DATA STRUCTURES

Data may be organized in many different ways. The logical or mathematical model of a particular organization of data is called a **data structure.**

Basic Terminology:

Data: Data are simply values or sets of values.

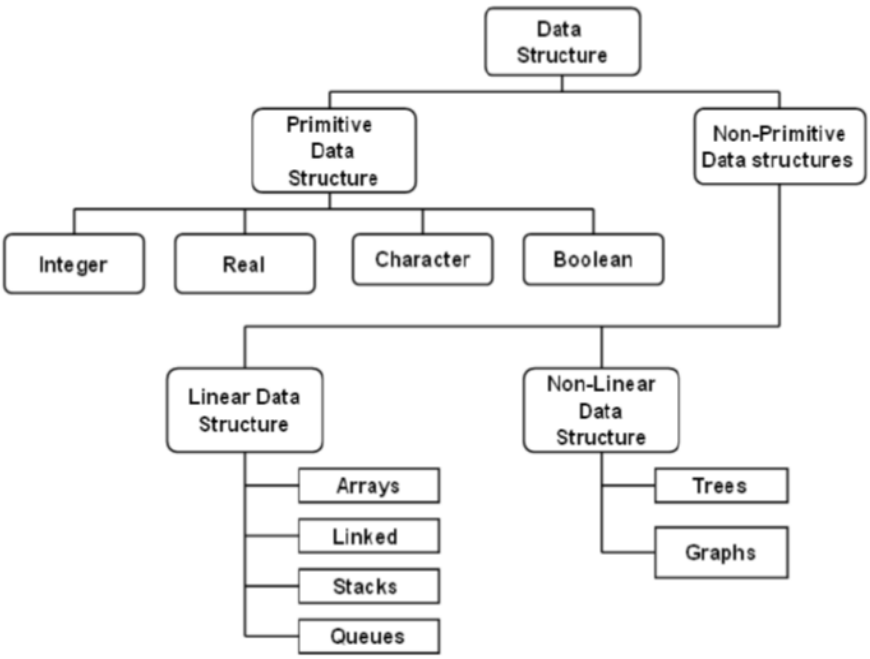
Data items: Data items refers to a single unit of values.

Data items that are not able to divide into sub-items are called Elementary items.

Entity: An entity is something that has certain attributes or properties which may be assigned values. The values may be either numeric or non-numeric.

Ex: Attributes- Names, Age, Sex, SSN

Values- Rohland Gail, 34, F, 134-34-5533



Figures 1.1 Classification of data structures.

- **Primitive data Structures**: Primitive data structures are the fundamental data types which are supported by a programming language. Basic data types such as integer, real, character and Boolean are known as Primitive data Structures .

- **Non- Primitive data Structures:** Non-primitive data structures are those data structures which are created using primitive data structures. Examples of non-primitive data structures is the processing of complex numbers, linked lists, stacks, trees, and graphs.

Based on the structure and arrangement of data, non-primitive data structures is further classified into

1. Linear Data Structure

2. Non-linear Data Structure

Linear data structure –

The arrangement of data in the sequential manner is known as linear data structure. The data structures used for this purpose are arrays , linked list , stacks , queues.

In this data structures , one element is connected to only one another element in a linear form.

Non-linear Data Structure

When one element is connected to the 'n' number of element known as non – linear data structures .

Example – tree and graphs .

ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before

implementation and after implementation. They are the following –

***Priori* Analysis** :- Here , priori analysis is the theoretical analysis of an algorithm which is done before implementing the algorithm.

***Posterior* Analysis :-** Here , posterior analysis is a practical analysis of an algorithm . The practical analysis is achieved by implementing algorithm using any programming language .

Types of Analysis –

The efficiency of some algorithms may vary for inputs of the same size.

Best Case Analysis :- If an algorithm takes the least amount of time to execute a specific set of

input, then it is called the best case time complexity.

Average Case Analysis :- If the time complexity of an algorithm for certain sets of inputs are on an

average, then such a time complexity is called average case time complexity.

Worst Case Analysis :- If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity.

Asymptotic Notations :-

The commonly used asymptotic notations used for calculating the running time complexity of an algorithm is given below .

Big-O Notation (O) :- (upper bond of an algorithm)

'O' is the representation for Big-O notation . The Big-O can also be denoted as f(n) = O(g(n)), where ***f(n)*** and ***g(n)*** are two non -negative functions and f(n) < g(n) if **g(n)** is multiple of some constant **c**.

$$f(n) \leqslant c * g(n)$$

where, n can be any number of inputs or outputs and f(n) as well as g(n) are

two non-negative functions. These functions are true only if there is a constant

c and a non-negative integer n0 such that, $n \geqslant n0$.

Omega Notation :-

'Ω' is the representation for Omega notation. Omega describes the manner in which an

algorithm performs in the best case time complexity .

Omega can also be denoted as f(n) = Ω (g(n)) where, **f** of **n** is equal to Omega of **g** of **n** .

The function **f(n)** is said to be in Ω (g(n)), if **f(n)** is bounded below by some constant multiple of **g(n)** for all large values of **n**, i.e., if there exists some positive constant **c** and some non-negative integer **n0,** such that $f(n) \geqslant c * g(n)$ for all $n \geqslant n0$.

$$f(n) \geqslant c * g(n)$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two non-negative

functions. These functions are true only if there is a constant **c** and a non-negative integer

n0 such that n>n0.

Theta Notation :-

'θ' is the representation for Theta notation. Theta notation is used when the

upper bound and lower bound of an algorithm are in the same order of magnitude.

Theta can also be denoted as f(n) = θ(g(n)) where, f of n is equal to Theta of g

of n . The function f(n) is said to be in θ (g(n)) if f(n) is bounded both above and below

by some positive constant multiples of g(n) for all large values of n, i.e., if

there exists some positive constant c1 and c2 and some non-negative integer

n0, such that C2g(n)⩽f(n)⩽ C1g(n) for all n⩾n0.

$$c1 * g(n) \leqslant f(n) \leqslant c2 * g(n) \text{ for all } n>n0$$

Where, **n** is any number of inputs or outputs and **f(n)** and **g(n)** are two nonnegative

functions. These functions are true only if there are two constants namely, **c1**, **c2,** and a non-negative integer **n0**.

Array

Arrays are defined as collection of similar type of data items stored at contiguous memory location

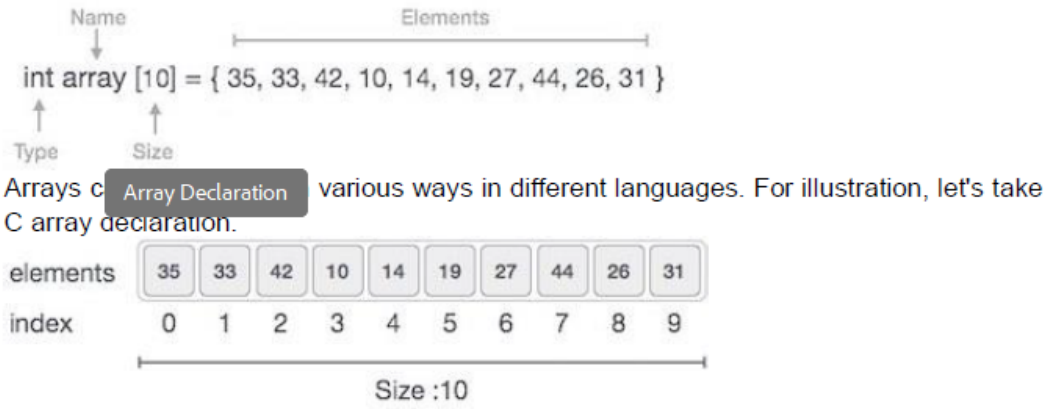
Arrays is the simplest data structure where each data element can be randomly accessed by using its index number .

Element – Each item stored in an array is called an element.

Array Representation:(Storage structure)

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 .
- **Sorting:** Arranging the records in some logical order .
- **Merging:** Combining the records in two different sorted files into a single sorted file.

Traversing operation

This operation is performed to traverse through the array elements. It prints all array elements one after another. We can understand it with the below program -

Example :-

```
1.  #include <stdio.h>
2.  void main() {
3.     int Arr[5] = {18, 30, 15, 70, 12};
4.  int i;
5.     printf("Elements of the array are:\n");
6.     for(i = 0; i<5; i++) {
7.        printf("Arr[%d] = %d,   ", i, Arr[i]);
8.     }
9.  }
```

Output :-

```
Elements of the array are:
Arr[0] = 18,  Arr[1] = 30,  Arr[2] = 15,  Arr[3] = 70,  Arr[4] = 12
```

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.

Example:-

```
#include <stdio.h>
int main()
{
    int arr[20] = { 18, 30, 15, 70, 12 };
    int i, x, pos, n = 5;
    printf("Array elements before insertion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");

    x = 50; // element to be inserted
    pos = 4;
    n++;

    for (i = n-1; i >= pos; i--)
        arr[i] = arr[i - 1];
    arr[pos - 1] = x;
    printf("Array elements after insertion\n");
    for (i = 0; i < n; i++)
        printf("%d ", arr[i]);
    printf("\n");
    return 0;
}
```

Output :-

```
Array elements before insertion
18 30 15 70 12
Array elements after insertion
18 30 15 50 70 12
```

Deletion Operation

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

Example :-

```
#include <stdio.h>

void main() {
   int arr[] = {18, 30, 15, 70, 12};
   int k = 30, n = 5;
   int i, j;

   printf("Given array elements are :\n");

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

   j = k;

   while( j < n) {
      arr[j-1] = arr[j];
      j = j + 1;
   }

   n = n -1;

   printf("\nElements of array after deletion:\n");
    for(i = 0; i<n; i++) {
      printf("arr[%d] = %d,  ", i, arr[i]);
   }
}
```

Output :-

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Elements of array after deletion:
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,
```

Search Operation

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

Example :-

```
1.  #include <stdio.h>
2.
3.  void main() {
4.     int arr[5] = {18, 30, 15, 70, 12};
5.     int item = 70, i, j=0 ;
6.
```

```
7.     printf("Given array elements are :\n");
8.
9.     for(i = 0; i<5; i++) {
10.       printf("arr[%d] = %d,  ", i, arr[i]);
11.    }
12.     printf("\nElement to be searched = %d", item);
13.    while( j < 5){
14.       if( arr[j] == item ) {
15.          break;
16.       }
17.
18.       j = j + 1;
19.    }
20.
21.    printf("\nElement %d is found at %d position", item, j+1);
22. }
```

Output :-

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12,
Element to be searched = 70
Element 70 is found at 4 position.
```

There are two algorithms to perform searching operation , linear search and binary search .

Binary search :-

Binary search is a search algorithm used to find the position of a target value within a **sorted** array. It works by repeatedly dividing the search interval in half until the target value is found or the interval is empty.

Example :-

```
#include<stdio.h>
int main()
{
   int arr[5]={1,2,3,4,5};
   int mid, low=0, high=5, tar;
   printf("enter the target elements:\n");
   scanf("%d",&tar);
   while(low<=high)
   {
      mid=(low+high)/2;
      if(tar== arr[mid]){
         printf("element found at index %d \n", mid);
         break;
      }
      if(tar < arr[mid])
      {
         high=mid-1;
```

```
        }
        else{
            low=mid+1;
        }
    }
}
```

Outputs :-

```
enter the target elements:
2
element found at index 1
```

Linear search :-

Linear search is a method for searching for an element in a collection of elements. In linear search, each element of the collection is visited one by one in a sequential fashion to find the desired element. Linear search is also known as **sequential search**.

Example :-

```
#include<stdio.h>
int main()
{
    int arr[5]={1,2,3,4,5};
    int tar, i, j, flag=0;
    printf("Enter the target element: \n");
    scanf("%d",&tar);

    for(j=0; j<5; j++)
    {
        if(arr[j]==tar)
        {
            flag = 1;
            break;
        }
    }
    if(flag == 1)
    {
        printf("Element found at index %d\n", j);

    }
    else
    printf("target is not found \n");
}
```

Outputs :-

```
Enter the target element: 3
Element found at index 2
```

Update Operation

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

Example :-

```
1.  #include <stdio.h>
2.
3.  void main() {
4.     int arr[5] = {18, 30, 15, 70, 12};
5.     int item = 50, i, pos = 3;
6.
7.     printf("Given array elements are :\n");
8.
9.     for(i = 0; i<5; i++) {
10.       printf("arr[%d] = %d,  ", i, arr[i]);
11.    }
12.
13. arr[pos-1] = item;
14.    printf("\nArray elements after updation :\n");
15.
16.    for(i = 0; i<5; i++) {
17.       printf("arr[%d] = %d,  ", i, arr[i]);
18.    }
19. }
```

Outputs :-

```
Given array elements are :
arr[0] = 18,  arr[1] = 30,  arr[2] = 15,  arr[3] = 70,  arr[4] = 12
Array elements after updation :
arr[0] = 18,  arr[1] = 30,  arr[2] = 50,  arr[3] = 70,  arr[4] = 12
```