Table of Contents

[Introduction to Arrays 1](#_Toc58445872)

[Accessing array elements 2](#_Toc58445873)

[Searching in an Array 3](#_Toc58445874)

[Insertion and Deletion of Arrays 4](#_Toc58445875)

[Insertion in Arrays 4](#_Toc58445876)

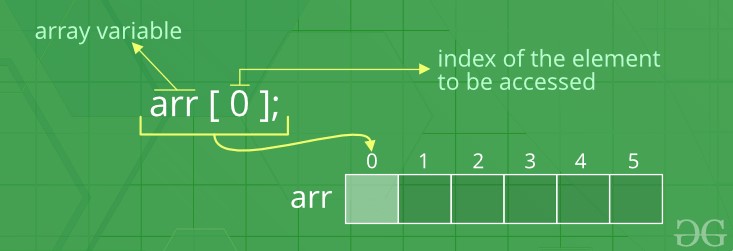
[Deletion in Arrays 5](#_Toc58445877)

# Introduction to Arrays

An array is a collection of items of the same data type stored at contiguous memory locations. This makes it easier to calculate the position of each element by simply adding an offset to a base value, i.e., the memory location of the first element of the array (generally denoted by the name of the array).  
  
For simplicity, we can think of an array as a fleet of stairs where on each step a value is placed (let’s say one of your friends). Here, you can identify the location of any of your friends by simply knowing the count of the step that they are on.  
  
**Remember**: “Location of the next index depends on the data type that we use”.  
  
The above image can be looked at as a top-level view of a staircase where you are at the base of the staircase. Each element can be uniquely identified by their index in the array (in a similar way where you could identify your friends by the step on which they were on in the above example).  
  
**Defining an Array**: Array definition is similar to defining any other variable. There are two things that are needed to be kept in mind, **the data type of the array elements** and the **size** of the array. The size of the array is fixed and the memory for an array needs to be allocated before use, the size of an array cannot be increased or decreased dynamically.  
  
Generally, arrays are declared as:

**dataType arrayName[arraySize];**  
  
An array is distinguished from a normal variable   
by brackets [ and ].

## Accessing array elements

Arrays allows to access elements randomly. Elements in an array can be accessed using indexes. Suppose an array named **arr** stores N elements. Indexes in an array are in the range of **0 to N-1**, where the first element is present at 0-th index and consecutive elements are placed at consecutive indexes. Element present at ith index in the array **arr[]** can be accessed as arr[i].  
  
The below image shows an array **arr[]** of size 5:  
  
  
**Advantages of using arrays:**

* Arrays allow random access of elements. This makes accessing elements by their position faster.
* Arrays have better [cache locality](https://en.wikipedia.org/wiki/Locality_of_reference) that can make a pretty big difference in performance.

**Examples**:

// A character array in C/C++/Java

char arr1[] = {'g', 'e', 'e', 'k', 's'};

// An Integer array in C/C++/Java

int arr2[] = {10, 20, 30, 40, 50};

// Item at i'th index in array is typically accessed

// as "arr[i]". For example arr1[0] gives us 'g'

// and arr2[3] gives us 40.

## Searching in an Array

Searching for an element in an array means to check if a given element is present in the array or not. This can be done by accessing elements of the array one by one starting from the first element and checking whether any of the elements matches with the given element.  
  
We can use [loops](https://www.geeksforgeeks.org/loops-in-c-and-cpp/) to perform the above operation of array traversal and access the elements, using indexes.  
  
Suppose the array is named **arr[**] with size **N** and the element to be searched is referred to as **key**. Below is the algorithm to perform the search operation in the given array.

for(i = 0; i < N; i++)  
{  
 if(arr[i] == key)  
 {   
 print "Element Found";  
 }  
 else  
 {  
 print "Element not Found";  
 }  
}

**Time Complexity** of this search operation will be O(N) in the worst case as we are checking every element of the array from 1st to last, so the number of operations is N.

# Insertion and Deletion of Arrays

## Insertion in Arrays

Given an array of a given size. The task is to insert a new element in this array. There are two possible ways of inserting elements in an array:

1. Insert elements at the end of the array.
2. Insert element at any given index in the array.

**Special Case**: A special case is needed to be considered is that whether the array is already full or not. If the array is full, then the new element cannot be inserted.

Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array.

* **Insert an element K at end in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

if(len < N)  
 // space left  
else  
 // array is full

If there is space left for the new element, insert it directly at the end at position **len + 1** and index **len**:

arr[len] = k;

***Time Complexity*** of this insert operation is constant, i.e. O(1) as we are directly inserting the element in a single operation.

* **Insert an element K at position, pos in arr[]**: The first step is to check if there is any space left in the array for new element. To do this check,

if(len < N)  
 // space left  
else  
 // array is full

Now, if there is space left, the element can be inserted. The index of the new element will be **idx = pos - 1**.  
  
Now, before inserting the element at the index idx, shift all elements from the index idx till end of the array to the right by 1 place. This can be done as:

for(i = len-1; i >= idx; i--)  
{  
 arr[i+1] = arr[i];  
}

After shifting the elements, place the element K at index idx.

arr[idx] = K;

***Time Complexity*** in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the right.

## Deletion in Arrays

To delete a given element from an array, we will have to first search the element in the array. If the element is present in the array then delete operation is performed for the element otherwise the user is notified that the array does not contains the given element.  
  
Consider the given array is **arr[]** and the initial size of the array is N, that is the array can contain a maximum of N elements and the length of the array is **len**. That is, there are *len*number of elements already present in this array.  
  
**Deleting an element K from the array arr[]**: Search the element K in the array arr[] to find the index at which it is present.

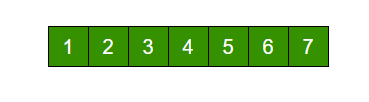
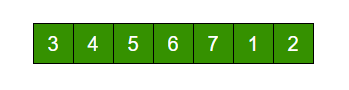
for(i = 0; i < N; i++)  
{  
 if(arr[i] == K)  
 idx = i; return;  
 else  
 Element not Found;  
}

Now, to delete the element present at index **idx**, left shift all of the elements present after *idx* by one place and finally reduce the length of the array by 1.

for(i = idx+1; i < len; i++)  
{  
 arr[i-1] = arr[i];  
}  
  
len = len-1;

***Time Complexity*** in worst case of this insertion operation can be linear i.e. O(N) as we might have to shift all of the elements by one place to the left.

# Array Rotation

As the term rotation signifies, array rotation means to rotate the elements of an array by given positions.  
  
Consider the below array:  
  
The above array is rotated counter-clockwise(towards left) by 2 elements. After rotation, the array will be:  
  
  
Visually, the process of counter clock-wise array rotation(rotated by say K elements) looks like:

* Shift all elements after K-th element to the left by K positions.
* Fill the K blank spaces at the end of the array by first K elements from the original array.

**Note**: The similar approach can also be applied for clockwise array rotation.

**Implementations**

* **Simple Method**: The simplest way to rotate an array is to implement the above visually observed approach by using extra space.  
  1. Store the first K elements in a temporary array say temp[].
  2. Shift all elements after K-th element to the left by K positions in the original array.
  3. Fill the K blank spaces at the end of the original array by the K elements from the temp array.

Say, arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2  
1) Store first K elements in a temp array  
 temp[] = [1, 2]  
2) Shift rest of the arr[]  
 arr[] = [3, 4, 5, 6, 7, 6, 7]  
3) Store back the K elements from temp  
 arr[] = [3, 4, 5, 6, 7, 1, 2]

**Time Complexity:** O(N), where N is the number of elements in the array.  
**Auxiliary Space:** O(K) where K is the number of places by which elements will be rotated.

* **Another Method (Without extra space)**: We can also rotate an array by avoiding the use of temporary array. The idea is to rotate the array one by one K times.

leftRotate(arr[], d, n)  
start  
 For i = 0 to i < d  
 Left rotate all elements of arr[] by one  
end

To rotate an array by 1 position to the left:

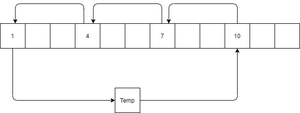
* 1. Store the first element in a temporary variable say temp.
  2. Left shift all elements after the first element by 1 position. That is, move arr[1] to arr[0], arr[2] to arr[1] and so on.
  3. Initialize arr[N-1] with temp.

**To rotate an array by K position to the left, repeat the above process K times.**  
Take the same example,

arr[] = [1, 2, 3, 4, 5, 6, 7], K = 2  
  
**Rotate arr[] one by one 2 times.**  
  
After 1st rotation: [2, 3, 4, 5, 6, 7, 1]  
After 2nd rotation: [ 3, 4, 5, 6, 7, 1, 2]

**Time Complexity:** O(N\*K), where N is the number of elements in the array and K is the number of places by which elements will be rotated.  
**Auxiliary Space:** O(1).

* **Juggling Algorithm**: This is an extension of the above method. Instead of moving one by one, divide the array in different sets, where number of sets is equal to GCD of N and K and move the elements within sets.  
    
  If GCD is 1 as is for the above example array (N = 7 and K = 2), then elements will be moved within one set only, we just start with temp = arr[0] and keep moving arr[I+d] to arr[I] and finally store temp at the right place.  
    
  Here is an example for N = 12 and K = 3. GCD of N and K is 3:

Let arr[] be {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12}  
  
a) Elements are first moved in first set – (See below   
 diagram for this movement)  
  
  
  
 arr[] after this step --> {4 2 3 7 5 6 10 8 9 1 11 12}  
  
b) Then in second set.  
 arr[] after this step --> {4 5 3 7 8 6 10 11 9 1 2 12}  
  
c) Finally in third set.  
 arr[] after this step --> {4 5 6 7 8 9 10 11 12 1 2 3}

**Time Complexity:** O(N), where N is the number of elements in the array.  
**Auxiliary Space:** O(1).