**Data Structures**

The efficiency of any data structure is rated based on some basic operations like **Accessing, Searching, Inserting** and **Deleting**.

Every data structure has its own advantages, some are good at searching, some are good at both inserting and accessing and some are good at deleting.

Based on the operation we want to perform we will choose the data structure to use. Let say we have 2 data structures X, Y. so, X is good at inserting and Y is good at searching.

We have a scenario like finding the element then we should go for data structure Y. we have a scenario like data is coming continuously and we want to store it then we should go for data structure X.

Now, let’s dig into the data structures. We will start with array.

**Array**

Array is most commonly and widely used data structure in computer programming.

Array is nothing but a collection of elements or items that are stored at contiguous memory locations.

Storing collection/group of similar data type elements/items as a single entity is nothing but an array.

**Note:** The elements in an array should be same data type.

Each array will have 3 properties associated.

1. Name of the array
2. Type of the array
3. Size of the array

Ex:

Names = [‘vissu’, ‘dinesh’, ‘dileep’]

Salaries = [25000, 35000, 30000]

**Parallel arrays**

Parallel arrays are nothing but two or more arrays having same size and each element in an array relate to the element in other array at same location. If we look at above two arrays names and salaries, we can say that vissu having salary of 25000. This means they both elements are related with each other.

**NOTE:** The size of an array is fixed upon creation of the array. This means while creating an array we need to specify the size of array. This size can’t be changed later. Size represents the total number of elements that we can store in an array.

**Defining an array**

we can create array in two ways.

1. Creating array and populating with elements instantly.

In Java

int array [ ] = {1, 2, 3};

In Python

Import array as arr

A = arr. array (“i”, [1, 2, 3]) # i ==🡺 integer array

This will create an integer array with size 3.

1. Creating array and populating elements later.

In Java

int array [ ] = new int [10];

In Python

Not possible

This will create an integer array with size 10. This size can’t be changed.

**Accessing elements in the array (array Indexing)**

Array will have indexing which starts from 0 instead of 1.

Let’s say we have an array with 10 elements the indexing of the array will have 0- 9.

If we want to access element at 5th index we will use “array\_name[index]”

Ex:

Int numbers [] = {1,2,3,4,5,6,7,8,9,10}

numbers [5]

**Modifying the elements in the array**

Ex:

numbers [5] = 60 # This will update/modify the value at 5th index with new value.

**Time complexity for accessing an element in an array:**

Time complexity of accessing elements in an array is O (1).

Let say we have an integer array of size 5.

EX:

int num\_arr [] = new int [5];

Now look at memory allocation for this array in computer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INDEX** | 0 | 1 | 2 | 3 | 4 |
| **TYPE** | Int | Int | Int | Int | Int |
| **VALUE** | 1 | 3 | 5 | 7 | 9 |

Now if we want to get the element at index 5.

We know that array index will start at 0. So, to get element at ‘X’ index we need to add 0+x. So, to get element at 5th index we just add 0+5. Similarly, we want to get element at index 100 then simply add start\_index (0) +100.

No matter how large/big the index number is. It’s just takes only one addition operation to get the element at given index location.

Res = num\_arr [5].

**Time complexity for searching an element in an array:**

Time complexity for searching an element in an array is O(n).

Let say we have an integer array of size 5.

EX:

int num\_arr [] = new int [5];

Now look at memory allocation for this array in computer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INDEX** | 0 | 1 | 2 | 3 | 4 |
| **TYPE** | Int | Int | Int | Int | Int |
| **VALUE** | 1 | 3 | 5 | 7 | 9 |

Let say we are searching for the element 1 in the array. In this case we find it at 1st position so O (1) but in Big-O we always interested to look at worst case scenario.

Let say the element we are searching for is 9. This is at the end of the array. In this case it is O(n) time complexity.

So, finally searching through an array is O(n).

**Time complexity for inserting an element in an array:**

Time complexity of inserting an element in an array is O (n).

Let say we have an integer array of size 5.

EX:

int num\_arr [] = new int [5];

Now look at memory allocation for this array in computer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INDEX** | 0 | 1 | 2 | 3 | 4 |
| **TYPE** | Int | Int | Int | Int | Int |
| **VALUE** | 1 | 3 | 5 | 7 |  |

Let’s say we want to insert an element into the array at index 2. So, we need to shift all indexes to one position next and make index 2 empty then we insert the value.

In worst case we want to insert new element at index 0 then we need to shift all indexes to one position next. So, the time complexity is O(n).

**Time complexity for deleting an element in an array:**

Time complexity of deleting an element in an array is O (n).

Let say we have an integer array of size 5.

EX:

int num\_arr [] = new int [5];

Now look at memory allocation for this array in computer.

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| **INDEX** | 0 | 1 | 2 | 3 | 4 |
| **TYPE** | Int | Int | Int | Int | Int |
| **VALUE** | 1 | 3 | 5 | 7 |  |

Let say we want to remove the element 5. So, we will shift all the indexes down to one.

num\_arr [2] = num\_arr [3]

num\_arr [3] = num\_arr [4]

num\_arr [4] = null.

In this case we shifted only 3 indexes. Let’s see about worst case scenario deleting element at index 0. Then we need to shift all indexes down to one.

So, the time complexity is O(n).

**Report of Array**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
|  | Accessing | Searching | Inserting | Deleting |
| Time complexity | O (1) | O(n) | O(n) | O(n) |

|  |  |
| --- | --- |
| **Pros** | **Cons** |
| Good for similar contiguous data | Size can’t be changed once initialized. |
| Efficient for accessing elements with constant time complexity. | Not efficient for searching, inserting and deleting. |
| Easy to learn | Can be wasting storage space. If we initialized array with size 10 but we stored only 7 elements then remaining 3 allocated memory locations are waste. |