Date
Page

Data structure: Data structure is logical and mathematical model of storing and organizing data in a particular way that it can be required for design and implimentation of algorithm.

Basic Terminology:
Data: Set of values / value
Data items: Single unit of value
Records: Collection of various data, items
File: Collection of records of one type
Field: Single elementary unit representing attribute of a entity.
Information: Data with attributes or meaning -ful data.

Classification of Data structure (DS.

Primitive DS	Non-primitive DS
int, Float, char, double, boolean	- Derived from primitive
Directly interpret with the system or machine / hardware part.	- Nonprimitive DS not directly interact with machine.
Primitive DS are the DS whose property or operation already known to compiler or computer system	- With the help of Primitive DS it can be interact with the machine.

- linear non-primitive DS
  - Array
  - Linked list
  - Stack
  - Queue
- Non-linear NPDs
  - Tree
  - Graph

Linear NPDS	Non-linear NPDS
1) All element are arrange in linear order - Each element has successar and predece -ssor exce ex: Arrag.	This data structure does not form a sequenti ex: Tree.
2) single level involve	2) Multi level involve
3) Data element in single run traversed	3) can't be traversed in single run.
4) Use in software developement	4) Use in AI, DIP, digital image processing.

* Data structure Operations:
1) Traversing
2) Searching
3) Inserting
4) Deleting
5) Sorting
6) Merging

1) Traversing: Visiting each and every element exactly once.

Algorithm:

Step 1: Read A [5]
Step 2: For (K=1 to K≤N)
Step 3: display A [K]
Step 4: exit.

```
# include<iostream.h>
  Void main ()
  {
    int A[7] = {10,20,30, 40, 50}
     for (i=1; i<=n; i++) {
      printf ("%d" , A[i]);
     }
```

Time complexity : O(n)

* Insertion: Adding an element in a linear array.

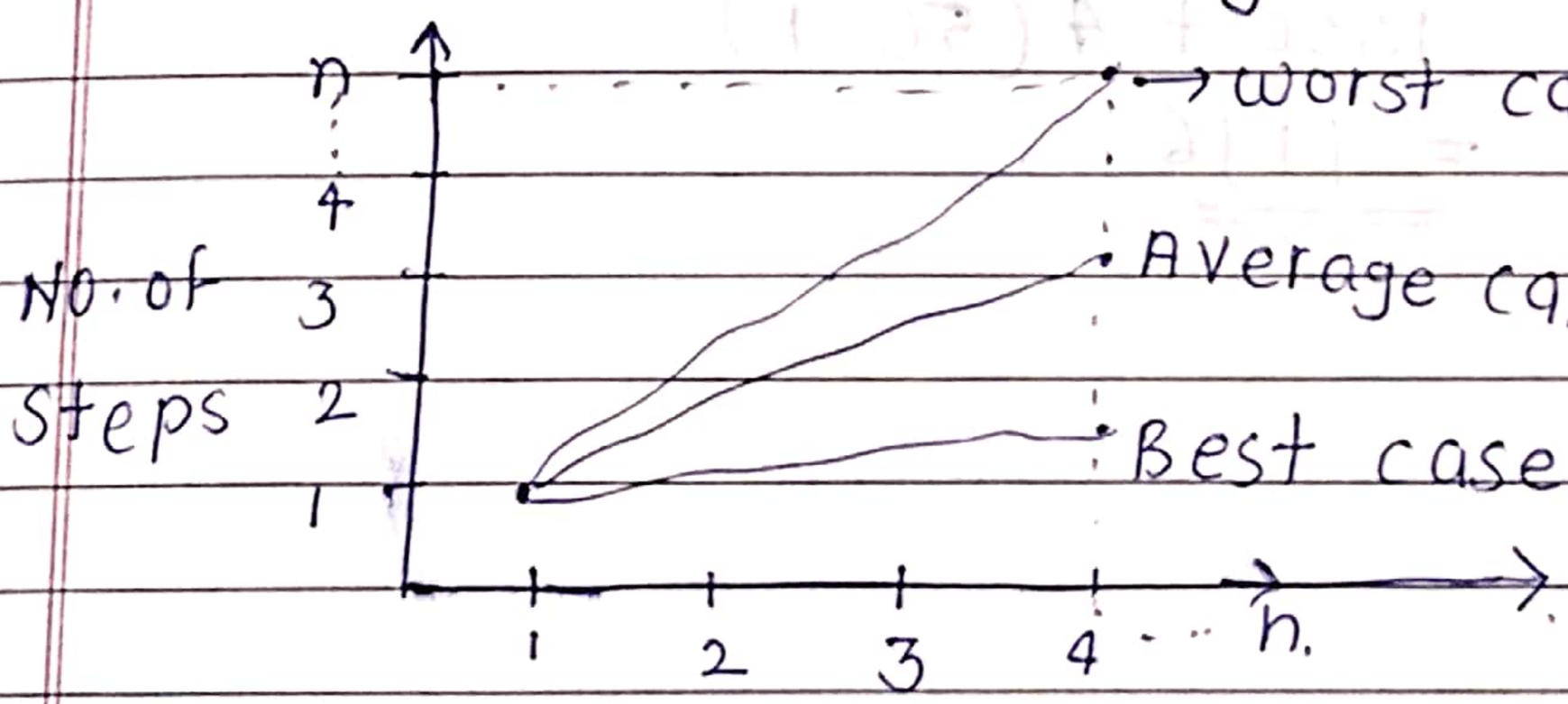
There are 3 cases : → O(n)

i) Insert ~~O(n)~~ element at begining : Worst case.

ii) Insert element at end : O(1) : best case

iii) Insert element at given ~~case~~ position : O(n): Average case.

* less time + less memory = efficient algorithm.



★ Algorithm of insertion.

Insert (A, N, K, item)

Step 1: Set J = N
Step 2: Repeat step 3 & 4 while J >= K
Step 3: say A[J+1] = A[J]
Step 4: J = J-1
Step 5: set A[K] = item
Step 6: set N = N+1
Step 7: Exit.

Q. Finding location of an array

array: 1 to 100 {1, 2, ---99, 100}
base address: 1000
size: 4 bytes
Find the location of a[50]:

$$a[50] = b + w(i - lb)$$
$$= 1000 + 4(50 - 1)$$
$$= 1196$$

Void insertion([], int)

```
void main ()
 {
  int a[10], len, ch,
  pf ("Enter no. of elements you want in
        array");
  sf ("%d", &len);
pf ("Enter no of elements");

For (i=0 ; i< len. ; i++)
  { sf ("%d", &a[i]);
   }
do
  { pf ("operation you want to perform");
    pf ("1. Insertion");
    pf ("2. deletion");
    pf ("3. Display");
    pf ("4. Exit");
    sf ("%d", &ch);

switch (ch)
 {
 case 1: insert (a, len);
         break;
case 2: insert delete (a, len);
         break;
case 3:
```

* linear search: Array can be sorted/not sorted.

Lsearch (A, N, item) = (sequential search).

Algo:

step1: Repeat step 2 for i = 0 to N-1

step 2: If (A[i] = item)

{ loc = i ;

break;

}

step3: IF (i == N)

Pf("Element not found");

else.

pf ("Element found at loc):

* ~~Best~~ complexity:

Best case	Worst case	Avg. case.
O(1)	O(n)	O(n)

* Binary search:

Array can be sorted.

Algorithm:

Binary search (A, low, high, item)

step1: mid : (low + high)/2.

2: Repeat step 3 & 4 while low ≤ high, &

a[mid ≠ ~~high~~] item

3: if (item < a[mid] then

set high = mid - 1;

else

set low = {mid + 1;

★ Application of stack:

1) Conversion of Arithematic expression:
   1) Infix expression
   2) Prefix expression (Polish notation)
   3) Postfix expression (Reverse polish notation)

★ Precedence of operators:

1) ( ) Brackets
2) ↑ exponential
3) * / Multiplication, Division
4) + - Addition, subtraction

1) Infix: operand 1 operator operand 2
   a + b.
   ex: (x-y) - (P+q)

2) Prefix: operator operand 1 operand 2
   + a b
   ex: ~~- (x-y) (P+q)~~
   (x-y) - (P+q) $\xrightarrow{\text{Prefix}}$ --xy+Pq

3) Postfix: Operand 1 operand 2 operator
   a b +
   (x-y) - (P+q) $\xrightarrow{\text{postfix}}$ ~~(xy-) (Pq+) =~~
   xy- Pq+ -

★ Infi

VIMP in exam must

1) Infix to postfix: using stack:
Algorithm:

step 1: Push '(' on to the stack & ')' to end of expre
step 2: Scan expression left to right and repeat step 3 to 6.
step 3: If an operand encountered add to postfix expression.
step 4: If left parenthesis encountered 'C' push to stack.
step 5: If operator encountered then
(a) Repeatedly pop from stack, add to postfix expression for each operator same or highest precidance.
(b) Add operator to stack
Step 6: If ')' right parenthesis encountered then
(a) Pop from stack and add to postfix expression untill left parenthesis.
(b) Remove left parenthesis.
Step 7: Exit.

ex: ① (A * B) + (C * D)
VIMP ② A * (B + D) / E - F * (G + H / K)

Symbol	stack	postfix exp.
A	(	
*	(	A
(	(*	A
B	(*(	*A
+	(*(+	AB
D	(*(+	AB
)	(*	ABD
		ABD+

/	(★/	ABD+
E	(★/	ABD+E
-	~~(★~~ (-	ABD+E/★
F	(-	ABD+E/★F
★	(-★	ABD+E/★F
(	(-★(	ABD+E/★F
G	(-★(	ABD+E/★FG
+	(-★(+	ABD+E/★FG
H	(-★(+	ABD+E/★FGH
/	(-★(+/	ABD+E/★FGH
K	(-★(+/	ABD+E/★FGHK
)	(-★	ABD+E/★FGHK/+
)	- -	ABD+E/★FGHK/+★-

③ a+(b+c★d+e)+f/g)

symbol	stack	postfix exp.
	(	
a	(	a
+	(+	a
(	(+(	a
b	(+(	ab
+	(+(+	ab
c	(+(+	abc
★	(+(+★	abc
d	(+(+★	abcd
+	(+(+	abcd★+
e	(+(+	abcd★+e
)	(+	abcd★+e+
+	(+	abcd★+e++
f	(+	abcd★+e++f
/	(+/	abcd★+e++f
g	(+/	abcd★+e++fg
)	-	abcd★+e++fg/+

4) A $B X C - D + E / F / (G + H))

symbol	stack	Postfix exp
~~A~~	C	
A	(	A
$	($	A
B	($	AB
X	($X	AB$
C	($X	ABC
-	(-$X	AB$CX
D	(-$X	AB$CX D
+	(-+	AB$CX D
E	(-+	AB$CX D - E
/	(-+/	AB$CX D - E
F	(-+/	AB$CX D - EF
)	(-+//	AB$CX D - EF
G	(-+//	AB$CX D - EFG
+	(-+//(+	AB$CX D - EFG
H	(-+//(+	AB$CX D - EFGH
)	(-+//	AB$CX D - EFGH+
)	--	AB$CX DEFGH+//+-

2) ★ Conversion of infix to prefix

step 1: Reverse the infix expression.
ex: (a+b) * c → c * (b+a))

~~(a*b (a+b)*c~~

prefix: * P C a+b = P.

* + a b c

reverse order: - C * (b + a)

symbol	stack	exp: postfix
	(	
c	(	c
*	(*	c
(	(*(	c
b	(*(	cb
+	(*(+	cb
a	(*(+	cba
)	(*	cba+
)	*--	cba+*

Reverse: * + a b c ⇒ Prefix

(d-c) * (b-a)

~~reverse~~:	X * Y	reverse order:
	* X Y	(a-b) * (c-d))
prefix: ⇒	* - d C - b a	

symbol	stack	postfix exp.
	(	
(	((	
a	((	a
-	((-	a
b	((-	ab
)	(	ab-
*	(*	ab-
(	(*(	ab-
c	(*(	ab- c

-	(*(-	ab-c
d	(*(-	ab-cd
)	(*(	ab-cd-
)	--	ab-cd-*

Reverse order ⇒ *-edc-ba ⇒ Prefix

2) EValution of postfix expression:

step 1: Add round bracket ')' at the end of expression.

step 2: Scan the expression from left to right untill closing ')' bracket encounters

step 3: If an operand encounters post it to stack.

step 4: IF an operator encounters then
i) Pop top two operand from stack
First pop operand is denoted by 'OP1'
Second pop operand is denoted by 'OP2'.
ii) Evaluate OP2 ⊕ OP1
iii) Put that answer to stack

step 5: POP of the ~~to~~ stack will be the final answer.

Step 6: Exit.

5 6 2 + * 12 4 / -)

Symbol	Stack	OP2 ⊕ OP1
5	5	
6	5 6	
2	5 6 2	
+	5 8	6 + 2 = 8
*	40	5 * 8 = 40
12	40 12	
4	40 12 4	
/	40 3	12 / 4 = 3
-	40 37	40 - 3 = 37

4 5 4 2 ^ + * 2 2 ^ 9 3 / * -)

Symbol	stack	Op2 ⊕ Op1
4	4	
5	4 5	
4	4 5 4	
2	4 5 4 2	
^	4 5 16	4 ^ 2 = 16
+	4 21	5 + 16 = 21
*	84	21 * 4 = 84
2	84 2	
2	84 2 2	
^	84 4	2 ^ 2 = 4
9	84 4 9	
3	84 4 9 3	
/	84 4 3	9 / 3 = 3
*	84 12	84 * 3 = 72
-	72	
)	72	

100 200 + 2 / 5 * 7 +)

Symbol	stack.	OP2 ⊕ OP①
100	100	
200	100 200	100 + 200 = 300
+	300	
2	300 2.	
/	150	300/2 = 150
5	150 5	
*	750	150 * 5 = 750
7	750 7	
+	757	750+7 = 757