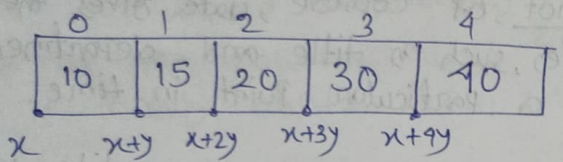
Rahul Biswas

Array Data Structure (Introduction)

arr[] = {10, 15, 20, 30, 40}

Contiguous memory

```
a = arr[0];
b = arr[2]; // x+2y
```



x = Address where array is stored

y = Size of an array element

Advantages:-

1. Random Access
2. Cache friendliness

Array Data structure (Types)

Types:

1. Fixed Sized arrays
2. Dynamic Sized arrays

1. Fixed Sized array:

```
int[] arr = new int[100]
int[] arr = new int[n]
int arr = {10, 15, 30, 40}
```

2. Dynamic Sized Arrays:

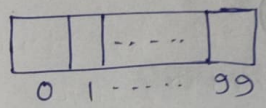
Resize Automatically

C++ : vector

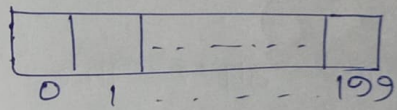
Java : Arraylist

Python : list

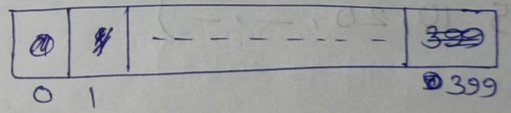
Initial



When Become Full First Time



When Become Full Second Time



Operations on Arrays

1. Searching (unsorted Array)

I/P : arr[] = {20, 5, 7, 25}
0 1 2 3

x = 5

O/P = 1

I/P : arr[] = {20, 15, 5, 7}

x = 25

O/P = -1

Sol

```
int search (int arr[], int n, int x)
{
    for(int i=0; i<n; i++)
        if (arr[i] == x)
            return i;
    return -1;
}
```

Linear search →

Time complexity = O(n)

Insert

I/P : arr[] = {5, 10, 20, —, —}

x = 7

Pos = 2

O/P : arr[] = {5, 7, 10, 20, —}

x = 3

Pos = 2

O/P : arr[] = {5, 3, 7, 10, 20}

Sol

```
int insert (int arr[], int n, int x, int cap,
            int pos)
{
    if (n == cap)
        return n;
    int idx = pos-1;
    for (int i = n-1; i >= idx; i--)
        arr[i+1] = arr[i];
    arr[idx] = x;
    return (n+1);
}
```

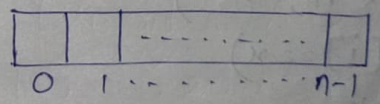
Time complexity : O(n)

Insert at the End : O(1)

Insert at the Beginning : O(n)

Insert at the End for Dynamic Sized Array

Initial Capacity :



Time complexity of every insert = O(1)
For first n inserts

Average Time complexity for (n+1)

Inserts at the end $= \frac{O(1)+O(1)+\ldots+O(n)+\Theta(n)}{n+1}$

$= O(1)$

Deletion :

I/P : arr[] = {3, 8, 12, 5, 6}

x = 12

O/P : arr[] = {3, 8, 5, 6, -}

I/P : arr[] = {3, 8, 12, 5, 6}

x = 6

O/P : arr[] = {3, 8, 12, 5, —}

Sol

```
int deleteEle (int arr[], int n, int x)
{
    int i;
    for (i=0; i<n; i++)
        if (arr[i]==x)
            break;
    if (i==n)
        return n;
    for (int j=i; j<n-1; j++)
        arr[j] = arr[j+1];
    return (n-1);
}
```

Time Complexity

Insert : O(n)

Search : O(n) for unsorted

O(log n) for sorted

Delete : O(n)

Get i-th Element : O(1)

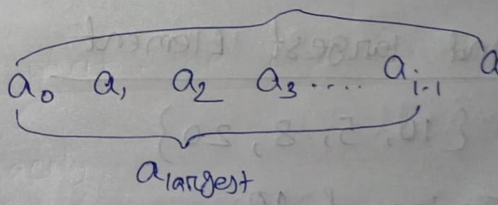
Update i-th Element : O(1)

Note: Insert at the end and delete from the end can be done in O(1) time.

Find the index of largest element:

I/P : arr = [40, 8, 50, 100]

O/P : 3 // Index of 100



$a_i <= a_{largest}$: Ignore

$a_i > a_{largest}$: largest = i

arr[] = {5, 8, 20, 10}

res = 0;

i = 1 : res = 1

i = 2 : res = 2

i = 3 : res = 2

Sol

```
int getlargest (int arr[])
{
    int res = 0;
    for (int i = 1; i < arr.length; i++)
        if (arr[i] > arr[res])
            res = i;
    return res;
}
```

Time Complexity = $\Theta(n)$

Second largest Element

I/P : arr[] = {10, 5, 8, 20}

O/P : 0 // Index of 10

I/P : arr[] = {20, 10, 20, 8, 12}

O/P : 4 // Index of 12

I/P : {10, 10, 10} = arr[]

O/P : -1 // No Second largest

Efficient Approach

```
int secondlargest (int arr[], int n)
{
    int res = -1, largest = 0;
    for (int i = 1; i < n; i++)
    {
        if (arr[i] > arr[largest])
        {
            res = largest;
            largest = i;
        }
        else if (arr[i] != arr[largest])
        {
            if (res == -1 || arr[i] > arr[res])
                res = i;
        }
    }
    return res;
}
```

Move all zeros to end

I/P : arr[] = {8, 5, 0, 10, 0, 20}

O/P : arr[] = {8, 5, 10, 20, 0, 0}

I/P : arr[] = {0, 0, 0, 10, 0}

O/P : arr[] = {10, 0, 0, 0, 0}

I/P : arr[] = {10, 20}

O/P : arr[] = {10, 20}

Native Solution:

```
Void moveToEnd (int arr[], int n)
{
    for (int i=0; i<n; i++)
    {
        if (arr[i]==0)
        {
            for (int j= i+1; j<n; j++)
                if (arr[j]!=0)
                    swap(arr[i], arr[j])
        }
    }
}
```

Efficient Solution:

```
Void movezeros (int arr[], int n)
{
    int count = 0;
    for (int i=0; i<n; i++)
    {
        if (arr[i]!=0)
        {
            swap(arr[i], arr[count]);
            count++;
        }
    }
}
```

Left Rotate an Array by one

I/P : arr[] = {1, 2, 3, 4, 5}
O/P : arr[] = {2, 3, 4, 5, 1}

I/P : arr[] = {30, 5, 20}
O/P : arr[] = {5, 20, 30}

```
Void lRotateOne (int arr[], int n)
{
    int temp = arr[0];
    for (int i=1; i<n; i++)
        arr[i-1] = arr[i];
    arr[n-1] = temp;
}
```

Time Complexity = $\Theta(n)$

Reverse an Array

I/P : arr[] = ~~{30, 7, 6, 5, 10}~~ {10, 5, 7, 30}
O/P : arr[] = ~~{10, 5, 6, 7, 30}~~ {30, 7, 5, 10}

① 10 (low=0) 5 7 30 (high=3)

② 30 5 (low=1) 7 (high=2) 10

③ 30 7 (high=1) 5 (low=2) 10

```
Void reverse (int arr[], int n)
{
    int low = 0, high = n-1;
    while (low < high)
    {
        int temp = arr [low];
        int temp = arr [high];
        arr [high] = temp;
        low ++;
        high --;
    }
}
```

swap arr [low] and arr [high]