★ What is the Abstract Data type (ADT) ?

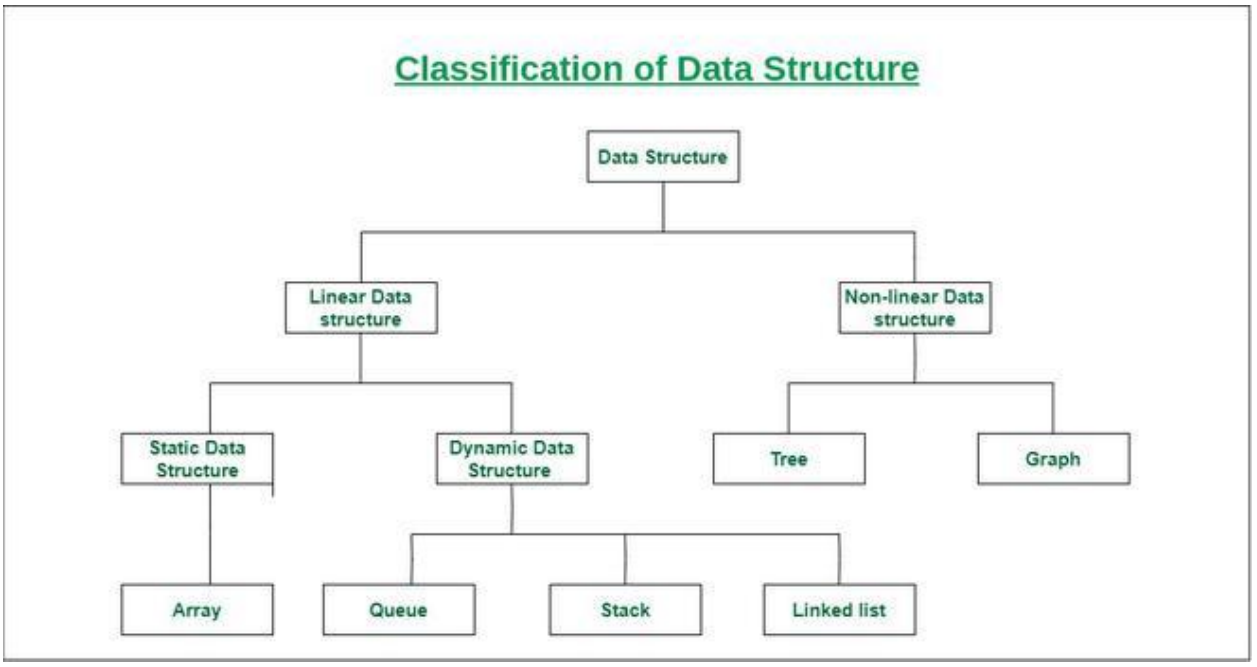
An Abstract Data Type is a collection of data together with a set of data management operations, called Access Procedures, defined on these data.

ADT is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.

Ex - list, stack, queue

★ What is a Data Structure ?

A data structure is a specialized format for organizing, processing, retrieving and storing data.



★ Properties of a Data Structure ?

- Efficient utilization of medium
- Efficient algorithms for creation,manipulation (insertion/deletion) and data retrieval (Find)
- A well-designed data structure allows using little (resources, execution time, memory space)

★ What is an algorithm ?

Algorithm is a finite, clearly specified sequence of instructions to be followed to solve a problem.

An Algorithm is a set of instructions to perform a task or to solve a given problem..

Problem : Write a program to calculate $\sum_{i=1}^{N} i^3$

```
int Sum (int N)

  PartialSum ← 0
  i  ← 1

  foreach (i > 0) and (i<=N)
    PartialSum ← PartialSum + (i*i*i)
     increase i with 1

  return value of PartialSum
```

```
int Sum (int N)
{
    int PartialSum = 0 ;

    for (int i=1; i<=N; i++)
          PartialSum += i * i * i;

    return PartialSum;
}
```

★ **An algorithm possesses the following properties:**

- It must be correct.
- It must be composed of a series of concrete steps.
- There can be no ambiguity as to which step will be performed next.
- It must be composed of a finite number of steps.
- It must terminate.

★ **What is the computer program ?**

A computer program is an instance, or concrete representation, for an algorithm in some programming language.

★ **The Process of Algorithm Development**

- Design
- Validation
- Analysis
- Implementation
- Testing

★ **There are mainly three asymptotic notations**

- Big-O Notation (O-notation) - upper bound / worst case — $0 \leq f(n) \leq c\, g(n)$
- Theta Notation (Θ-notation) - average vase — $C_1(n) \leq f_n \leq C_2\, g(n)$
- Omega Notation (Ω-notation) - lower bound / best vase — $f(n) \geq cg(n)$

★ **Big Oh Notation (O)**

Big-O notation represents the upper bound of the running time of an algorithm. Therefore, it gives the worst-case complexity of an algorithm. (c is a constant value)

$$f(n) = O(g(n))$$

Ex- 01

f (n) = 3n +2
0 ≤ f (n) ≤ c g(n)

0 ≤ 3n+2 ≤ 3n
n=1, 5 ≤ 3
n=2, 8 ≤ 6
n=3, 10 ≤ 9
n=4, 14 ≤ 12

Complexity is = O(n) , *(big O notation = order of c g(n))*

Ex- 02

f (n) = $2n^2$ + 10
0 ≤ f (n) ≤ c g(n)

0 ≤ $2n^2$ + 10 ≤ $2n^2$
n=1, 12 ≤ 2

Complexity is = $O(n^2)$, *(big O notation = order of c g(n))*

Ex- 03

f (n) = $2n^2$ + n
0 ≤ f (n) ≤ c g(n)

0 ≤ $2n^2$ + n ≤ $2n^2$
n=1, 3 ≤ 2
n=2, 10 ≤ 8

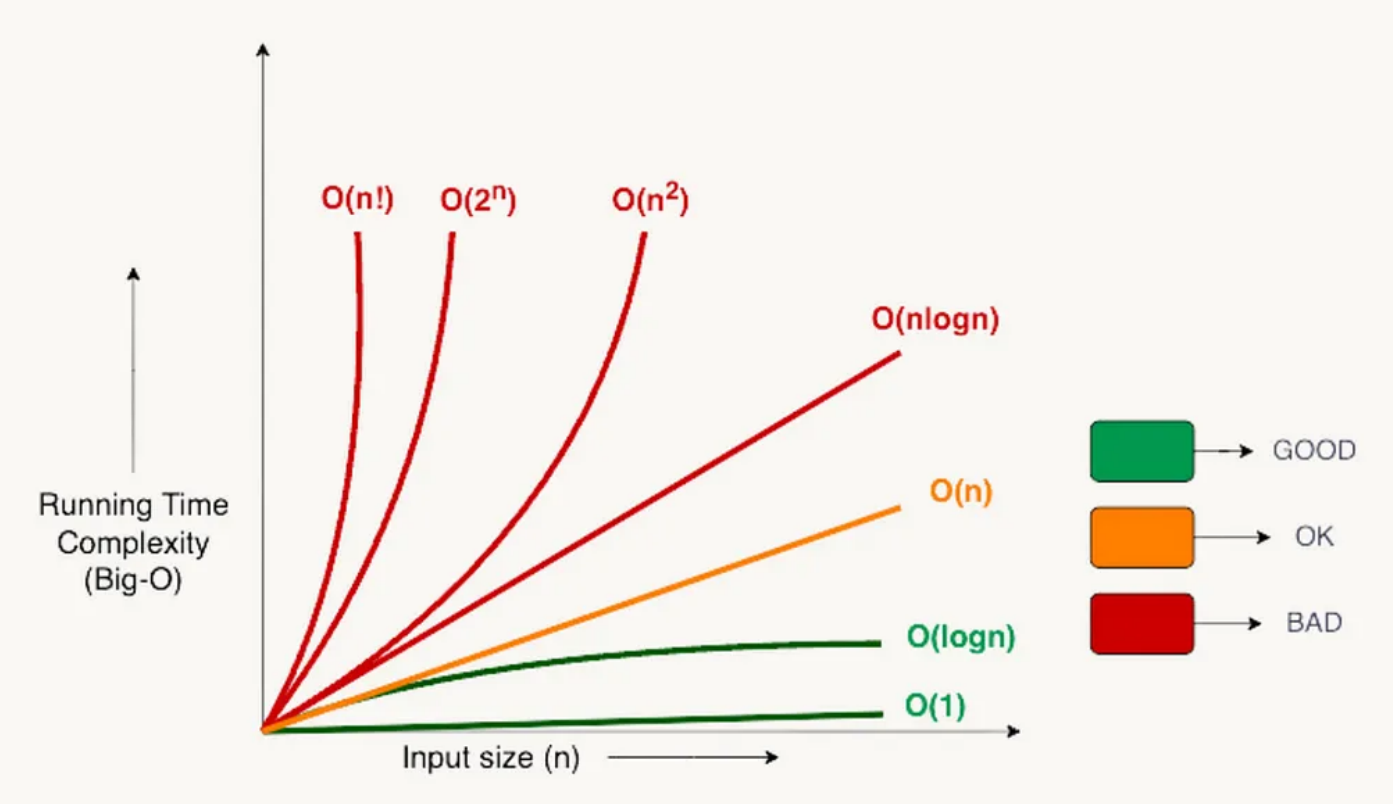
Complexity is = $O(n^2)$, *(big O notation = order of c g(n))*

Some common complexity

- O (2^n) - Exponential
- O (n^2) - quadratic
- O (n log n)
- O (n) - linear
- O (log n) - logarithmic
- O (1) - constant

(more time)

(least time)



★ **What is the Data abstraction ?**

Data abstraction is the process of defining a collection of data and relative operations, by specifying what the operations do on the data.

★ **What is the Abstract Data type (ADT) ?**

An Abstract Data Type is a collection of data together with a set of data management operations, called Access Procedures, defined on these data.

ADT is a type (or class) for objects whose behavior is defined by a set of values and a set of operations.

★ **What is an Array ?**

An array is a collection of items of the same data type stored at contiguous memory locations.

Types of Array

1. **Static Array:** Memory allocated at compile time .It is defined as fixed size of array. It cannot edit or update the size of this array by user. Ex -

```
int Number[5] = {10,12,14,18,100}
char ch[10]={'a','e','i','o','u','A','E','I','O','U'}
```

Size of the array define as numbers values in the Array
Array size cannot be changed once defined.

- **Declaring Arrays**

int myArray[];
declares myArray to be an array of integers

myArray = new int[8];
sets up 8 integer-sized spaces in memory, labeled myArray[0] to myArray[7]

int myArray[] = new int[8];
combines the two statements in one line

- **Assigning Values**

refer to the array elements by index to store values in them.

```
myArray[0] = 3;
myArray[1] = 6;
myArray[2] = 3;   ...
```

can create and initialize in one step:

```
int myArray[] = {3, 6, 3, 1, 6, 3, 4, 1};
```

- **Iterating Through Arrays**

for loops are useful when dealing with arrays:

```
for (int i = 0; i < myArray.length; i++)
{
  myArray[i] = getsomevalue();
}
```

- ***Notes on Arrays***

- index starts at 0.
- arrays can't shrink or grow.
  e.g., use Vector instead.(Vector , It will discuss next session more details)
- each element is initialized.
- array bounds checking (no overflow!)
  ArrayIndexOutOfBoundsException
- Arrays have array.length

- **Example program**

```
public class OneDArray
  {
      public static void main (String args[])
          {
              int month_date[];
              month_date=new int[5];
              // or int month_date[]=new int[5];
                  month_date[0]=31;
                  month_date[1]=28;
```

```
                                        month_date[2]=31;
                                        month_date[3]=30;
                                        month_date[4]=31;

            System.out.println("April has "+ month_date[3]+" Days.");

            }
        }
```

```
public class OneDArray
      {
            public static void main (String args[])
                        {
                                    int month_date[]={31,28,31,30,31}

            System.out.println("April has "+ month_date[3]+" Days.");

            }
       }
```

```
class MultiDimArrayDemo
{
            public static void main(String[] args)
                        {
String[][] names = {{"Mr. ", "Mrs. ", "Ms. "}, {"Smith", "Jones"}};
System.out.println(names[0][0] + names[1][0]);     //Mr. Smith
System.out.println(names[0][2] + names[1][1]);     //Ms. Jones
                         }
}
```

```
public class TwoDArray
   {
            public static void main (String args[])
                        {
                                    int TwoD[][]=new int[3][4];
                                    int i,j,k=0;

                                    for(i=0; i<3; i++) {
                                                for(j=0; j<4; j++)
                                                            {
                                                            TwoD[i][j]=k;
                                                            k=k+1
                                                            }
                                    for(i=0; i<3; i++) {
                                                for(j=0; j<4; j++)
                                                System.out.print(TwoD[i][j]+" ");
```

```
                                        System.out.println();
                                        }
                                }
                        }
```

2. Dynamic Array: memories are allocated at run time. So not having a fixed size. Suppose, user wants to declare any random size of an array.

What is dynamic array

- A dynamic array size assign at running time
- It can be resized automatically.
- Its array expands as it adds more elements. So I do not need to determine the size ahead of time.
- It is a growable array
- It is a resizable array

Advantage of Dynamic Array

- Random Access of Elements
- Good locality of reference and data cache utilization
- Easy to Insert /delete at the time

Disadvantage of Dynamic Array

- Waster more memory
- Shifting elements is time consuming
- Expanding /Shrinking the array is time consuming

★ Dynamic data structure

1. Array List

- ArrayList is a dynamic array .
- it can be used to store the duplicate element of different data types.
- it is maintains insertion order
- it is non-synchronized (multiple threads can work on ArrayList at the same time)
- it elements can be randomly accessed

Ex-

```
import java.util.*;
class TestCollection1{
public static void main(String args[]) {
                //Creating arraylist
                ArrayList<String> l=new ArrayList<String>();
                        l.add("Saman");  //Adding object in arraylist
                        l.add("Kamal");
                        l.add("Saman");
                        l.add("Maharoof");
              //Traversing list through Iterator
                Iterator IT=l.iterator();
                while(IT.hasNext()){       //travers end of the list
                        System.out.println(IT.next());
```

```
}}}
```
//hash Next() and next method use to interator

2. Linked List

- LinkedList uses a doubly linked list to store elements internally.
- LinkedList can store duplicate elements.
- It also maintain insert order
- it is not synchronized
- LinkedList manipulation is fast because that no shifting is required

3. Vector

- It is a dynamic array used to data store elements.
- It is same as ArrayList
- Vector is synchronized (which means only one thread at a time can access the code)

Ex-

```
import java.util.*;
public class TestCollection3 {
public static void main(String args[]){
        Vector<String> ve=new Vector<String>();
        ve.add("Saman");  //Adding object in Vector
        ve.add("Kamal");
        ve.add("Saman");
        ve.add("Maharuf");
    //Traversing list through Iterator
        Iterator IT=ve.iterator();
        while(IT.hasNext()) {     //travers end of the Vector
                System.out.println(IT.next());
}}}
```

★ Array and List

Basic Operations in the Arrays

- 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.
- Display − Displays the contents of the array.

1. Traverse

Algorithm

1 Start
2. Initialize an Array of certain size and datatype.
3. Initialize another variable ‘i’ with 0.
4. Print the i th value in the array and increment i.
5. Repeat Step 4 until the end of the array is reached.
6. End

Ex -

```
public class ArrayTraversal {
   public static void main(String[] args) {
      Int [ ] X = { 2, 4, 6, 8, 10 };

      // Printing elements of the array using a for loop
      System.out.println("Using for loop:");
      for (int i = 0; i < X.length; i++) {
         System.out.println("X [" + i + "] = " + X [ i ] );
      }

      // Printing elements of the array using an enhanced for loop (for-each loop)
      System.out.println("\nUsing enhanced for loop:");
      for (int num : X) {
         System.out.println("Element: " + num);
      }
   }
}
```

```
Using for loop:
X[0] = 2
X[1] = 4
X[2] = 6
X[3] = 8
X[4] = 10

Using enhanced for loop:
Element: 2
Element: 4
Element: 6
Element: 8
Element: 10
```

2. Insertion Operation

Algorithm

1. Start
2. Create an Array of a desired data type and size.
3. Initialize a variable ‘i’ as 0.
4. Enter the element at it n index of the array.
5. Increment i by 1.
6. Repeat Steps 4 & 5 until the end of the array.
7. Stop

Ex -

```
public class ArrayInsert {

    public static void main(String[] args) {
        int X[] = new int[3];
        System.out.println("Array Before Insertion:");
        for (int i = 0; i < 3; i++) {
            System.out.println("X[" + i + "] = " + X[i]); // prints empty array
        }
        System.out.println("Inserting Elements..\n");

        // Defining elements to insert
        int newArray[] = { 2, 4, 7 };

        // Printing Array after Insertion
        System.out.println("Array After Insertion:");
        for (int i = 0; i < newArray.length; i++) {
            X[i] = newArray[i];
            System.out.println("X[" + i + "] = " + X[i]);
        }
    }
}
```

```
Array Before Insertion:
X[0] = 0
X[1] = 0
X[2] = 0
Inserting Elements..

Array After Insertion:
X[0] = 2
X[1] = 4
X[2] = 7
```

3. Deletion

Algorithm

1. Start
2. Set J = K
3. Repeat steps 4 and 5 while J < N
4. Set X[J] = X[J + 1]
5. Set J = J+1
6. Set N = N-1
7. Stop

Ex-

```
public class delete {

    public static void main(String[] args) {
        int X [ ] = new int [3];

        int myArray[] = { 2, 4, 7 };

        int indexToDelete = 1;

        // Shifting elements to the left starting from the index to be deleted
        for (int i = indexToDelete; i < myArray.length - 1; i++) {
            myArray[i] = myArray [ i + 1];
        }
```

```
myArray[0] = 2
myArray[1] = 7
```

```
        // Displaying the updated array
        for (int i = 0; i < myArray.length-1; i++) {
            System.out.println("myArray [" + i + "] = " + myArray[ i ]);

        }
    }
}
```

Multidimensional Arrays

	Column 0	Column 1	Column 2
Row 0	x[0][0]	x[0][1]	x[0][2]
Row 1	x[1][0]	x[1][1]	x[1][2]
Row 2	x[2][0]	x[2][1]	x[2][2]

Print 2D array

```
public class Mavenproject1 {

    public static void main(String[] args)
    {
        Int [ ] [ ] arr = { { 1, 2 }, { 3, 4 } };

        for (int i = 0; i < 2; i++) {
            for (int j = 0; j < 2; j++) {
                System.out.print(arr [ i ] [ j ] + " ");
            }

            System.out.println();
        }
    }
}
```

★ **Advantages of ArrayList**

- Can store and delete data randomly
- Various types of Objects can be entered.
- Since the size is resized during the running time, it is easy during implementation requirement changes.
- Because many predefined methods can be supported, it is very easy to store object during manipulation.

★ **Disadvantages of ArrayList**

- If a data entry is added or removed from an array-based list, the data must be transferred to update the list.
- In the worst case, for an array-based list with n Data insertions, additions, and deletions are O(n) Time.

- Also, all data in an array-based list must be stored in memory in order. Large lists require significant contiguous chunks of memory.

★ Limitation of arrays

- Need to know the size at the start (at least some good estimate)
- Can not go beyond that

★Advantages of arrays

- Contiguous in memory (good cache locality if accessed sequentially)
- Very limited book keeping is required (only need to know the start of the array and its size)

★ Linked List Data Structure

Linked List is a linear data structure, in which elements are not stored at a contiguous location, rather they are linked using pointers. Linked List forms a series of connected nodes, where each node stores the data and the address of the next node.

Alternative - Linked list used when we do not know the size in advance and size varies a lot

What is Pointer?

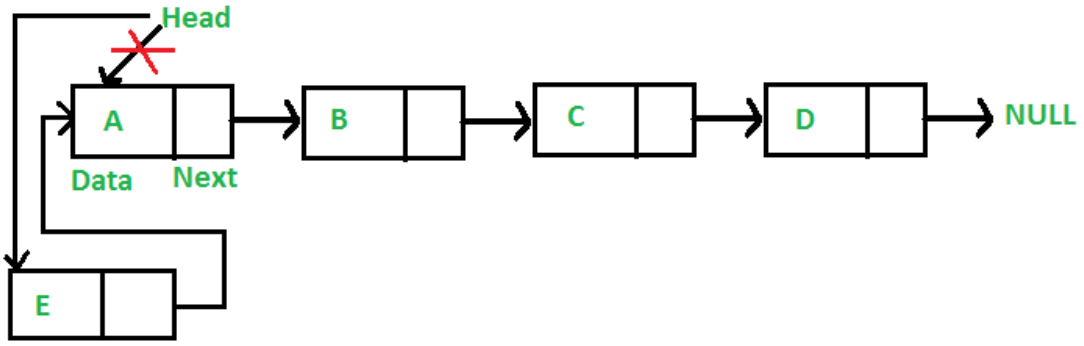
Pointer contains memory address of a particular type of data.

Common operations of Linked List are:

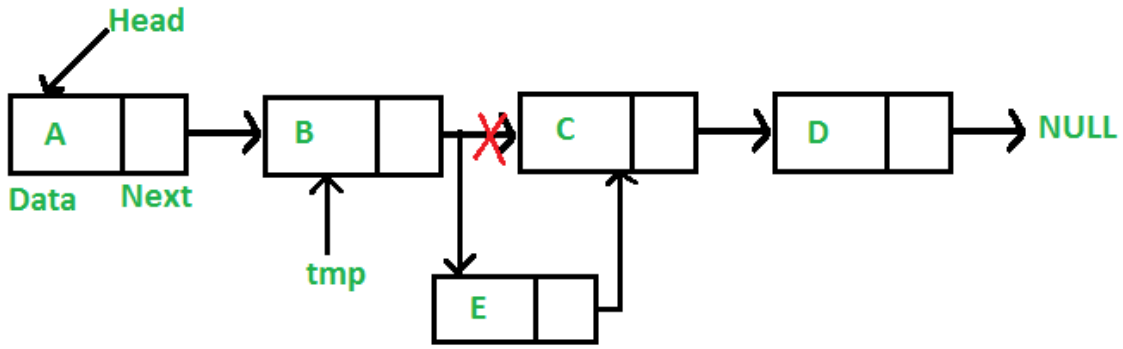
- initializeList() - initializes the list as an empty list.
- insertFirstElt(int elt) - inserts a new element into an empty list.
- insertAtFront(int elt) - inserts an element at the beginning of the list.
- insertAtEnd(int elt) - inserts an element at the end of the list (appendElt or appendNode).
- insertAfter(int oldElt, int newElt) - inserts an element after a specified element.
- deleteElt(int elt) - deletes a specified element.
- displayList() - displays all the elements in the list
- isEmpty() - returns true if the list has no elements, false otherwise.
- isFull() - returns false if the list is full, false otherwise.

How to Insert a Node at the Front/Beginning of Linked List

1. Make the first node of Linked List linked to the new node
2. Remove the head from the original first node of Linked List
3. Make the new node as the Head of the Linked List.

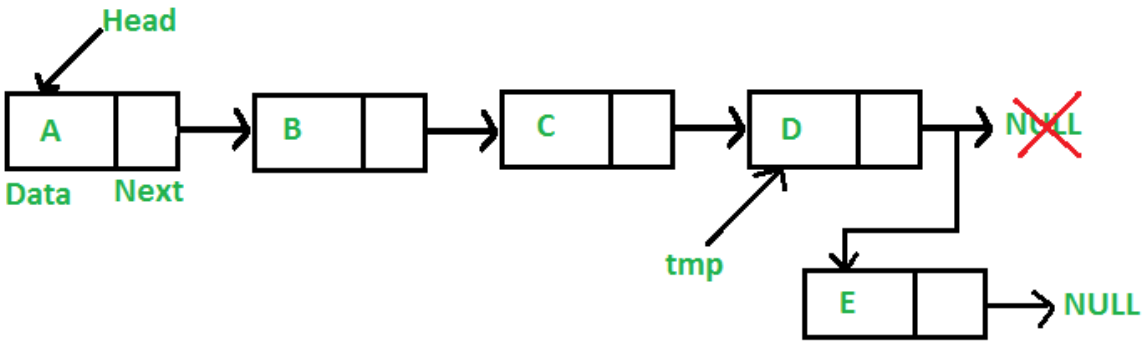


How to Insert a Node after a Given Node in Linked List

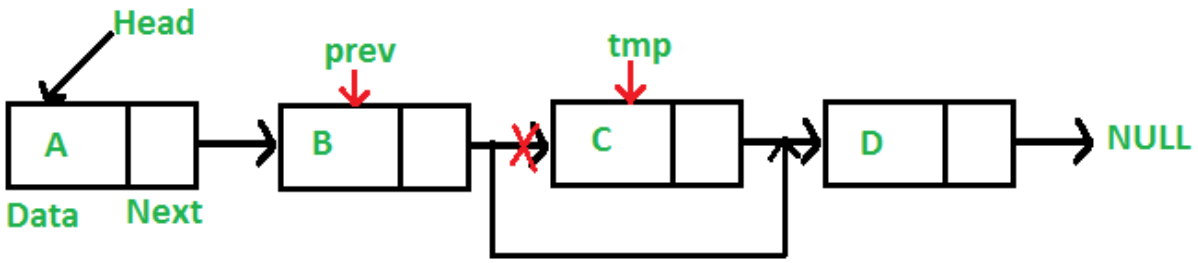


How to Insert a Node at the End of Linked List

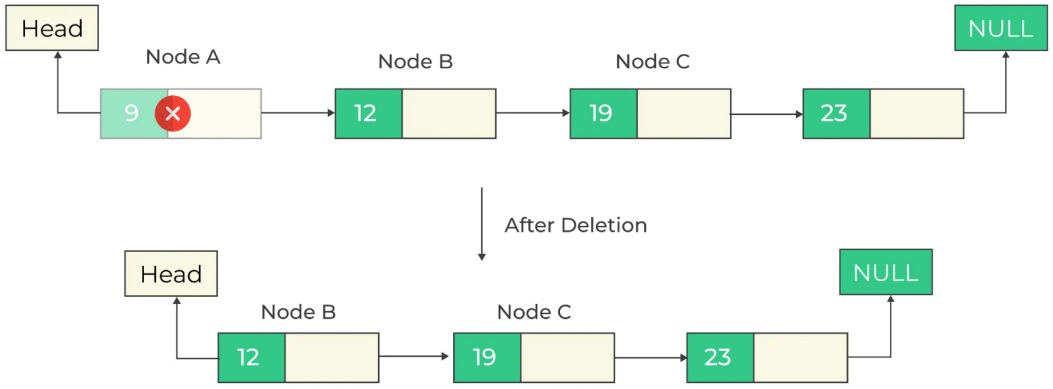
1. Go to the last node of the Linked List
2. Change the next pointer of last node from NULL to the new node
3. Make the next pointer of new node as NULL to show the end of Linked List



How to deletion Node at the middle

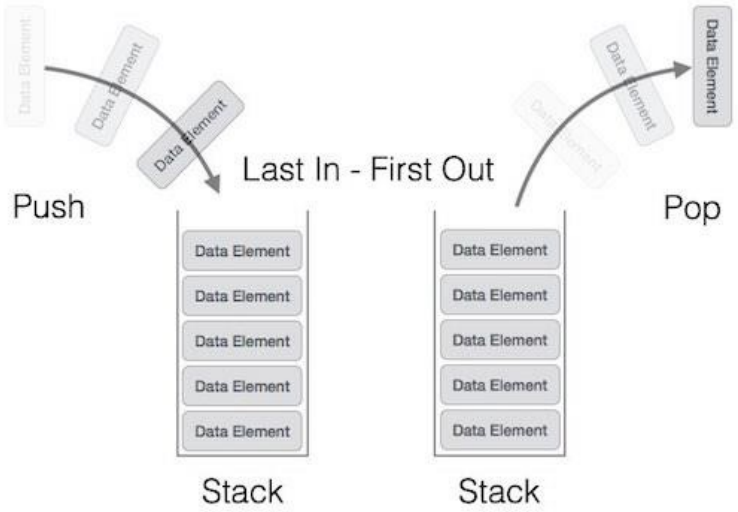


How to deletion Node at the beginning



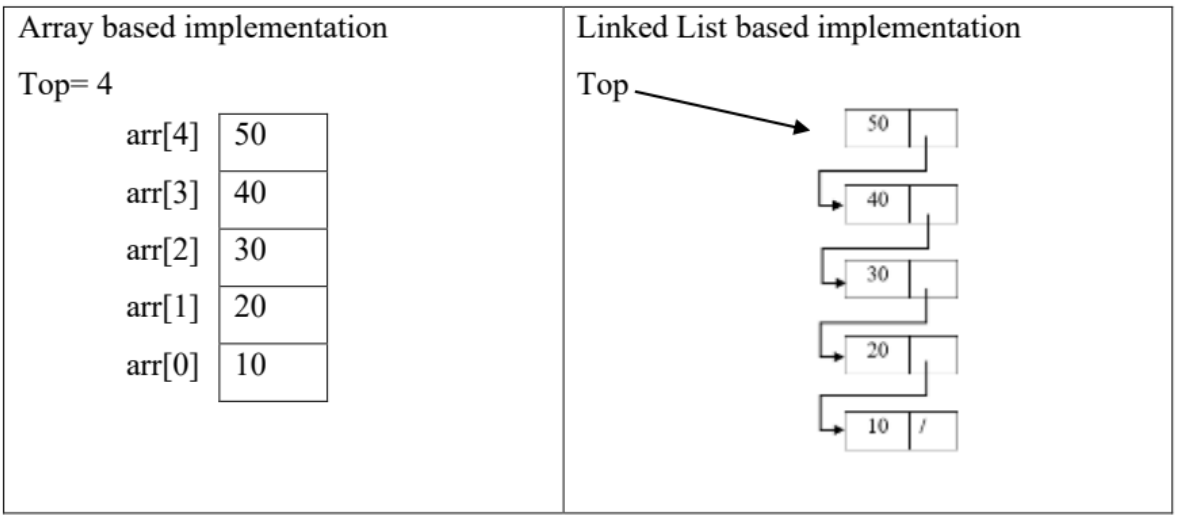
★ Stacks Data Structure - LIFO (Last In First Out)

A stack is a linear data structure which can be accessed only at one of its ends for storing and retrieving data.The order may be LIFO (Last In First Out)



Two different implementations of Stack

1. Array based implementation (Static implementation)
2. Linked List based implementation (dynamic implementation)



This short note making resources are,google search engine,chat gpt and lecture slides

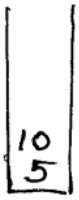
WhatsApp group (click on this/new syllabus notes only) Credit - Kanishka viraj

Graphically illustrate the following stack operations sequentially.

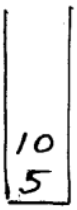
i. initializeStack()

ii. push(5)

iii. push(10)

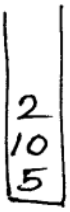


iv. a = topEement( )

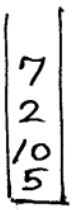


a = 10

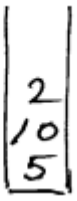
v. push(2)



vi. push(7)



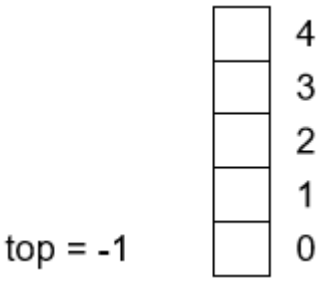
vii. b = pop()



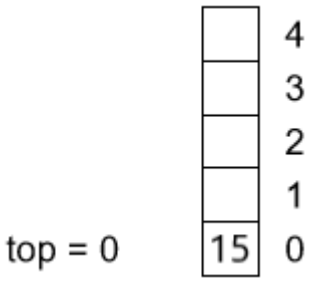
b = 7

Show the graphical **implementation** of following Stack operations

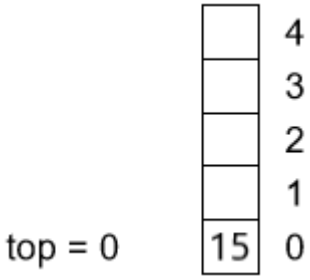
i. initializeStack()



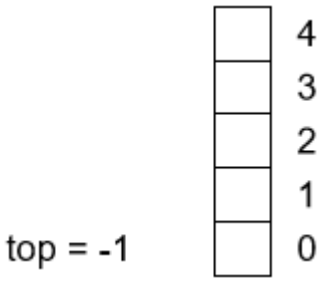
ii. push(15)



iii. y = topElement( )
y=15



iv. x= pop()
x=15



Array implementation

Public operations

- void push(x) – Insert x
- void pop() – Remove most recently inserted item
- boolean isEmpty() – Return true if empty, false otherwise
- void display() – Display all items
- Boolean isFull()-Return true is Stack is Full, false otherwise

★ Queue Data Structure

A Queue is defined as a linear data structure that is open at both ends and the operations are performed in First In First Out (FIFO) order.

Common operations of Queue are:

- initializeQueue() – initializes the queue as an empty queue.
- enQueue() - adds an element at the rear of the queue.
- deQueue() - removes and returns the front element from the queue.
- frontElt() - returns the front element without removing it.
- isEmpty() - returns true if the queue has no elements and false otherwise.

- isFull() - returns true if the queue is full of elements and false otherwise.
- displayQueue() - displays all elements from front to rear.

Graphically illustrate the following Queue Operations

1. initializeQueue()

2. p=isEmpty()

   p = true

3. enQueue(5)

   5

4. enQueue(9)
   enQueue(7)

   5 9 7

5. x=deQueue()
   x=5

   9 7

6. enQueue(2)
   enQueue(6)

   9 7 2 6

7. q = isFull()
   q = false

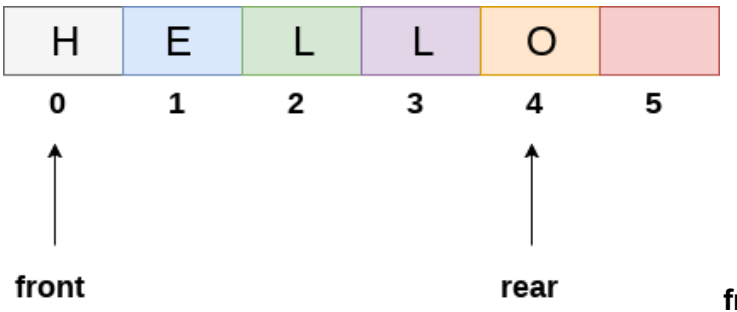
   9 7 2 6

8. enQueue(3)

   9 7 2 6 3

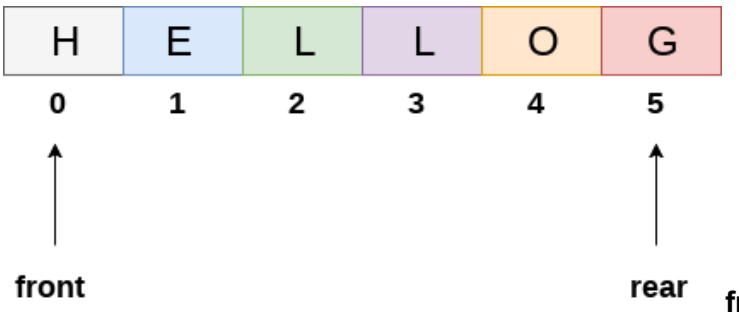
Graphically illustrate the **implementation** of the following queue operations.

For an empty queue, Front = - 1 , rear = - 1 and size = 0



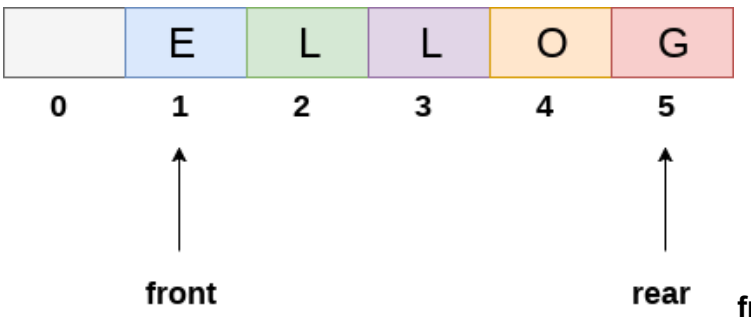
front = - 1, rear = 4 , size = 5

enQueue(G)



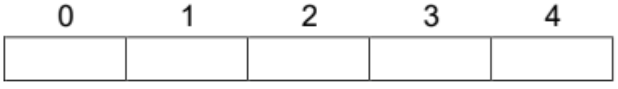
front = - 1, rear = 5 , size = 6

deQueue(H)



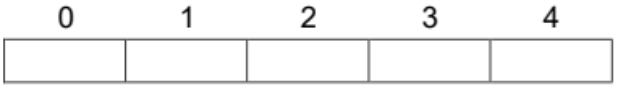
front = 0, rear = 5 ,size = 5

1. initializeQueue()



front	-1
rear	-1
size	0

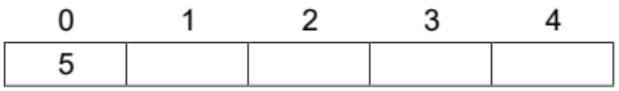
2. p=isEmpty()



front -1
rear -1
size 0

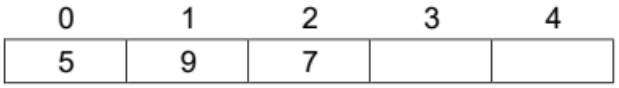
p = true

3. enQueue(5)



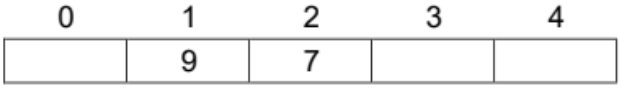
front -1
rear 0
size 1

4. enQueue(9)
enQueue(7)



front -1
rear 2
size 3

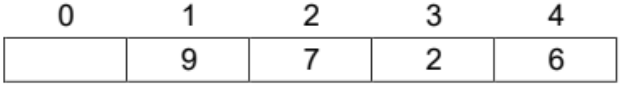
5. x=deQueue()



front 0
rear 2
size 2

x = 5

6. enQueue(2)
enQueue(6)



front 0
rear 4
size 4

		0	1	2	3	4
7. q = isFull()			9	7	2	6
	front	0				
	rear	4				
q = false	size	4				

		0	1	2	3	4
8. enQueue(3)		3	9	7	2	6
	front	0				
	rear	0				
	size	5				

		0	1	2	3	4
9. r = isFull() y = deQueue()		3		7	2	6
	front	1				
r = true	rear	0				
y = 9	size	4				

circular array implementation

All the positions before the front are unused and can thus be recycled. When either rear or front reaches the end of the array, we reset it to the beginning. This operation is called a circular array implementation

		C	D	E	F	G	H	I

Size=7,front=1,rear=8

J		C	D	E	F	G	H	I

Size=8,front=1,rear=0

★ Tree Data Structure

Tree is one of the most important non-linear data structures in computing. It allows us to implement faster algorithms.

A Tree is a set of nodes storing elements in a parent-child relationship with the following properties: It has a special node called root.

Tree types

- **Binary tree — each node has at most two children**

  A binary tree is a finite set of nodes that is either empty or consists of a root and two disjoint binary trees called the left subtree and the right subtree.

1. full Binary tree

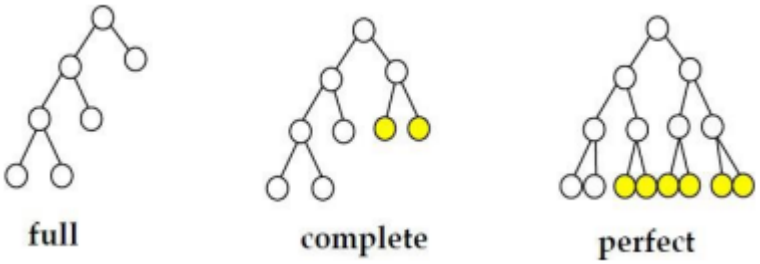
   A full Binary tree is a special type of binary tree in which every parent node/internal node has either two or no children.
   It is also known as a proper binary tree.

2. Complete Binary Tree

   A complete binary tree is a binary tree in which all the levels are completely filled except possibly the lowest one, which is filled from the left.

3. perfect binary tree

   A perfect binary tree is a special type of binary tree in which all the leaf nodes are at the same depth, and all non-leaf nodes have two children. In simple terms, this means that all leaf nodes are at the maximum depth of the tree, and the tree is completely filled with no gaps.



- **General tree — each node can have an arbitrary number of children.**
- **Binary search trees**
- **AVL trees**

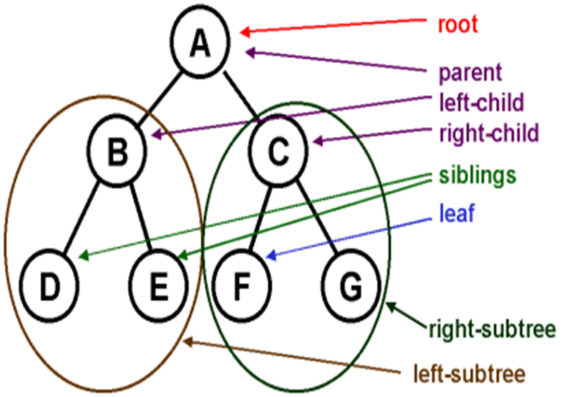
  An AVL tree is defined as a self-balancing BST where the difference between heights of left and right sub trees for any node cannot be more than one.

  The difference between the heights of the left sub tree and the right sub tree for any node is known as the balance factor of the node.

Basic Terminologies In Tree Data Structure:

I. Root – the unique node which do not have a parent
II. Leaf – a node which has no children
III. Size of a tree – number of nodes that a tree has
IV. Depth of a node – number of edges from the root to the given node
V. Degree of a node – number of children that owned by the node

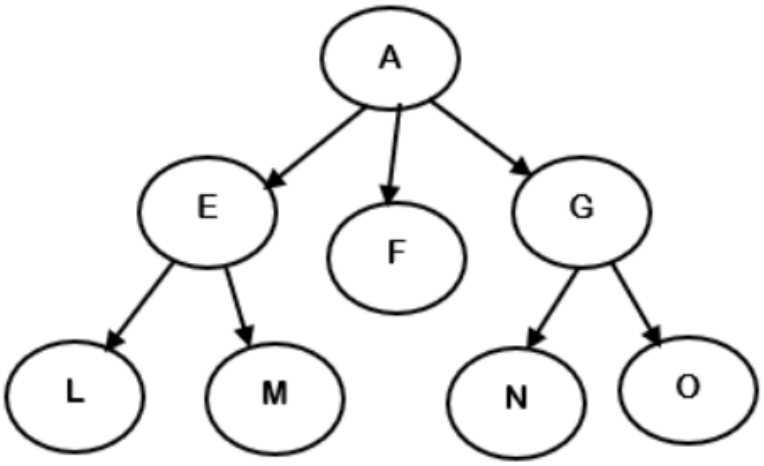
VI. Degree of a tree – the maximum degree of any of nodes within the tree

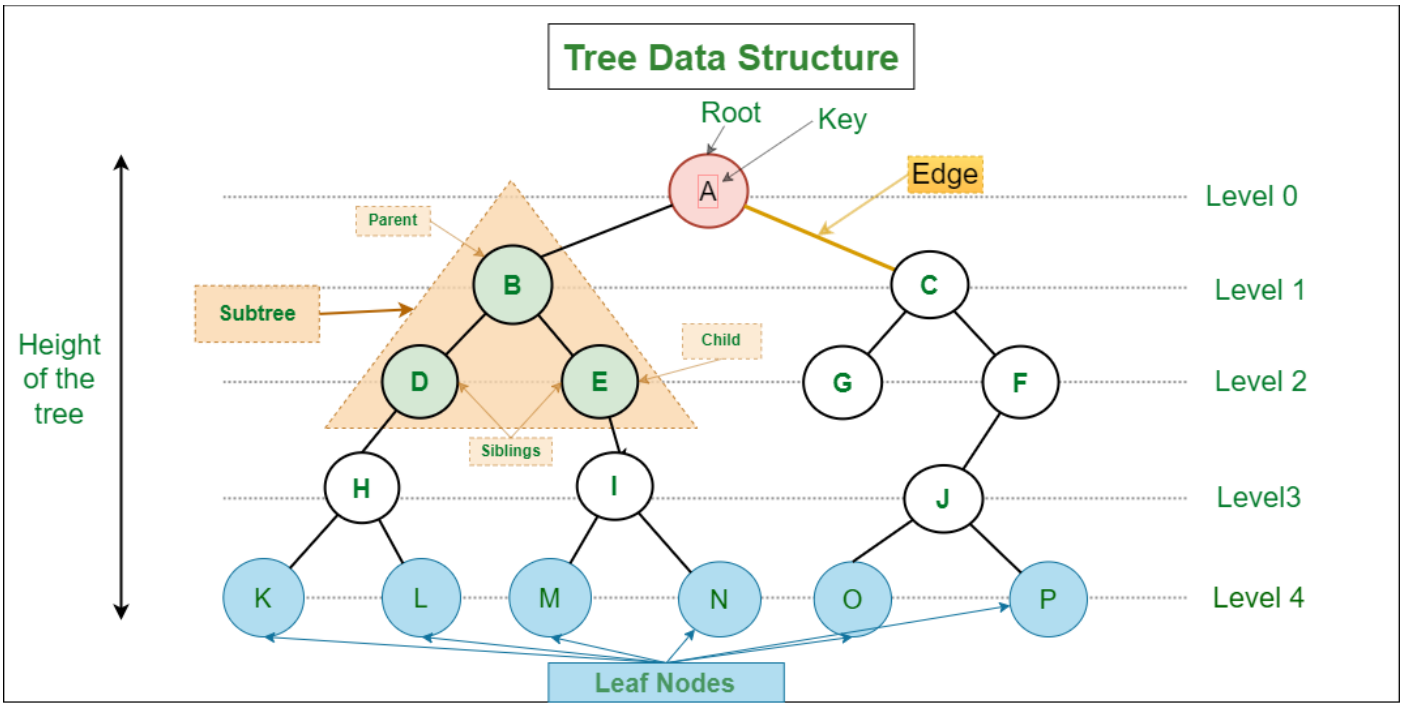


Draw a Tree diagram using the following list of information.

Root node: A
Leaf nodes: L, M, N, O

Level one nodes: E, F, G
Children's of node E: L, M
Parent of N and O: G





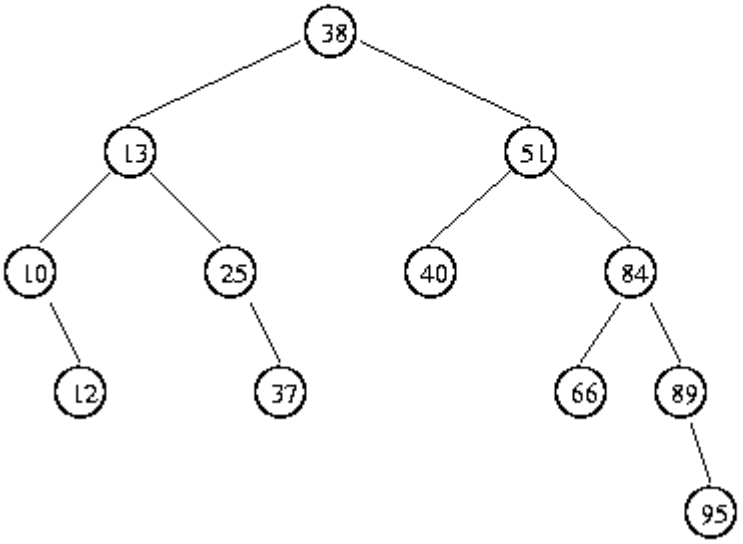
Binary Search Tree

Binary Search Tree is a node-based binary tree data structure.

The left subtree of a node contains only nodes with keys lesser than the node's key.
The right subtree of a node contains only nodes with keys greater than the node's key.

Note: if duplicate keys are allowed, then nodes with values that are equal to the key in node N can be either in N's left sub tree or in its right sub tree (but not both). In these notes, we will assume that duplicates are not allowed.

Tree resulting from the following insertions: 38, 13, 51, 10, 12, 40, 84, 25, 89, 37, 66, 95



Differentiate the Binary Tree and Binary Search

Binary Tree	Binary Search Tree
Binary Tree is a specialized form of tree which represents hierarchical data in a tree structure.	Binary Search Tree is a type of binary tree which keeps the keys in a sorted order for fast lookup.
Each node must have at the most two child nodes with each node being connected from exactly one other node by a directed edge.	The value of the nodes in the left subtree are less than or equal to the value of the root node, and the nodes to the right subtree have values greater than or equal to the value of the root node.
There is no relative order to how the nodes should be organized.	It follows a definitive order to how the nodes should be organized in a tree.
It's basically a hierarchical data structure that is a collection of elements called nodes.	It's a variant of the binary tree in which the nodes are arranged in a relative order.
It is used for fast and efficient lookup of data and information in a tree structure.	It is mainly used for insertion, deletion, and searching of elements.

★ Sorting Algorithm

What is sorting?

Arranging items in ascending or descending order is called sorting.

Selection Sort Algorithm:

Here we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array.

Example: Sort the numbers 6, 7,72, 4, 32, 65, 9, 56 using selection sort.

	0	1	2	3	4	5	6	7	
Pass0	6	7	72	4	32	65	9	56	Original
Pass1	4	7	72	6	32	65	9	56	
Pass2	4	6	72	7	32	65	9	56	
Pass3	4	6	7	72	32	65	9	56	
Pass4	4	6	7	9	32	65	72	56	
Pass5	4	6	7	9	32	65	72	56	
Pass6	4	6	7	9	32	56	72	65	
Pass7	4	6	7	9	32	56	65	72	Sorted

Bubble Sort Algorithm:

Here we repeatedly move the largest element to the highest index position of the array.

Example: Sort the numbers 6, 7,72, 4, 32, 65, 9, 56 using bubble sort.