It is specified by using three subscripts:

1. Block size
2. Row size
3. Column size

More dimensions in an array mean more data can be stored in that array.
Example:



Three-Dimensional Array
with 24 Elements

3-D array

To find the address of any element in 3-Dimensional arrays there are the following two ways-

- Row Major Order
- Column Major Order

1. Row Major Order:

To find the address of the element using row-major order, use the following formula:

Address of A[i][j][k] = B + W *(P* N * (i-x) + P*(j-y) + (k-z))

Here:
B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of Row
y = Lower Bound of Column
z = Lower Bound of Width

Example: Given an array, **arr[1:9, -4:1, 5:10]** with a base value of **400** and the size of each element is **2 Bytes** in memory find the address of element **arr[5][-1][8]** with the help of row-major order?

Solution:

Given:

Block Subset of an element whose address to be found I = 5
Row Subset of an element whose address to be found J = -1
Column Subset of an element whose address to be found K = 8
Base address B = 400
Storage size of one element store in any array(in Byte) W = 2
Lower Limit of blocks in matrix x = 1

Lower Limit of row/start row index of matrix y = -4
Lower Limit of column/start column index of matrix z = 5
M(row) = Upper Bound – Lower Bound + 1 = 1 – (-4) + 1 = 6
N(Column)= Upper Bound – Lower Bound + 1 = 10 – 5 + 1 = 6

Formula used:
*Address of[I][J][K] =B + W (M * N(i-x) + N *(j-y) + (k-z))*
Solution:
*Address of arr[5][-1][8] = 400 + 2 * {[6 * 6 * (5 – 1)] + 6 * [(-1 + 4)]} + [8 – 5]*
*= 400 + 2 * (6*6*4)+(6*3)+3*
*= 400 + 2 * (165)*
= 730

2. Column Major Order:

To find the address of the element using column-major order, use the following formula:1
Address of A[i][j][k]=B+W×(M×P×(k−z)+M×(j−y)+(i−x))
Here:
B = Base Address (start address)
W = Weight (storage size of one element stored in the array)
M = Row (total number of rows)
N = Column (total number of columns)
P = Width (total number of cells depth-wise)
x = Lower Bound of block (first subscipt)
y = Lower Bound of Row
z = Lower Bound of Column

Example: Given an array **arr[1:8, -5:5, -10:5]** with a base value of **400** and the size of each element is **4 Bytes** in memory find the address of element **arr[3][3][3]** with the help of column-major order?
Solution:
Given:
Row Subset of an element whose address to be found I = 3
Column Subset of an element whose address to be found J = 3
Block Subset of an element whose address to be found K = 3
Base address B = 400
Storage size of one element store in any array(in Byte) W = 4
Lower Limit of blocks in matrix x = 1
Lower Limit of row/start row index of matrix y = -5
Lower Limit of column/start column index of matrix z = -10
M (row)= Upper Bound – Lower Bound + 1 = 8 – 1 + 1 = 8
N (column)= Upper Bound – Lower Bound + 1 = 5 – (-5) + 1 = 11
Formula used:
Address of A[i][j][k]=B+W×(M×P×(k−z)+M×(j−y)+(i−x))

Solution:

*Address of arr[3][3][3] = 400 + 4 * ((11*8*(3-(-10)+ 8*(3-(-5)+ (3-1))*

*= 400 + 4 * ((88 * 13+ 8 * 8 + 2)*

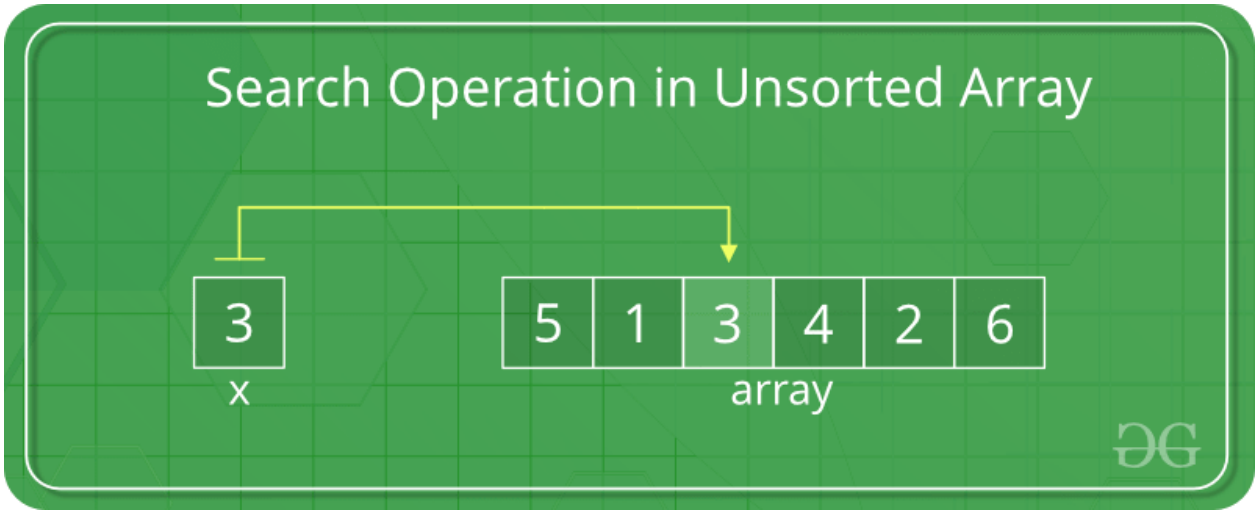
*= 400 + 4 * (1210)*

= 400 + 4840

= 5240

Search Operation:

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



Coding implementation of the search operation:

Try it on GfG Practice

C++CJavaPythonC#JavaScriptPHP

```
// C++ program to implement linear
```

```cpp
// search in unsorted array
#include <bits/stdc++.h>
using namespace std;

// Function to implement search operation
int findElement(int arr[], int n, int key)
{
    int i;
    for (i = 0; i < n; i++)
        if (arr[i] == key)
            return i;

      // If the key is not found
    return -1;
}

// Driver's Code
int main()
{
    int arr[] = { 12, 34, 10, 6, 40 };
    int n = sizeof(arr) / sizeof(arr[0]);

    // Using a last element as search element
    int key = 40;

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

    if (position == -1)
        cout << "Element not found";
    else
        cout << "Element Found at Position: "
             << position + 1;

    return 0;
}

// This code is contributed
// by Akanksha Rai
```

Output

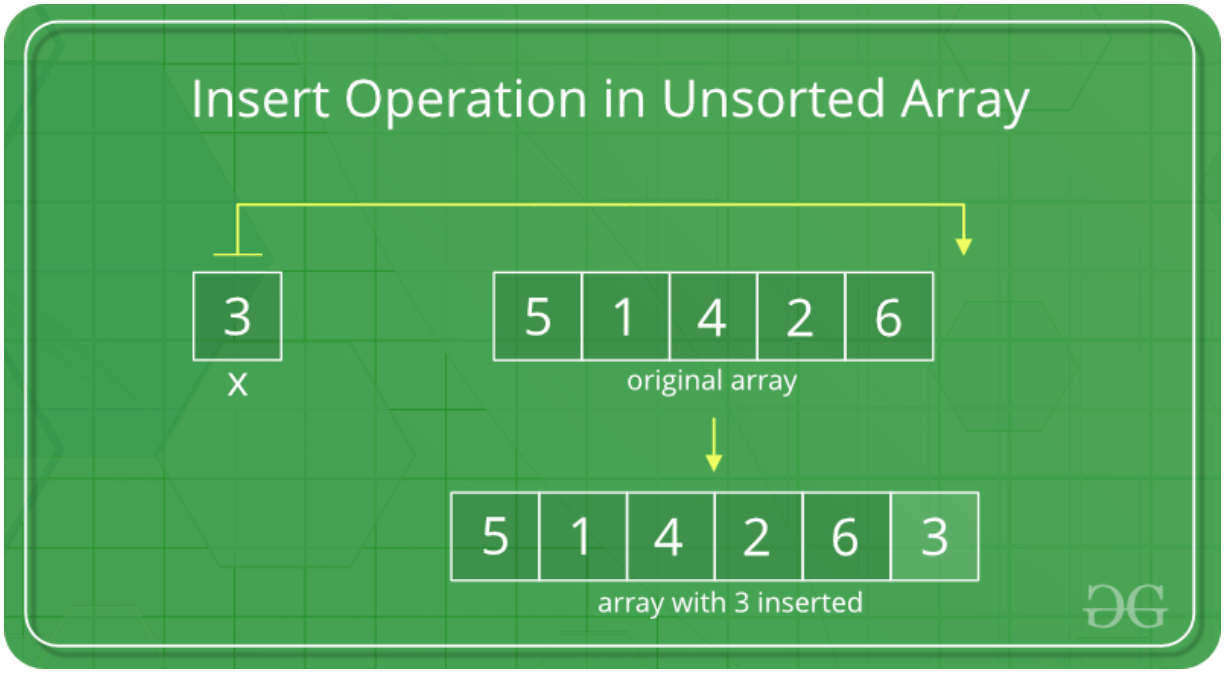
```
Element Found at Position: 5
```

Time Complexity: O(N)
Auxiliary Space: O(1)

Insert Operation:

1. Insert at the end:

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



Coding implementation of inserting an element at the end:

C++CJavaPythonC#JavaScriptPHP

```cpp
#include <iostream>
using namespace std;

// Inserts a key in arr[] of the given capacity.
// n is the current size of arr[]. This
// function returns n + 1 if insertion
// is successful, else n.
int insertEnd(int arr[], int n, int key, int capacity)
{
    // Cannot insert more elements if n is
    // already more than or equal to capacity
    if (n >= capacity)
        return n;

    arr[n] = key;
    return (n + 1);
}

int main()
{
    int arr[20] = { 12, 16, 20, 40, 50, 70 };
    int capacity = sizeof(arr) / sizeof(arr[0]);
    int n = 6;
    int i, key = 26;

    cout << "Before Insertion: ";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";

    // Inserting key
```

```
    n = insertEnd(arr, n, key, capacity);

    cout << "\nAfter Insertion: ";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}
```

Output

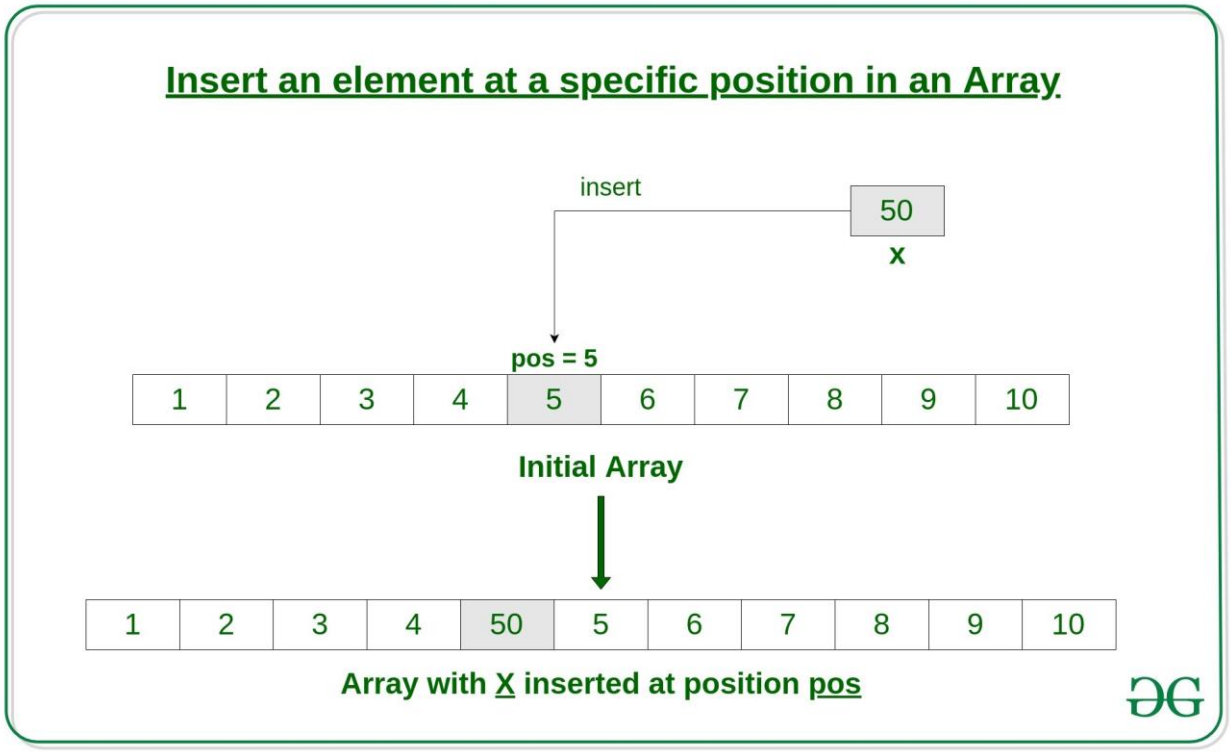
```
 Before Insertion: 12 16 20 40 50 70

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

Time Complexity: O(1)
Auxiliary Space: O(1)

2. Insert at any position

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



Coding implementation of inserting an element at any position:

C++CJavaPythonC#JavaScriptPHP

```
// C++ Program to Insert an element
// at a specific position in an Array

#include <bits/stdc++.h>
using namespace std;

// Function to insert element
// at a specific position
```

```
void insertElement(int arr[], int n, int x, int pos)
{
    // shift elements to the right
    // which are on the right side of pos
    for (int i = n - 1; i >= pos; i--)
        arr[i + 1] = arr[i];

    arr[pos] = x;
}

// Driver's code
int main()
{
    int arr[15] = { 2, 4, 1, 8, 5 };
    int n = 5;

    cout<<"Before insertion : ";
    for (int i = 0; i < n; i++)
        cout<<arr[i]<<" ";

    cout<<endl;

    int x = 10, pos = 2;

      // Function call
    insertElement(arr, n, x, pos);
    n++;

    cout<<"After insertion : ";
    for (int i = 0; i < n; i++)
        cout<<arr[i]<<" ";

    return 0;
}
```

Output

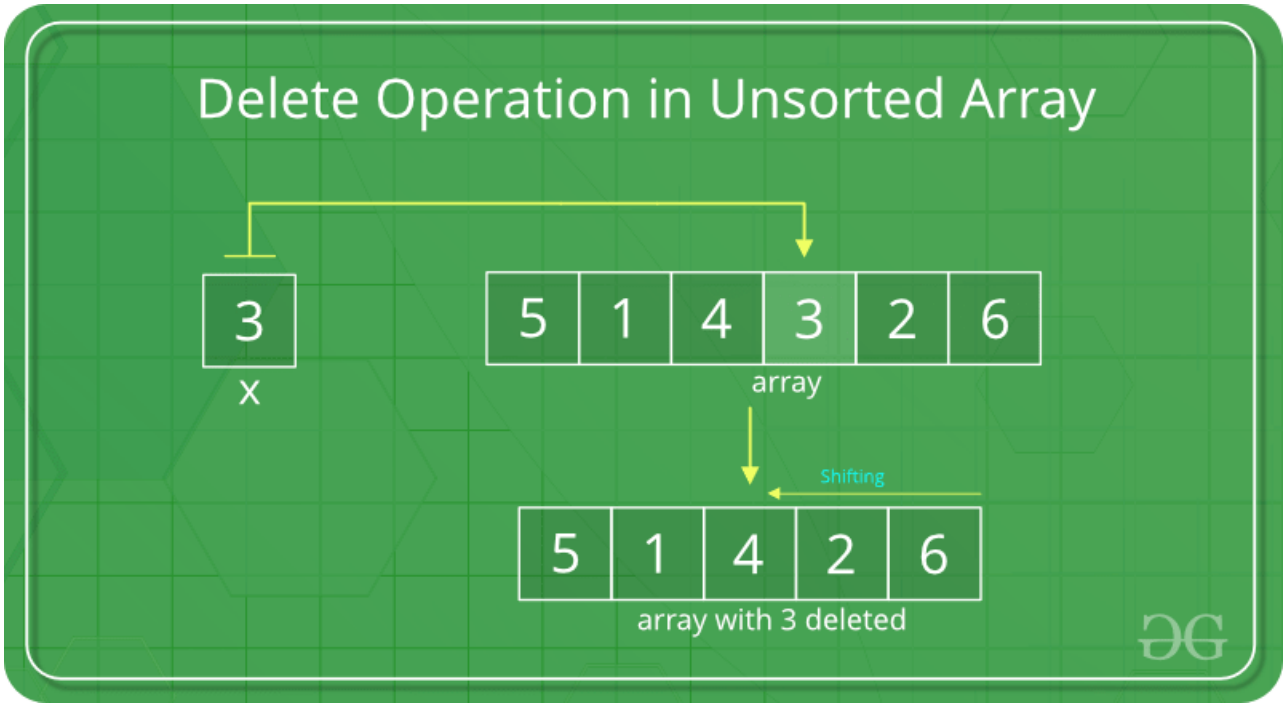
```
Before insertion : 2 4 1 8 5

After insertion : 2 4 10 1 8 5
```

Time complexity: O(N)
Auxiliary Space: O(1)

Delete Operation:

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



C++CJavaPythonC#JavaScriptPHP

```cpp
// C++ program to implement delete operation in a
// unsorted array
#include <iostream>
using namespace std;

// To search a key to be deleted
int findElement(int arr[], int n, int key);

// Function to delete an element
int deleteElement(int arr[], int n, int key)
{
    // Find position of element to be deleted
    int pos = findElement(arr, n, key);

    if (pos == -1) {
        cout << "Element not found";
        return n;
    }

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

    return n - 1;
}

// Function to implement search operation
int findElement(int arr[], int n, int key)
{
    int i;
```

```
    for (i = 0; i < n; i++)
        if (arr[i] == key)
            return i;

    return -1;
}

// Driver's code
int main()
{
    int i;
    int arr[] = { 10, 50, 30, 40, 20 };

    int n = sizeof(arr) / sizeof(arr[0]);
    int key = 30;

    cout << "Array before deletion\n";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";


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

    cout << "\n\nArray after deletion\n";
    for (i = 0; i < n; i++)
        cout << arr[i] << " ";

    return 0;
}

// This code is contributed by shubhamsingh10
```

Output

```
Array before deletion

10 50 30 40 20


Array after deletion

10 50 40 20
```

Time Complexity: O(N)
Auxiliary Space: O(1)