Array is a container which can hold a fix number of items and these items should be of the same type. Most of the data structures make use of arrays to implement their algorithms. Following are the important terms to understand the concept of Array.

* **Element** − Each item stored in an array is called an element.
* **Index** − Each location of an element in an array has a numerical index, which is used to identify the element.

Array Representation

Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



Arrays can be declared in various ways in different languages. For illustration, let's take C array declaration.



As per the above illustration, following are the important points to be considered.

* Index starts with 0.
* Array length is 10 which means it can store 10 elements.
* Each element can be accessed via its index. For example, we can fetch an element at index 6 as 9.

Basic Operations

Following are the basic operations supported by an array.

* **Traverse** − print all the array elements one by one.
* **Insertion** − Adds an element at the given index.
* **Deletion** − Deletes an element at the given index.
* **Search** − Searches an element using the given index or by the value.
* **Update** − Updates an element at the given index.

Traverse Operation

This operation is to traverse through the elements of an array.

Example

Following program traverses and prints the elements of an array:

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Insertion Operation

Insert operation is to insert one or more data elements into an array. Based on the requirement, a new element can be added at the beginning, end, or any given index of array.

Here, we see a practical implementation of insertion operation, where we add data at the end of the array −

Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/YpAUzN)

#include <stdio.h>

main() {

int LA[] = {1,3,5,7,8};

int item = 10, k = 3, n = 5;

int i = 0, j = n;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

n = n + 1;

while( j >= k) {

LA[j+1] = LA[j];

j = j - 1;

}

LA[k] = item;

printf("The array elements after insertion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after insertion :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 10

LA[4] = 7

LA[5] = 8

For other variations of array insertion operation [click here](https://www.tutorialspoint.com/data_structures_algorithms/array_insertion_algorithm.htm)

Deletion Operation

Deletion refers to removing an existing element from the array and re-organizing all elements of an array.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to delete an element available at the Kth position of LA.

1. Start

2. Set J = K

3. Repeat steps 4 and 5 while J < N

4. Set LA[J] = LA[J + 1]

5. Set J = J+1

6. Set N = N-1

7. Stop

Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/3FCSPD)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

j = k;

while( j < n) {

LA[j-1] = LA[j];

j = j + 1;

}

n = n -1;

printf("The array elements after deletion :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after deletion :

LA[0] = 1

LA[1] = 3

LA[2] = 7

LA[3] = 8

Search Operation

You can perform a search for an array element based on its value or its index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to find an element with a value of ITEM using sequential search.

1. Start

2. Set J = 0

3. Repeat steps 4 and 5 while J < N

4. IF LA[J] is equal ITEM THEN GOTO STEP 6

5. Set J = J +1

6. PRINT J, ITEM

7. Stop

Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/613FSK)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int item = 5, n = 5;

int i = 0, j = 0;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

while( j < n){

if( LA[j] == item ) {

break;

}

j = j + 1;

}

printf("Found element %d at position %d\n", item, j+1);

}

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

Found element 5 at position 3

Update Operation

Update operation refers to updating an existing element from the array at a given index.

Algorithm

Consider **LA** is a linear array with **N** elements and **K** is a positive integer such that **K<=N**. Following is the algorithm to update an element available at the Kth position of LA.

1. Start

2. Set LA[K-1] = ITEM

3. Stop

Example

Following is the implementation of the above algorithm −

[Live Demo](http://tpcg.io/EQ4FEy)

#include <stdio.h>

void main() {

int LA[] = {1,3,5,7,8};

int k = 3, n = 5, item = 10;

int i, j;

printf("The original array elements are :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

LA[k-1] = item;

printf("The array elements after updation :\n");

for(i = 0; i<n; i++) {

printf("LA[%d] = %d \n", i, LA[i]);

}

}

When we compile and execute the above program, it produces the following result −

Output

The original array elements are :

LA[0] = 1

LA[1] = 3

LA[2] = 5

LA[3] = 7

LA[4] = 8

The array elements after updation :

LA[0] = 1

LA[1] = 3

LA[2] = 10

LA[3] = 7

LA[4] = 8

**Parallel Array:**Also known as structure an array (SoA), multiple arrays of the same size such that i-th element of each array is closely related and all i-th elements together represent an object or entity. An example parallel array is two arrays that represent x and y co-ordinates of n points.

Below is another example where we store the first name, last name and heights of 5 people in three different arrays.

first\_name = ['Bones', 'Welma', 'Frank', 'Han', 'Jack']

last\_name = ['Smith', 'Seger', 'Mathers', 'Solo', 'Jackles']

height = [169, 158, 201, 183, 172]

This way we could easily store the information and for accessing, the first index of each array corresponds to the data belonging to the same person.  
  
**Application:**  
Two of the most essential applications performs on an array or a record are searching and sorting.

* **Searching:** Each index in a parallel array corresponds to the data belonging to the same entity in a record. Thus, for searching an entity based on a specific value of an attribute, e.g. we need to find the name of a person having height >200 cms in the above example. Thus, we search for the index in the height array having value greater than 200. Now, once we have obtained the index, we print the values of the index in the first\_name and last\_name arrays. This way searching becomes an easy task in parallel array.
* **Sorting:** Now, using the same fact that each index corresponds to data items in different arrays corresponding to the same entity. Thus, we sort all arrays based on the same criteria. For example, in the above-displayed example, we need to sort the people in increasing order of their respective heights. Thus, when we swap the two heights, we even swap the corresponding values in other arrays using the same index.

**Implementation:**  
1) The code below stores the first name, second name, and height of 10 students.  
2) Sorts them in increasing order of the height using [quicksort](http://www.geeksforgeeks.org/quick-sort/" \t "_blank) algorithm.  
3) Searches the name of the 2nd tallest student, the 3rd shortest student and the student having a height equal to 158 cms in the record.

**Advantages:**

* They can be used in languages which support only arrays of primitive types and not of records (or perhaps don’t support records at all).
* Parallel arrays are simple to understand and use, and are often used where declaring a record is more trouble than it’s worth.
* They can save a substantial amount of space in some cases by avoiding alignment issues.
* If the number of items is small, array indices can occupy significantly less space than full pointers, particularly on architectures with large words.
* Sequentially examining a single field of each record in the array is very fast on modern machines, since this amounts to a linear traversal of a single array, exhibiting ideal locality of reference and cache behavior.

**Disadvantages:**

* They have significantly worse locality of reference when visiting the records non-sequentially and examining multiple fields of each record.
* They have little direct language support (the language and its syntax typically express no relationship between the arrays in the parallel array).
* They are expensive to grow or shrink since each of several arrays must be reallocated. Multi-level arrays can ameliorate this problem, but impacts performance due to the additional indirection needed to find the desired elements.

**sparse array**

A sparse array is an array of data in which many elements have a value of zero. This is in contrast to a dense array, where most of the elements have non-zero values or are “full” of numbers. A sparse array may be treated differently than a dense array in digital data handling.

* A sparse array is simply an array most of whose entries are zero (or null, or some other default value)
* For example: Suppose you wanted a 2-dimensional array of course grades, whose rows are Penn students and whose columns are courses
  + There are about 22,000 students
  + There are about 5000 courses
  + This array would have about 110,000,000 entries
  + Since most students take fewer than 5000 courses, there will be a lot of empty spaces in this array
  + This is a big array, even by modern standards
* There *are* ways to represent sparse arrays efficiently
* We will start with sparse one-dimensional arrays, which are simpler
  + We’ll do sparse two-dimensional arrays later
* Here is an example of a sparse one-dimensional array:
* Here is how it could be represented as a linked list:

True fact: In an ordinary array, indexing to find an element is the only operation we really need

True fact: In a sparse array, we can do indexing reasonably quickly