Data Structure. and its Classification

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

Classification

- Primitive v/s Non Primitive Data Structure
- Static v/s Dynamic Data Structure
- Homogenous v/s Heterogeneous Data Structure
- Linear v/s Non Linear Data Structure

Examples

Primitive DS:
Refers to fundamental data structure.
integer,Float, char, Boolean

Non Primitive DS
Refers to non basic data structure. They are non atomic in nature
Structure, array, union, stack, queue, Linked list, tree etc.

Static DS
Data structures whose memory allocation is fixed at compile time.
Arrays, Structure

Dynamic DS
Data structures whose memory allocation is done at run time. These data structures size may be reduced or extended at run time
Linked List, Trees, Graphs

Linear DS
In linear DS, each element has unique successor and unique predecessor.
Arrays, Linked List

Non Linear DS
In non linear DS, element may or may not have unique successor element. Memory allocation is also non contagious.
Trees, Graphs

List of common operations on DS

Insertion, Deletion, Traversing, Sorting, Merging, copying etc.

Address Calculation

Address Calculation:

One Dimensional Array: A of 10 integers

A[1]	A[2]	A[3]	A[4]	A[5]	A[6]	A[7]	A[8]	A9]	A[10]
2000	2002	2004	2006	2008	2010	2012	2014	2016	2018

Total no of elements: (Highest index – lowest index + 1)

Base address: 2000
Find out the address of A[10]
Address of(A[K]) = Base Address of A + width(K- lowest bound of given array)

Two Dimensional Array
Given array A

Row major Array
Col Major Array

A_{11}	**A_{12}**	A_{13}	A_{14}
A_{21}	A_{22}	A_{23}	A_{24}
A_{31}	A_{32}	A_{33}	A_{34}

Base Address = 2000
Memory Representation of Row Major array

A_{11}	**A_{12}**	A_{13}	A_{14}	A_{21}	A_{22}	A_{23}	A_{24}	A_{31}	A_{32}	A_{33}	A_{34}
2000	**2002**	2004	2006	2008	2010	2012	2014	2016	2018	2020	2022

Address of A[j][k] = Base A + width * [n *(j-lower bound of rows) + (k- lower bound of columns)]
Where A is of m x n rows and cols respectively.
A[2][2] = 2000 + 2(4*(2-1) +(2-1)) = 2000+10 = 2010
A[[3][2] = 2000 +2 (4*(3-1) + (2-1)) = 2000 + 18 = 2018

Memory Representation of Column Major array

A_{11}	**A_{21}**	A_{31}	A_{12}	A_{22}	A_{32}	A_{13}	A_{23}	A_{33}	A_{14}	A_{24}	A_{34}
2000	**2002**	2004	2006	2008	2010	2012	2014	2016	2018	2020	2022

Address of A[j][k] = Base A + width * [m *(k-lower bound of cols) + (j- lower bound of row)]
Where A is of m x n rows and cols respectively.
A[2][2] = 2000 + 2(3*(2-1) +(2-1)) = 2000+10 = 2008
A[[3][2] = 2000 +2 (3*(2-1) + (3-1)) = 2000 + 18 = 2010

Exercise 1
A 2D array defines as a [4…7,-1…3] requires 2 bytes of storage space for each element. If the array is stored in row major form, then calculate the address of element at location [6, 2]. Give that base address is 100.
Answers: 126

Exercise 2

Each element of an array A [-20...20, 10..35] requires one byte of storage. If array is major col and begining of array is at location 500(base address). Determine the address of A[0][30]
Answer =

Exercise 3

Suppose multidimensional array A declared using A(-2:2, 2:22). Find the length of each dimension.

Exercise 4

Let A a 2 D array declared as follows;
A: array[1…100][1…15] of integer;
Assuming that each integer takes one memory location in row major order and first element of the array is stored at location 100, what is the address of the element A[i][j]?

Insertion in 1 D Array

```
/*This program is made for insert a element in a array*/
#include<stdio.h>

int main()
{

int a[100],i,n,j,b;

printf("enter the size of array:- ");
scanf("%d",&b);
printf("\nenter the elements in array:- ");

for(i=0;i<b;i++)
   {
   scanf("\n%d",&a[i]);
   }

printf("\nat what position,you want to insert a number ?:-  ");
scanf("%d",&n);

printf("\nwhat value want to insert:-  ");
scanf("%d",&j);

for (i=b-1;i>=n-1;i--)
  {
  a[i+1]=a[i];
  }
a[n-1]=j;

printf("\n value after insertion");

for (i=0;i<b+1;i++)
  {
  printf("\n%d",a[i]);
  }
}
```

Deletion in array

```
/*This program is made for delete a number in a array*/

#include<stdio.h>
int main()
{
int a[100],i,j,n;
printf("\nenter the size of array:- ");
scanf("%d",&n);
printf("\nenter the numbers in array:- ");
for(i=0;i<n;i++)
  {
  scanf("\n%d",&a[i]);
  }
printf("\nenter a position of that number which u want to delete:- ");
scanf("%d",&j);
for(i=j-1;i<n;i++)
  {
  a[i]=a[i+1];
  }

for(i=0;i<n-1;i++)
  {
  printf("\n%d",a[i]);
  }
}
```

<u>Linear Search</u>

Suppose **K** is a linear array (One D array) with **n** elements.

We have to **find out** the given element i.e **Item** in **array K**.

In this we compare Item with each element of array **K** one by one.

i.e we test whether K[0] == Item and then we test whether K[1] == Item , and so on.

This method which traverses array **K** sequentially to locate **Item**, is called Linear search or sequential search.

Function Linear_Search(K, N, X) Given an unordered array K consisting of (N+1) elements. This algorithm searches the array for a particular element having the value X. Array K[N+1] serves as sentinel element and receives the value of X prior to search. The function returns the index of array element if the search is successful, and returns -1 otherwise.	

1	[initialize search]
	I ← 1
	K[N] ← X

2	[Search the array] Repeat while K[I] <> X I ← I+1

3	If (I == N)
	Then Write (“Unsuccessful Search”)
	        Return (-1)
	Else
	        Write (“Successful Search”)
	        Return (I)

/* Linear Search Program */

```
# include <stdio.h>
main()
{
int item, a[100], i, loc, n;
clrscr();
printf(“ How many elements U want in array: ”);
scanf(“%d”, &n);
printf(“\nEnter elements in array:\n”);

for(i=0; i<n; i++)
  {
    printf(“\n Enter %dth element = ”, i);
    scanf(“%d”, &a[i]);
  }

printf(“\n input Item to be found in array = ”);
scanf(“%d”, &item);

a[n] = item;

i = 0; loc = 0;
while (a[i] != item)
  {
    loc = loc +1;
    i = i + 1;
  }
if (loc == n)
   {
     printf(“\nSearch Unsuccessful”);
   }
else
   printf(“\nItem found at location : %d”, loc);
```

}

10	15	45	34	12	50

Binary Search

Necessary and Sufficient condition for searching is "Array must be sorted/arranged in meaningful order.(i.e ascending/descending/Lexical Order)

Function BINARY_SEARCH(K, N, X) Given an Ordered array K consisting of (N) elements. This algorithm searches the array for a particular element having the value X. The variables LOW, MIDDLE, and HIGH denote the lower, middle and upper limits of the search interval respectively. The function returns the index of array element if the search is successful, and returns -1 otherwise.

```
1   [Initialize]
    LOW ← 0
    HIGH ← N-1

2   [Perform Search]
      Repeat thru step 4 while LOW <= HIGH

3   [Obtain index of midpoint of Interval]
      MIDDLE ← (LOW + HIGH) /2

4   [Compare]
      If ( X < K[MIDDLE])
       Then
             HIGH ← MIDDLE – 1
       ELSE
           IF ( X > K[MIDDLE])
               Then  LOW ← MIDDLE + 1
                ELSE   Write "Successful search"
                     Return (MIDDLE)

5   [Unsuccessful search]
    Write "Unsuccessful Search"
    Return (-1)
```

```
/* Binary search */
# include<stdio.h>
# include<conio.h>
main()
 {
  int a[100], i, step,n, temp, item, beg, end, mid;
  clrscr();
  printf("\nHow many elements U want in array : ");
  scanf("%d",&n);
  printf("\n Input elements in array: \n");

  for(i = 0; i<n; i++)
    {
     scanf("%d", &a[i]);
    }
  printf("\nEnter the Item to be found : ");
  scanf("%d", &item);

  for( step = 1; step <= n-1; step++)
     {
       for (i = 1; i <= n-step; i++)
              {
                if (a[i-1] > a[i])
                    {
                         temp = a[i-1];
                         a[i-1] = a[i];
                         a[i] = temp;
                         } //end of if
              }     // end of inner loop
     } // end of outer loop
 printf("\n after sorting \n");
 for(i = 0; i<n; i++)
    {
     printf("%d\n", a[i]);
    }

beg = 0;
end = n-1;
mid = (beg+end)/2;

while(beg<=end  && a[mid] != item)
 {
  if(item < a[mid])
                 end = mid-1;
  else
                 beg = mid+1;
  mid = (beg+end)/2;
  }

if(a[mid] == item)
               printf("\nItem found at Loc : %d",mid);
else
                printf("\nItem not found");

}
```

Selection Sort

his type of sorting is called "Selection Sort" because it works by repeatedly element. It works as follows: first find the smallest in the array and exchange it with the element in the first position, then find the second smallest element and exchange it with the element in the second position, and continue in this way until the entire array is sorted.

Procedure Selection_Sort(K, N) This procedure will arrange the elements in ascending order i.e k[0]<=k[1]<=k[2]<=…k[N-1]. The **variable PASS** will denote the pass index and the position of first element in array which is to be examined during a particular PASS. The variable **MIN_INDEX** denotes the position of the smallest element encountered in a particular PASS. The variable I is used to index elements K[PASS] to K[N-1] in a given PASS. All variables are of type integer.	
1	[Loop on PASS index] Repeat thru step 4 for PASS = 0,1,2, … N-2
2	[Initialize minimum index] MIN_INDEX ← PASS

3	[Make a Pass and obtain element with smallest value]
	Repeat for I = PASS+1, PASS+2, … N-1
	    If(K[I] < K[MIN_INDEX])
	        Then MIN_INDEX ← I

4	[Exchange Elements] If (MIN_INDEX <> PASS) Then K[PASS] ↔K[MIN_INDEX]
5	[Finished] Return

/* Selection Sort */

```
# include<stdio.h>
main()
 {
  int a[100], i, step,n, temp, smallest,loc;
  clrscr();
  printf("\nHow many elements U want in array : ");
  scanf("%d",&n);
  printf("\n Input elements in array: \n");
  for(i = 0; i<n; i++)
    {
    scanf("%d", &a[i]);
    }

  for( step = 0; step < n-1; step++)
     {
      smallest = a[step];
      loc = step;
      for (i = step+1; i <= n-1; i++)
              {
               if ( smallest > a[i] )
                   {
                        smallest = a[i];
                        loc = i ;
                        } //end of if
              }     // end of inner loop
      If (loc != step)
         {
          temp = a[step-1];
         a[step-1] = a[loc];
         a[loc] = temp;
        }
     } // end of outer loop
printf(“\nSorted array”);
for(i = 0; i<n; i++)
    {
     printf("%d", a[i]);
    }
} // End of Main
```

From the comparions presented here, one might conclude that selection sort should never be used. It does not adapt to the data in any way (notice that the four animations above run in lock step), so its runtime is always quadratic.

However, selection sort has the property of minimizing the number of swaps. In applications where the cost of swapping items is high, selection sort very well may be the algorithm of choice.

Insertion Sort

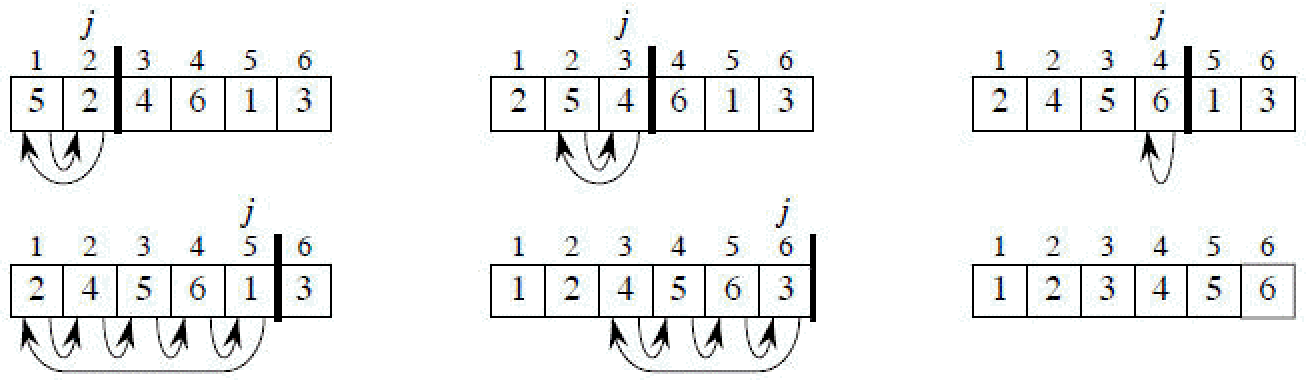
INSERTION_SORT(K, N)	
1	Repeat through step 5 for PASS = 1,2, … N-1
2	TEMP 🡐 K[PASS]
3	PTR 🡐 PASS-1

4	Repeat while (PTR>= 0 AND TEMP < K[PTR]
	        K[PTR+1] 🡐 K[PTR]
	        PTR 🡐 PTR +1

5	K[PTR+1] = TEMP
6	RETURN

PASS 1	K[0] by itself trivially sorted.
PASS 2	K[1] is inserted either before or after k[0] so that k[0], k[1] is sorted
PASS 3	K[2] is inserted in to its proper place in k[0], k[1]. Before k[0] In between k[0] and k[1] After k[1]
PASS 4	K[3] is inserted in to proper place in k[0], k[1], k[2] so that K[0], K[1], K[2], K[3] is sorted.
	And so on…

Although it is one of the elementary sorting algorithms with O(n^2) worst-case time, insertion sort is the algorithm of choice either when the data is nearly sorted (because it is adaptive) or when the problem size is small (because it has low overhead).



Animated sorting

http://www.sorting-algorithms.com/

http://www.cs.pitt.edu/~kirk/cs1501/animations/Sort2.html

http://www.cs.auckland.ac.nz/~jmor159/PLDS210/ds_ToC.html

http://www.programmerinterview.com/index.php/data-structures/big-o-notation/

http://www.personal.kent.edu/~rmuhamma/Algorithms/MyAlgorithms/Sorting/selectionSort.htm

Big-O Notation

Analysis of Algorithms
(how fast does an algorithm grow with respect to N)
(Note: Best recollection is that a good bit of this document comes from *C++ For You++,* by Litvin & Litvin)
The time efficiency of almost all of the algorithms we have discussed can be characterized by only a few growth rate functions:
I. O(l) - constant time
This means that the algorithm requires the same fixed number of steps regardless of the size of the task.
Examples (assuming a reasonable implementation of the task):
A. Push and Pop operations for a stack (containing n elements);
B. Insert and Remove operations for a queue.

II. O(n) - linear time
This means that the algorithm requires a number of steps proportional to the size of the task.
Examples (assuming a reasonable implementation of the task):
A. Traversal of a list (a linked list or an array) with n elements;
B. Finding the maximum or minimum element in a list, or sequential search in an unsorted list of n elements;
C. Traversal of a tree with n nodes;
D. Calculating iteratively n-factorial; finding iteratively the nth Fibonacci number.

III. O(n2) - quadratic time
The number of operations is proportional to the size of the task squared.
Examples:
A. Some more simplistic sorting algorithms, for instance a selection sort of n elements;
B. Comparing two two-dimensional arrays of size n by n;
C. Finding duplicates in an unsorted list of n elements (implemented with two nested loops).

IV. O(log n) - logarithmic time
Examples:
A. Binary search in a sorted list of n elements;
B. Insert and Find operations for a binary search tree with n nodes;
C. Insert and Remove operations for a heap with n nodes.

V. O(n log n) - "n log n " time
Examples:
A. More advanced sorting algorithms - quicksort, mergesort

VI. O(an) (a > 1) - exponential time
Examples:
A. Recursive Fibonacci implementation
B. Towers of Hanoi
C. Generating all permutations of n symbols

The best time in the above list is obviously constant time, and the worst is exponential time which, as we have seen, quickly overwhelms even the fastest computers even for relatively small n. **Polynomial** growth (linear, quadratic, cubic, etc.) is considered manageable as compared to exponential growth.

Order of asymptotic behavior of the functions from the above list:

Using the "<" sign informally, we can say that

O(l) < O(log n) < O(n) < O(n log n) < O(n2) < O(n3) < O(an)

A word about O(log n) growth:

As we know from the Change of Base Theorem, for any a, b > 0, and a, b   1

logb n = $(\log_a n)/\log_a b$
Therefore, **loga n = C logb n**

where C is a constant equal to loga b.

Since functions that differ only by a constant factor have the same order of growth, **O(log2 n)** is the same as **O(log n).**

Therefore, when we talk about logarithmic growth, the base of the logarithm is not important, and we can say
simply **O(log n).**

Stacks

Principle used : Last in First Out (LIFO)

It is a linear data structure, in which insertion and deletions are made from one end called top of stack.

Operations

Push: inserting element in to stack
Pop: Deletion from stack

Applications in computer Science

- Conversion of infix expression in to Polish notations.
- Evaluation of Polish notation Expression
- Recursion
- Bracket Matching Algorithm

Algorithm for Push, Pop and display

Push(Stack, item, top, capacity)

Where stack is a array of size capacity, top is the tracker of topmost element of stack. Initially top = -1
Item is the value which is to be push

```
1 if (top >= capacity-1)
        Output " Stack Overlow"
        Return top
2 top = top +1
3 stack[top] = item
4 return top
```

POP (Stack, top)

```
1 if (top == -1)
       Output " Stack Underflow"
       Return -1
2 top = top -1
3 return top
```

Display(stack, top)

```
1 i=0
2 while (i < top)
    {
       Output stack[i]
        I = I +1
    }
3 return
```

Infix Expression: A * B

Postfix Expression: AB*

Prefix Expression: *AB

Algo for Infix to PostFix Expression

1. Print operands as they arrive.
2. If the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
3. If the incoming symbol is a left parenthesis, push it on the stack.
4. If the incoming symbol is a right parenthesis, pop the stack and print the operators until you see a left parenthesis. Discard the pair of parentheses.
5. If the incoming symbol has higher precedence than the top of the stack, push it on the stack.
6. If the incoming symbol has equal precedence with the top of the stack, use association. If the association is left to right, pop and print the top of the stack and then push the incoming operator. If the association is right to left, push the incoming operator.
7. If the incoming symbol has lower precedence than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

Example

Infix Expression	**Postfix Expression**
A + B	**A B +**
A + B * C	A B C * +
(A + B) * C	A B + C *
A + B * C + D	A B C * + D +
(A + B) * (C + D)	A B + C D + *
A * B + C * D	A B * C D * +

Example

Convert the following infix epression into postfix using tabular approach
A * (B + C * D) + E

	Current symbol	**Operator Stack**	**Postfix string**
1	**A**		**A**
2	*	*	**A**
3	**(**	* (	**A**
4	**B**	* (	**A B**
5	+	* (+	**A B**
6	**C**	* (+	**A B C**
7	*	* (+ *	**A B C**
8	**D**	* (+ *	**A B C D**
9	**)**	*	**A B C D * +**
10	+	+	**A B C D * + ***
11	**E**	+	**A B C D * + * E**
12			**A B C D * + * E +**

Exercises for Infix to Postfix conversion using tabular approach

Exercise 1

a+b*(c^d-e)^(f+g*h)-i

Here ^ is Exponent symbol

Ans abcd^e-fgh*+^*+i-

Exercise 2

(300+23)*(43-21)/(84+7)

Ans 300 23 + 43 21 - * 84 7 + /

Algorithm for Evaluation of Postfix Expression

1. Add) to postfix expression.
2. Read postfix expression Left to Right until) encountered
3. If operand is encountered, push it onto Stack
   [End If]
4. If operator is encountered, Pop two elements
   i) A -> Top element
   ii) B-> Next to Top element
   Evaluate B operator A
   iii)

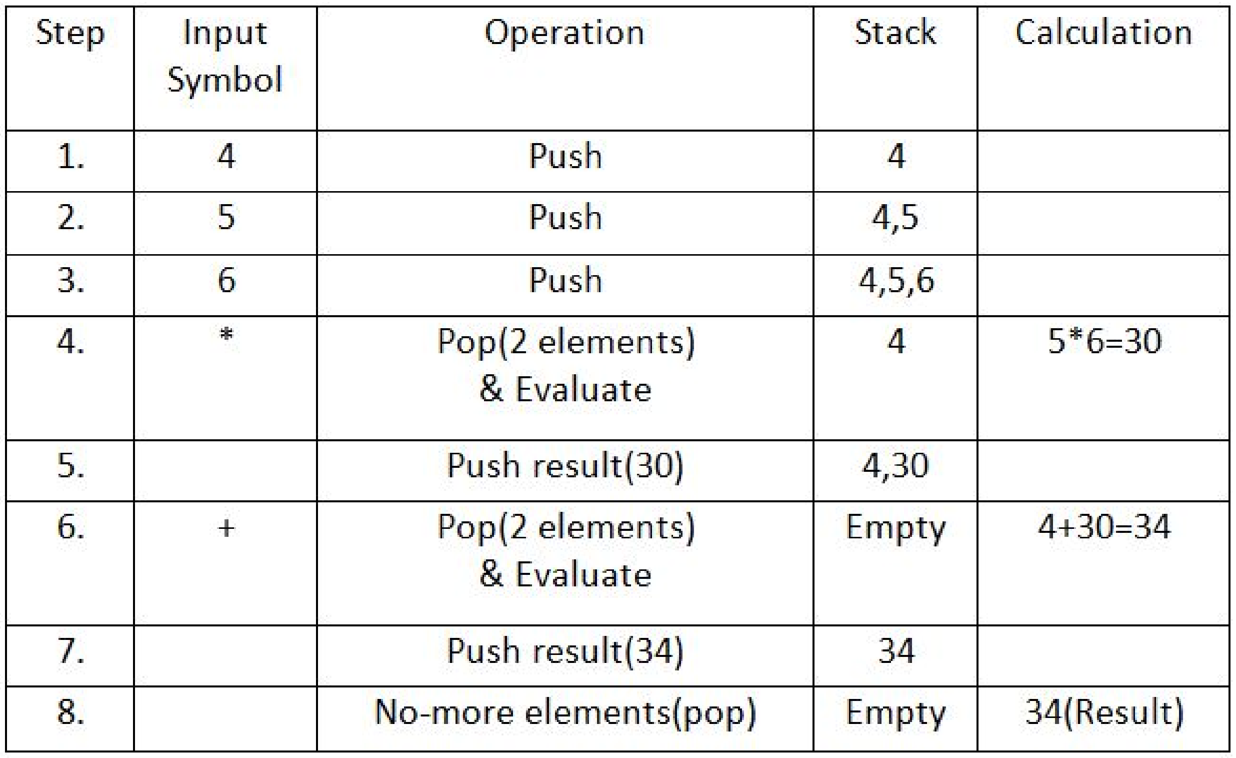
5 push B operator A onto Stack

6 Set result = pop

END

Example

4 5 6 * +



Step	Input Symbol	Operation	Stack	Calculation
1.	4	Push	4	
2.	5	Push	4,5	
3.	6	Push	4,5,6	
4.	*	Pop(2 elements) & Evaluate	4	5*6=30
5.		Push result(30)	4,30	
6.	+	Pop(2 elements) & Evaluate	Empty	4+30=34
7.		Push result(34)	34	
8.		No-more elements(pop)	Empty	34(Result)