Arrays

An Array is a collection of similar data elements grouped under a single name.

```
int a[5];
```

Memory allocated will be in a contiguous order. i.e, all the elements are side by side.

Elements can be accessed using indices starting from 0.

Declaration and Initialization

1. `int A[5];`

This syntax will declare an array, but will not initialize it. Accessing an uninitialized array gives garbage(useless) values.

2. `int A[5] = {2, 4, 6, 8, 10};`

This syntax will declare and initialize an array with the values specified in the curly braces.

3. `int A[5] = {2,4};`

This declaration will initialize the first 2 elements with 2, 4 and rest will be initialized with 0.

4. `int A[5] = {0};`

This will declare and initialize an array filled with zeroes.

5. `int A[] = {2, 4, 6, 8, 10};`

Omitting the size is legal, if the array is also initialized.

Accessing

1.

```
int A[5] = {2, 4, 6, 8, 10};
printf("%d", A[1]);  // O/P - 4
```

2.

```
for (int i = 0; i < 5; i++)
{
    printf("%d ", A[i]); // 2 4 6 8 10
}
```

3.

```
printf("%d", 2[A]); // 6
```

4.

```
printf("%d", *(A+2)); // 6
```

Static vs Dynamic Arrays

Static Arrays: When an array of constant size is created, the memory will be allocated in the stack as an activation record. In `C` , the memory for the array is allocated at compile time so the size must be known beforehand whereas in `C++` it can be determined at runtime also.

Dynamic Arrays: When the memory for an array is allocated in the heap, it is known as a dynamic array. The array itself may not change in size but some alternate methods can be used(discussed below). Array in a heap, returns a pointer to itself.

Increasing size of an Array

Size of an array, once allocated, cannot be changed but we can create an array of larger size and can copy the elements in the larger array. For copying we can use `for` loop or `memcpy()` .

Declaration and Initialization of 2D Arrays

1.

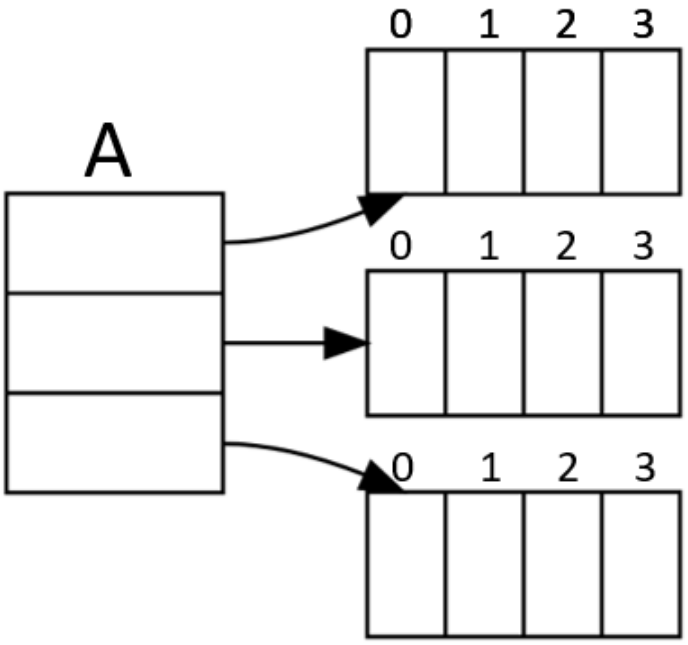
```
int A[3][4] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};
```

This declaration of the array will create an array with 3 rows and 4 columns. The memory is allocated linearly in the system but we can access elements with 2 indices.

2.

```
int* A[3];
```

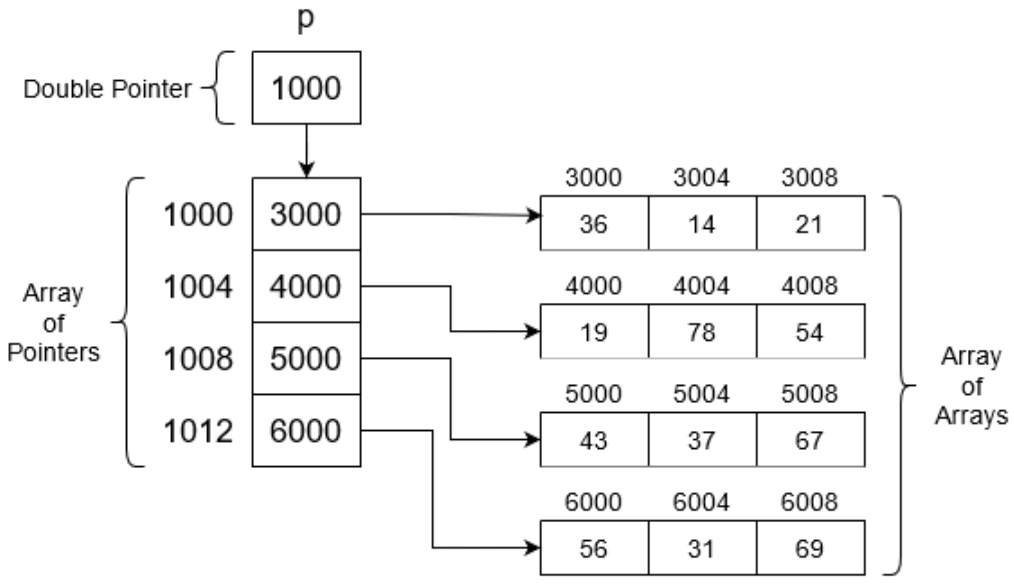
This declaration creates an array of 3 pointers. We can then use those pointers with new or malloc() to point them to any array of arbitrary size.



3.

```
int** A;
A = new int*[3];
A[0] = new int[4];
A[1] = new int[4];
A[2] = new int[4];
```

Here we use a double pointer(pointer to pointer) to point to an array of pointers. Pointers in arrays can then be pointed to arrays of their own.



Accessing

We can access an array using nested for loop by

```
for (int i = 0; i < n; i++)
{
    for (int j = 0; j < m; j++)
    {
        printf("%d ", arr[i][j]);
    }
    printf("\n");
}
```

Representation of 2D Arrays in compiler

Code that we write gets converted to machine code and machine code doesn't have variable names but memory addresses. During execution, every variable is assigned to a memory address.
But the compiler can't know the address before runtime i.e before execution of the program.

So to resolve this issue, the compiler keeps a base address of the array and by using the base address calculates the memory address of a particular element. There are 2 ways compiler can map this address in RAM:-

1. Row-major Mapping:

```
int A[3][4];
      m x n
```

address(`A[i][j]`) = $L_0 + [i * n + j] * w$

L_0 = Base address
n = maximum number of columns
w = size of data type

2. Column-major Mapping:

```
int A[3][4];
      m x n
```

address(`A[i][j]`) = $L_0 + [j * n + i] * w$

L_0 = Base address
m = maximum number of rows
w = size of data type

Representation for 3D Arrays

1. Row-major:

```
int A[l][m][n];
    --------->
```

address(`A[i][j][k]`) = $L_0 + [i * m * n + j * n + k] * w$

2. Column-major:

```
int A[l][m][n];
    <---------
```

address(`A[i][j][k]`) = $L_0 + [k * l * m + j * l + i] * w$

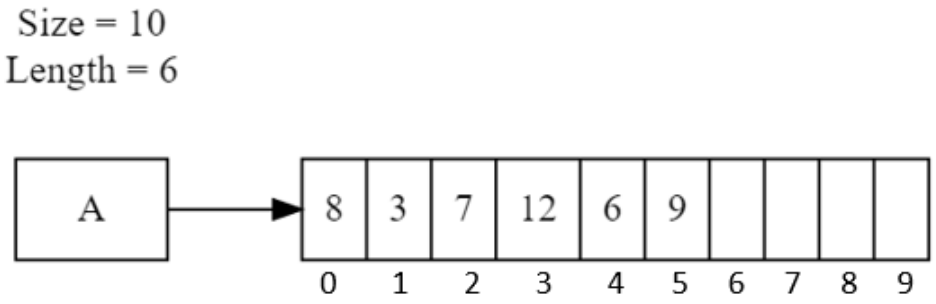
Array as Abstract Data Type

Abstract data type means representation of data and the set of operations that can be performed on it.

Array is a very basic data type and almost all languages have it. Therefore, the representation is done by the compiler itself but the operations have to be implemented by us.

Representation of Data:

1. **Array space:** Can be in stack or heap(static or dynamic).
2. **Size:** Required space for array.
3. **Length:** Number of elements present in array. If the array is full, it means length and size are the same.



Operations

Some operations that can be performed on Array ADT are:

Display()

We can display all the elements of the array using a simple for loop.

Pseudocode

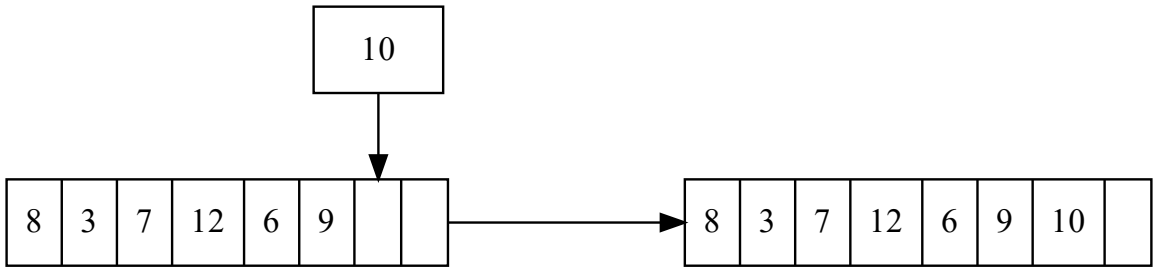
```
for(i = 0; i < Length; i++)
{
    print(A[i]);
}
```

Analysis

Time - O(n)

Add(x)/Append(x)

Appending an element is simply adding it to the end of the array i.e position after the last element.



Pseudocode

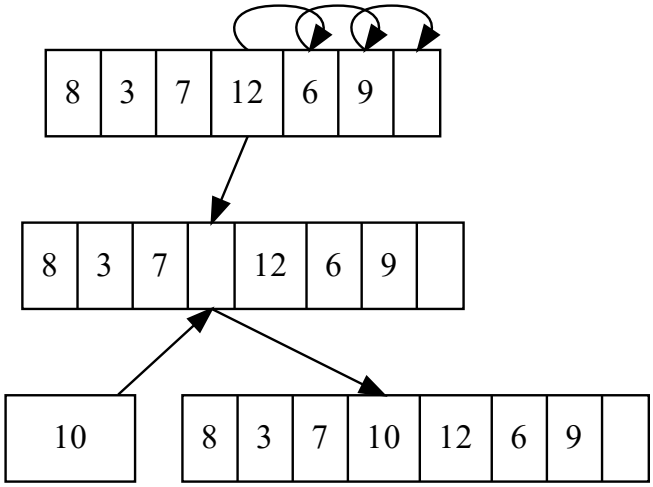
```
A[Length] = x;
Length++;
```

Analysis

Time - O(1)

Insert(index, x)

With Insert() we can insert any element at any given index. We start from the right side and copy the elements in the previous position until we get to the index that we want. Then we set that index's value to x and increase the length by 1.



Pseudocode

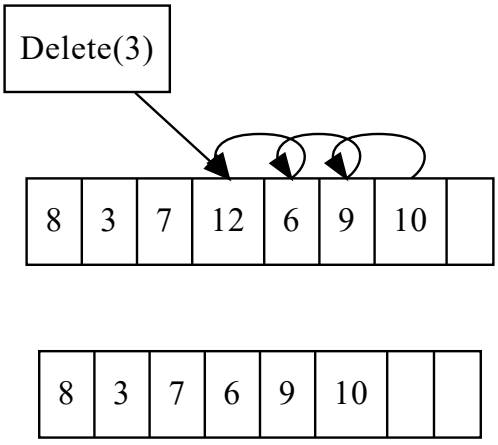
```
for(i = Length; i > index; i--)
{
    A[i] = A[i-1];
}
A[index] = x;
Length++;
```

Analysis

$$Time - \begin{cases} Min - O(1) \\ \\ Max - O(n) \end{cases}$$

Delete(index)

Deleting an element is removing it from the array. Whenever an element gets deleted, the position which it occupied should not be left vacant, otherwise more work will be required when later we access the elements of the array. Therefore, we shift all the elements which are right to that deleted position by 1 place to left.



Pseudocode

```
x = A[index];
for(i = index; i < Length-1; i++)
{
    A[i] = A[i+1];
}
Length--;
```

Analysis

$$Time - \begin{cases} Min - O(1) \\ \\ Max - O(n) \end{cases}$$

Search(x)

i. Linear Search

In this method, we go through the whole array and visit every location at least once until the element is found. Element to be searched is also called "Key". It can be successfully executed if the element is found and will return its index otherwise in case of unsuccessful search, it will return an invalid index.

Pseudocode

```
for (i = 0; i < Length; i++)
{
    if (Key == A[i])
    {
        return i;
    }
}
return -1;  // unsuccessful search
```

Analysis

$$Time - \begin{cases} Best & - O(1) \\ \\ Average & - O(n) \\ \\ Worst & - O(n) \end{cases}$$

Improvements in Linear search

1. Transposition: If an element is searched repeatedly, we can swap the element with the previous position every time it is asked to be searched, This will help in reducing the time complexity by a few units.

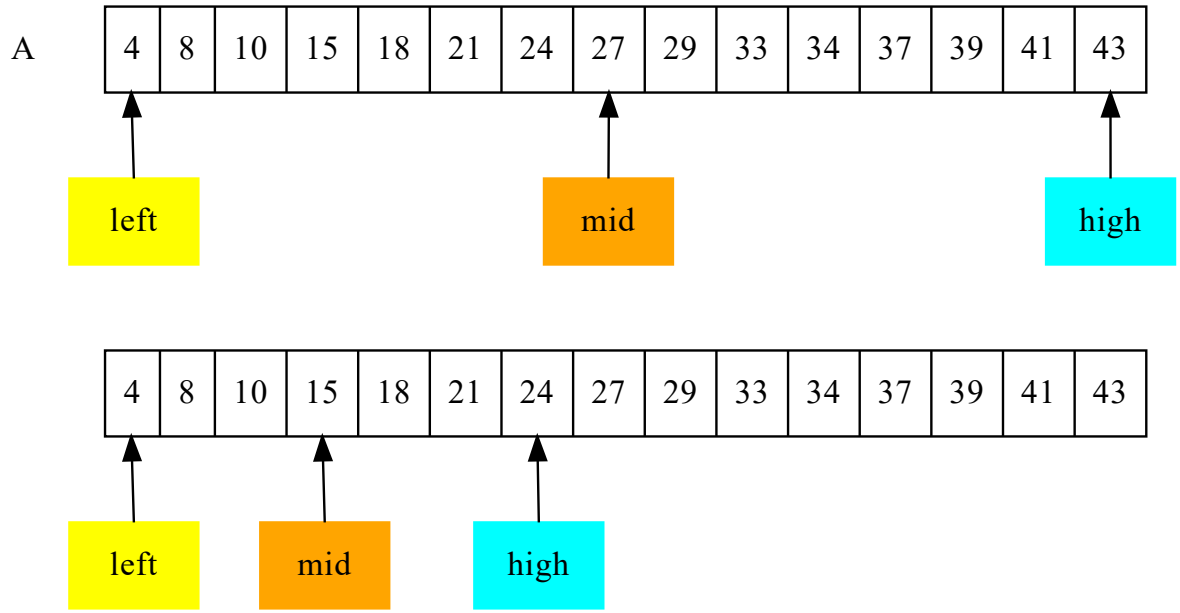
2. Move to Front/Head: If an element is searched repeatedly, that element can be shifted to first position so that it can be accessed in constant time, next time around. This method drastically reduces access time.

ii. Binary Search

We start with two indices in a loop, `low` and `high` at extreme ends of the array. We have a `mid` index which can be known as low and high using the formula $\left\lfloor \frac{l+h}{2} \right\rfloor$. For every iteration we can compare with `mid`, If the key(element to be searched) is less than `mid` index element, we update `high = mid - 1` and if the key is higher than mid index element, then `low = mid + 1`. We will terminate the loop, if mid is equal to key's index. The most **important** condition for binary search is the array has to be **sorted**.

```
Algorithm BinSearch(l, h, key)
{
    while (l <= h)
    {
        mid = floor((l+h)/2)
        if (key == A[mid])
        {
            return mid;
        }
        else if (key < A[mid])
        {
            h = mid - 1;
        }
        else
        {
            l = mid + 1;
        }
    }
    return -1;
}
```

Example: Searching for `key = 15` in this sorted array would look like this



Analysis

$$Time - \begin{cases} Best & -O(1) \\ Average & -O(logn) \\ Worst & -O(logn) \end{cases}$$

Get(index)

For getting an element in an array. We check if the index given is valid or not.

Pseudocode

```
if(index >= 0 && index < Length)
{
    return A[index];
}
```

Set(index, x)

This method is used to replace the value in an array. We again check if the index is valid or not.

Pseudocode

```
if(index >= 0 && index < Length)
{
    A[index] = x;
}
```

Max()/Min()

i. Max()

We traverse through the entire array and keep a variable max, which will be changed if an element greater than max is found.

Pseudocode

```
max = A[0];
for(i = 1;i < Length; i++)
{
    if (A[i] > max)
    {
        max = A[i];
    }
}
return max
```

ii. Min()

Similar to Max(), but here we change the if condition to accomodate for Min().

Pseudocode

```
min = A[0];
for(i = 1; i < Length; i++)
{
    if (A[i] < min)
    {
        min = A[i];
    }
}
return min;
```

Analysis

Time - O(n)

Sum()

For finding the sum of all the elements, we must traverse the whole array and keep a tally of total.

Pseudocode

```
sum = 0;
for (i = 0; i < Length; i++)
{
    sum += A[i];
}
return sum;
```

Recursive definition

$$Sum(n) = \begin{cases} 0 & n < 0 \\ Sum(n-1) + A[n] & n >= 0 \end{cases}$$

Pseudocode

```
Sum(A, n)
{
    if (n < 0) return 0;
    else return Sum(A, n-1) + A[n];
}
```

Average()

Pseudocode

```
avg = 0;
for (i = 0; i < Length; i++)
{
    avg += A[i];
}
return avg/Length;
```

Reverse()

i. Using Auxiliary Array:

Here an extra array of the same length as the array to be reversed is used. The elements of the initial array are copied to the new array from the back. After that, the elements of the new array are then copied to the initial array, which therefore reverses it.

A	9	18	2	6	7	4

B	4	7	6	2	18	9

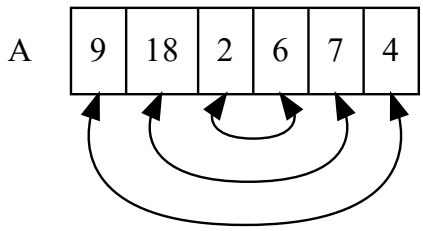
Pseudocode

```
for (i = Length - 1, j = 0; i >= 0; i--, j++)
{
    B[i] = A[i];  // loop for reverse copying A to B
}

for (k = 0; k < Length; k++)
{
    A[i] = B[i];  // loop for copying B to A
}
```

ii. Using two pointers

Here we use two integers; i, j, as logical pointers, with i assigned to the start of the array and j at the end. A loop is then used to swap the elements at index i and j and then i is incremented and j is decremented, until i > j.



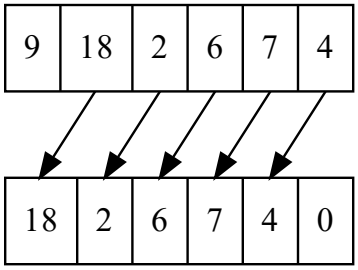
Pseudocode

```
for (i = 0, j = Length - 1; i < j; i++, j--)
{
    temp = A[i];
    A[i] = A[j];
    A[j] = temp;
}
```

Shift()/Rotate()

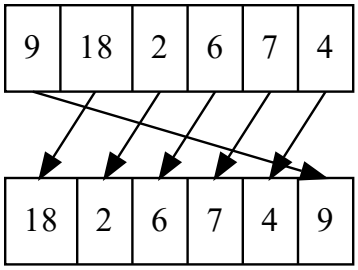
Left Shift

When the elements of the array are shifted to the previous position, it is called left shift. First element will be lost in the left shift and the last element will become zero.



Left Rotate

It is similar to left shift with the difference being that the first element will now be moved to the last position in the array.

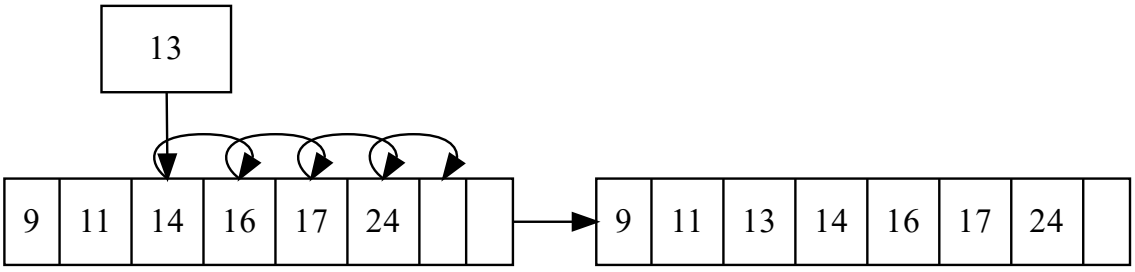


Right shift/rotate are the same as that of left shift/rotate with the difference only in direction.

This function is useful in the case of LED billboards where the text is moving around.

Inserting in a sorted array

Given a sorted array of size n, we start by shifting from right until the number smaller than the element to be inserted is found.




```
Insert(x)
{
    i = Length - 1;
    while (A[i] > x)
    {
        A[i+i] = A[i];
        i--;
    }
    A[i+1] = x;
    Length++;
}
```

Checking for sorted array

We start from the first element and compare it to the next element. If the previous element is smaller(in ascending order) than the next element, we continue until the condition is false.

```
IsSorted(A, n)
{
    for (i = 0; i < n-1)
    {
        if (A[i] > A[i+1])
            return false;
    }
    return true;
}
```

Analysis

Time: O(n)

Moving Negative elements to the left side

Here we have to bring all the negative numbers to the left side of the array and after that we can bring positive numbers to the right side. For that we use `i` and `j` as two indices to find the positive and negative numbers, respectively. We continue moving `i` until we encounter a positive number and we continue moving `j` until we find a negative number. After that, when `i` and `j` have their respective numbers indexed in them, we then swap those numbers.

```
i = 0;
j = Length - 1;
while (i < j)
{
    while (A[i] < 0) i++;  // continue moving i till
    while (A[j] >= 0) j++; // positive number is found, and j
    if (i < j)             // until negative number is found
        swap(A[i], A[j]);
}
```

Analysis

Time: O(n)

Merging two arrays

We can merge 2 sorted arrays, to get a bigger array which is also sorted. We take 3 indices `i, j, k` and compare `i` to `j` and if `i` is smaller, we insert element at index `i` to third array which of `size = size(arr1) + size(arr2)`. We insert `j` otherwise.

```
Merge()
{
    int i = 0, j = 0, k = 0;
    while (i < m && j << n)
    {
        if (A[i] < B[j])
            C[k++] = A[i++];
        else
            C[k++] = B[j++];
    }

    while (i < m)
        C[k++] = A[i++];
    while (j < n)
        C[k++] = B[j++];
}
```

Challenges

1. Finding missing elements

i. Single missing element in a sorted array

a. Natural Numbers starting from 1:

A	1	2	3	4	5	6	7	9	10	11	12

We can find the missing element by summing all the elements in the array and then subtracting it from the formula $\frac{n(n+1)}{2}$, which gives the sum of elements if the element was not missing.

$$Sum(A) = 70$$
$$ActualSum(A) = 78$$
$$\therefore MissingNumber = 78 - 70 = 8$$

Code

```
sum = 0;
for (int i = 0; i < Length; i++)
{
    sum += A[i];
}
actualSum = n * (n+1) / 2;
return actualSum - sum;
```

b. Natural numbers starting from any number:

A	6	7	8	9	10	11	13	14	15	16	17

Here we can make use of indices, we can find the difference between the element at index `i` and `i` itself. Wherever the difference is not the same as the previous element, we can return the difference + the index at which it isn't the same, to get the missing element.

6(element at A[0]) - 0(index 0) = 6
7 - 1 = 6
8 - 1 = 6
.
.
.
13 - 6 = 7 ❌ $\therefore$ we add $6(\text{diff}) + 6(\text{index}) = 12$

Code

```
diff = A[0]; // A[0] - 0
for (int i = 0; i < Length; i++)
{
    if (A[i] - i != diff)
    {
        return diff + i;
    }
}
```

ii. Multiple missing elements in a sorted array

a.

A	6	7	8	9	11	12	15	16	17	18	19

Here we use the technique used above, where we find differences and when we find a change in difference, we print the diff + index, which is the missing number. Now to find multiple missing numbers, we add 1 to diff and repeat the above process until the end of the array.

6 - 0 = 6
7 - 1 = 6
8 - 2 = 6
.
.
.

11 - 4 = 5 ∴ 6 + 4 = 10 is missing, now we increment diff by 1.

12 - 5 = 7

15 - 6 = 9

∴ Some elements are missing between the index 5 and 6, we can find them by adding diff and index, until the diff is not equal to A[i] - i.

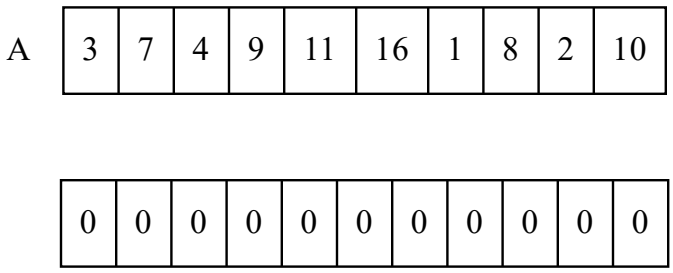
6 + 7 = 13; diff++

6 + 8 = 14; diff++

Now the diff is the same as 9(15 - 6) so we can continue.

```
diff = A[0];
for (int i = 0; i < Length; i++)
{
    if (A[i] - i != diff)
    {
        while (diff < arr[i] - i)
        {
            printf("%d ", diff + i);
            diff++;
        }
    }
}
```

b. Using HashTable:



Here we use an auxiliary array of size maximum element in the array which is initialized with all 0. Now every time we encounter an element, we take that element as the index in the aux array and increment by 1. Now we go through the aux array and print all the indexes whose value is still 0.

In this technique, we decrease the time complexity but the space complexity increases. The aux array here is called a **HashTable**. In this HashTable, the key is the index and the value is the count of that index in the original array.

Code

```
int H[max(A)];
for (in i = 0; i < Length; i++)
{
    H[A[i]]++;
}
for (int i = 0; i < max(A); i++)
{
    if (H[i] == 0)
    {
        printf("%d ", i)
    }
}
```

2. Finding duplicate elements

a. In a sorted array:

A	3	6	8	8	10	12	15	15	15	20

i.

We have to find the duplicate elements in the given array. We start at the first element and check if it is equal to the next element. We also check if it is not equal to the last duplicate element found. If the condition is true we can print the element and set the last duplicate to that element so if there are more than 1 duplicates, it skips them.

Code

```
lastDup = 0;
for (int i = 0; i < Length - 1; i++)
{
    if (A[i] == A[i+1] && A[i] != lastDup)
    {
        printf("%d ", A[i]);
        lastDup = A[i];
    }
}
```

ii.

A	3	6	8	8	10	12	15	15	15	20

In this method, we check if the element at index `i` is equal to the element at index `i+1`. Then we set j to i+1 and got into a while loop to check if there are more than one duplicates and print them.

Code

```
for (int i = 0; i < Length - 1; i++)
{
    if (A[i] == A[i+1])
    {
        j = i+1;
        while (A[j] == A[i]) j++;
        printf("%d appears %d times", A[i], j-i);
        i = j-1;
    }
}
```

iii.

A	3	6	8	8	10	12	15	15	15	20

Code

```
for (int i = 0; i < Length; i++)
{
    H[A[i]]++;
}
for (int i = 0; i <= max(A); i++)
{
    if (H[i] > 1)
    {
        printf("%d appears %d times", H[i], i);
    }
}
```

b. In an Unsorted array:

A	3	6	8	8	10	12	15	15	15	20

H	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Code

```
for (int i = 0; i < Length; i++)
{
    count = 1;
    if (A[i] != -1)
    {
        for (int j = i+1; i < Length; j++)
        {
            if (A[i] == A[j])
            {
                count++;
                A[j] = -1;
            }
        }
        if (count > 1)
        {
            printf("%d appears %d time", A[i], count);
        }
    }
}
```