***@ Data Structure***

# A data structure is a particular way of organizing data in a computer so that data can be accessed and modified efficiently.

@ Shorting

# Bubble Short

Bubble sort is an internal sorting technique in which **adjacent elements** are compared and exchanged if necessary

The working procedure for **bubble sort** is as follows:

1. Let us consider an array of **n** elements (i.e., **a[n]**) to be sorted.
2. Compare the first two elements in the array i.e., **a[0]** and **a[1]**, if **a[1]** is less than **a[0]** then **interchange** the two values.
3. Next compare **a[1]** and **a[2]**, if **a[2]** is less than **a[1]** then interchange the values.
4. Continue this process till the last two elements are **compared** and **interchanged**.
5. Repeat the above steps for **n - 1** passes.

# Insertion Short

Insertion sort is one that **sorts** a set of elements by inserting an element into the existing sorted elements.

The working procedure for **insertion sort** is as follows:

1. Let us consider an array of **n** elements (i.e., **a[n]**) to be sorted.
2. The **first element a[0]** in the array is itself trivially sorted.
3. The **second element a[1]** is compared with first element **a[0]** and it will be inserted either before or after first element, so that **first** and **second** elements are sorted.
4. The **third element a[2]** is compared with **a[0]** and **a[1]** and it will be inserted into its proper place by checking conditions, so that first three elements are sorted.
5. Repeat the same process for **n - 1** passes.

# Selection Short

Selection sort process can be done in **two** ways, one is the largest element method and the other is smallest element method.

The working procedure for selection sort using the **largest element method** is as follows:

1. Let us consider an array of **n** elements (i.e., **a[n]**) to be sorted.
2. In the first step, the **largest element** in the list is searched. Once the largest element is found, it is exchanged with the element which is placed at the **last position**. This completes the first pass.
3. In the next step, it searches for the **second largest element** in the list and it is interchanged with the element placed at **second last position**. This is done in second pass.
4. This process is repeated for **n - 1** passes to sort all the elements.

The working procedure for **selection sort smallest element method** is as follows:

1. Let us consider an array of **n** elements (i.e., **a[n]**) to be sorted.
2. In the first step, the **smallest element** in the list is searched. Once the smallest element is found, it is exchanged with the element which is placed at the **first position**. This completes the first pass.
3. In the next step, it searches for the **second smallest element** in the list and it is interchanged with the element placed at **second position**. This is done in second pass.
4. This process is repeated for **n - 1** passes to sort all the elements.

@ Searching

# Searching specifies the way to **search** an element from the list of elements.

Linear search (or) Sequential search is to scan each entry in the list in a **sequential** manner until the desired element is found. i.e., it means to find a particular **key element** in a list of elements in a sequential manner.

The working procedure for **linear search** is as follows:

1. Let us consider an array of **n** elements and a **key element** which is going to be search in the list of elements.
2. Compare the **key element** with the first element **a[0]**, if it is **matched** then stop the process and print the **index** of the key element where it is found, otherwise **repeat** the same process with **a[1]**.
3. Compare the **key element** with the second element **a[1]**, if it is **matched** then stop the process and print the **index** of the key element where it is found, otherwise **repeat** the same process with **a[2]**.
4. Continue this process until a match is found (or) until all the elements have been searched.

# Binary search is **faster** than **linear search**, as it uses **divide and conquer** technique and it works on the sorted list either in ascending or descending order.

Binary search (or) Half-interval search (or) Logarithmic search is a search algorithm that finds the position of a **key element** within a sorted array.

Binary search compares the **key element** to the **middle element** of the array; if they are **unequal**, the half in which the **key element** cannot lie is eliminated and the search continues on the remaining half until it is successful.

The working procedure for **binary search** is as follows:

1. Let us consider an array of **n** elements and a **key element** which is going to be search in the list of elements.
2. The main principle of binary search has first divided the list of elements into two halves.
3. Compare the **key element** with the **middle element**.
4. If the comparison result is **true** the print the **index position** where the **key element** has found and stop the process.
5. If the **key element** is greater than the **middle element** then search the key element in the second half.
6. If the **key element** is less than the **middle element** then search the key element in the first half.
7. Repeat the same process for the sub lists depending upon whether **key** is in the **first half** or **second half** of the list until a match is found (or) until all the elements in that half have been searched.

***Array***

# 1D array

An array is a kind of data structure that holds a fixed number of values of a single type, each identified by an array index.

Essentially an array can be thought of as a sequence of buckets. The first bucket is identified with number 0, the second bucket with 1 and so on. This number is called the index.

The length of an array is specified at the time of creating the array. And can be access with **length** field. For example:

* Int[] Arr1 = new int[5]; // defining array
* Arr1[0] = 8; // storing 8 at 0th index
* Arr1.length; // this will return length of array

In Java, we can create an array in the following two ways:

SYNTAX 1 : dataType[] referenceName = new dataType[size];

int[] marksArr = new int[10]; // this creates an empty array of size 10 and initializes all values to 0

or

SYNTAX 2 : dataType[] referenceName = {value1, value2, value3, .....valuen };

int[] marksArr = { 341, 425, 563, 231, 334, 446, 872, 492, 532, 747};

# Multidimensional Array

In Java, a multidimensional array is implemented as an array of arrays.

A multidimensional array is an array whose components are themselves arrays

In Java, the arrays / rows inside a multidimensional array can be of different lengths, like below -

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | ➞ | Array at index 0 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| 47 | 49 | 42 | 45 | 60 | 66 | ➞ | Array at index 1 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 15 | 16 | 17 | 18 | 19 | 20 | 21 | 22 | 23 | 24 | 25 | 26 | **⇾** | Array at index 2 |

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| 32 | 10 | 22 | ➞ | Array at index 3 |

Each array inside of a multidimensional array can be accessed with an index number as shown in the above figure. The first array is identified with number 0, the second array with 1 and so on.

**Note:**The data types of all the arrays inside a multidimensional array should be the same.

In Java, we can create a multidimensional array in the following ways:

SYNTAX 1 : dataType[][] referenceName = new dataType[rowsize][];

int[][] marksArr = new int[3][]; // this creates an empty multidimensional array with 3 arrays.

marksArr[0] = new int[10]; //this create an empty array of size 10 and adds to marksArray at index 0

marksArr[1] = new int[2]; //this create an empty array of size 2 and adds to marksArray at index 1

marksArr[2] = new int[5]; //this create an empty array of size 5 and adds to marksArray at index 2

**or**

SYNTAX 2 : dataType[][] referenceName = { {value1, value2, value3, .....valuen},

 {value1, value2, value3, .....valuep},

....................

 {value1, value2, value3, .....valuer}};

int[][] marksArr = {

{341, 425, 563, 231, 334},

{446, 532, 747},

{872, 492}

};