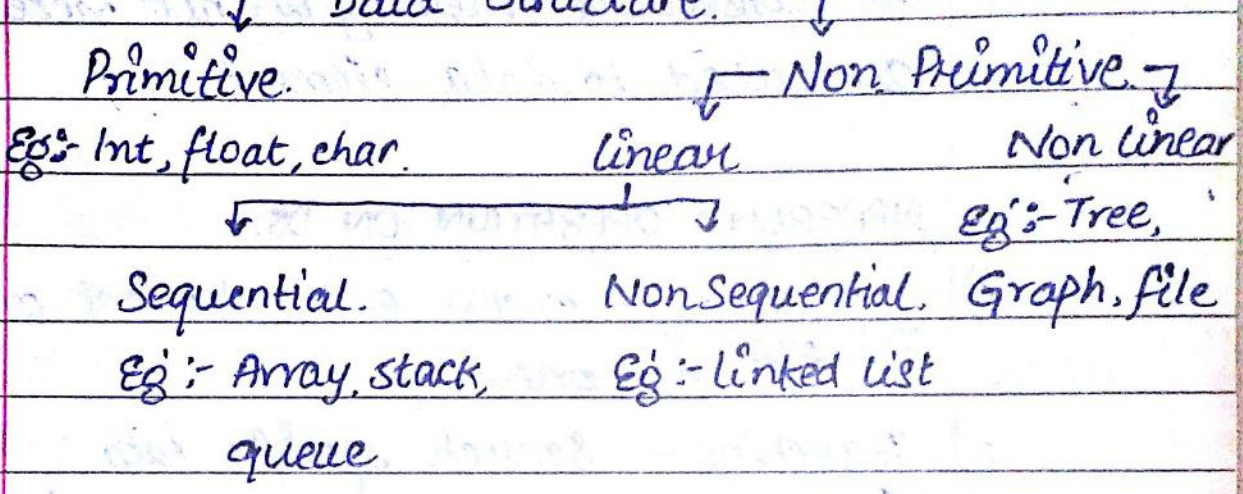
3 August 2012

Date:

DATA STRUCTURE

The organised way of data in a logical and in a mathematical terms is called data structure



NON PRIMITIVE DS.

Having no. specific instruction to store the individual data item

PRIMITIVE DS

The DS which is directly supported by machine

LINEAR DS

DS using sequential memory for storing data elements.

NON LINEAR DS.

DS using link allocation for the data elements are non linear. DS.

SEQUENTIAL DS.

DS with consecutive memory location are used for storing data elements

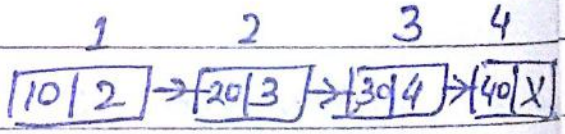
NON SEQUENTIAL DS.

The consecutive memory is not necessarily allocated to data element.

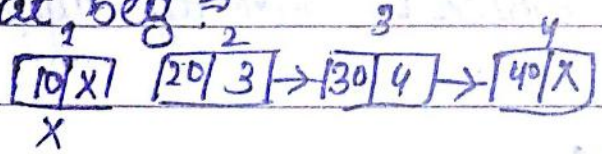
DIFFERENT OPERATION ON DS.

1) ~~Transversing~~ Traversing - excess each element one by one
2) Searching - Search specific data
3) Sorting - storing data in ascending/descending order
4) Copying
5) Merging - combining two lists
6) Deletion

Eg:- In linked list

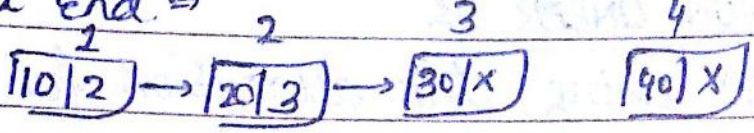


at beg ⇒



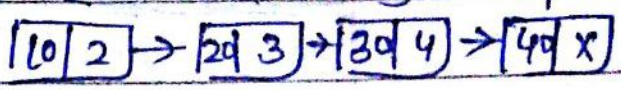
at mid ⇒

at End ⇒

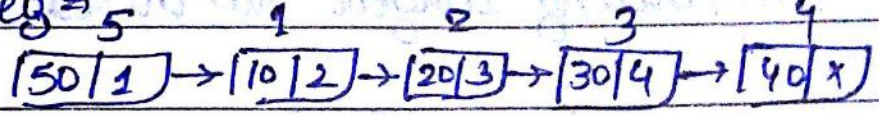


7) Insertion.

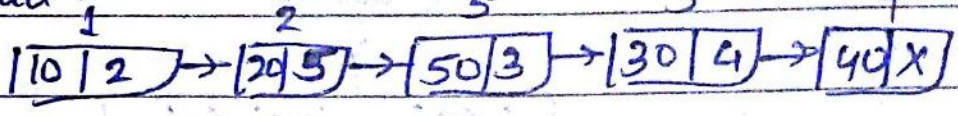
Eg:- In linked list.



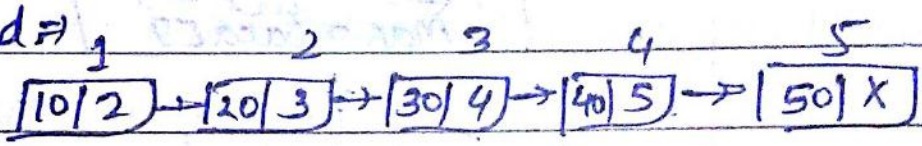
at beg ⇒



at mid ⇒



at End ⇒



COMPLEXICITY OF ALGORITHM.

It gives running time and space in terms of input size.

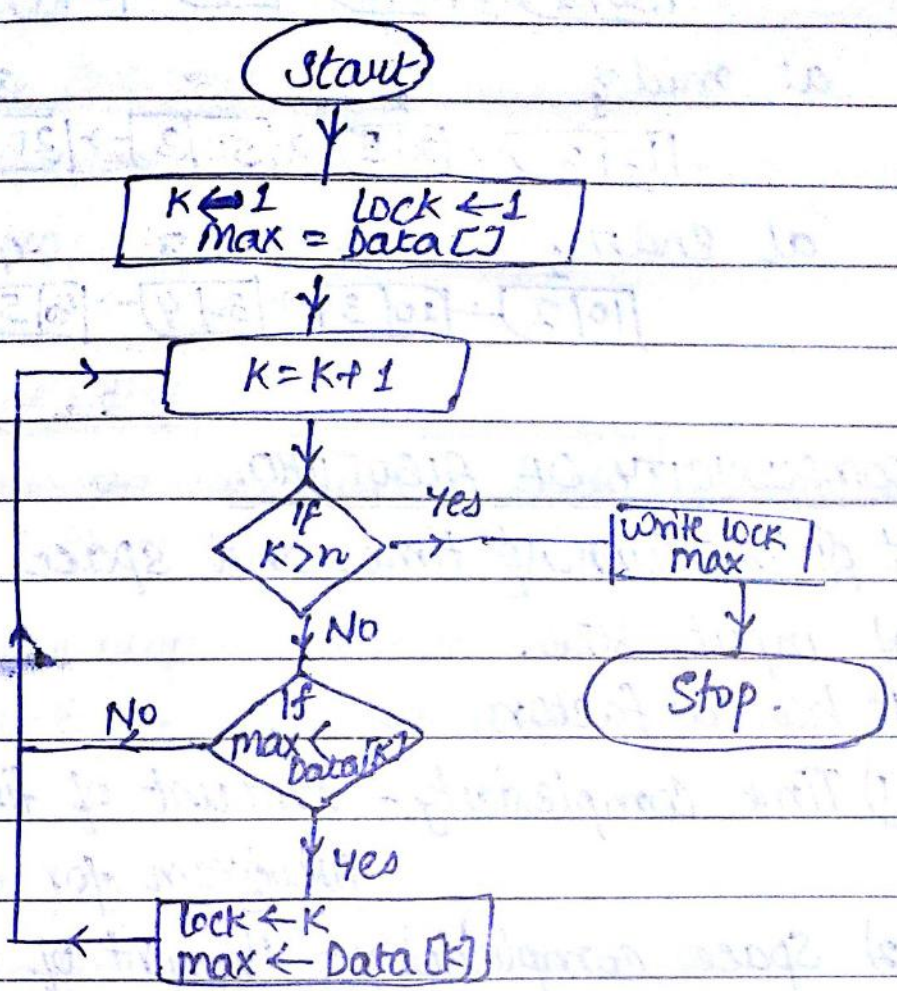
It has 2 factors.

1) Time complexicity - Amount of time rqd. by program for execution.
2) Space complexicity - Amount of space rqd. by program for execution.

August 2012

ALGORITHM

1) Finding Largest no. in an array



Algorithm :-

1) Set K ← 1 lock = 1 and max = Data[1]
2) Set K = K+1
3) If K > N, then write lock, max and exit
4) If max < Data[K] then set loc = K and max = Data[K]
5) Go to Step 2.

ALGORITHM ANALYSIS

They are of three type

1) worst case
2) Best case
3) Average case.

Max. time taken by program to execute is worst case. (lower mode)

Best/least time taken refers to best case (upper mode)

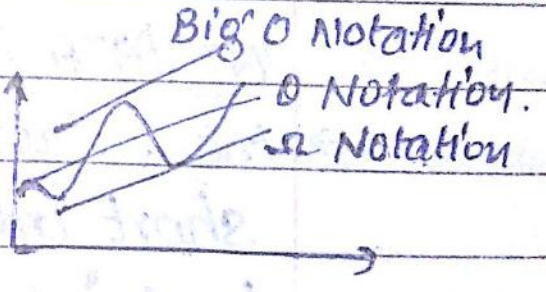
Average time → Average case.

ASYMPTOTIC NOTATION.

It refers to time taken.

three notations.

i) Big -O Notation. — upper bond
ii) Θ Theta Notation — Average bond
iii) Ω Notation — Lower mode (bond)

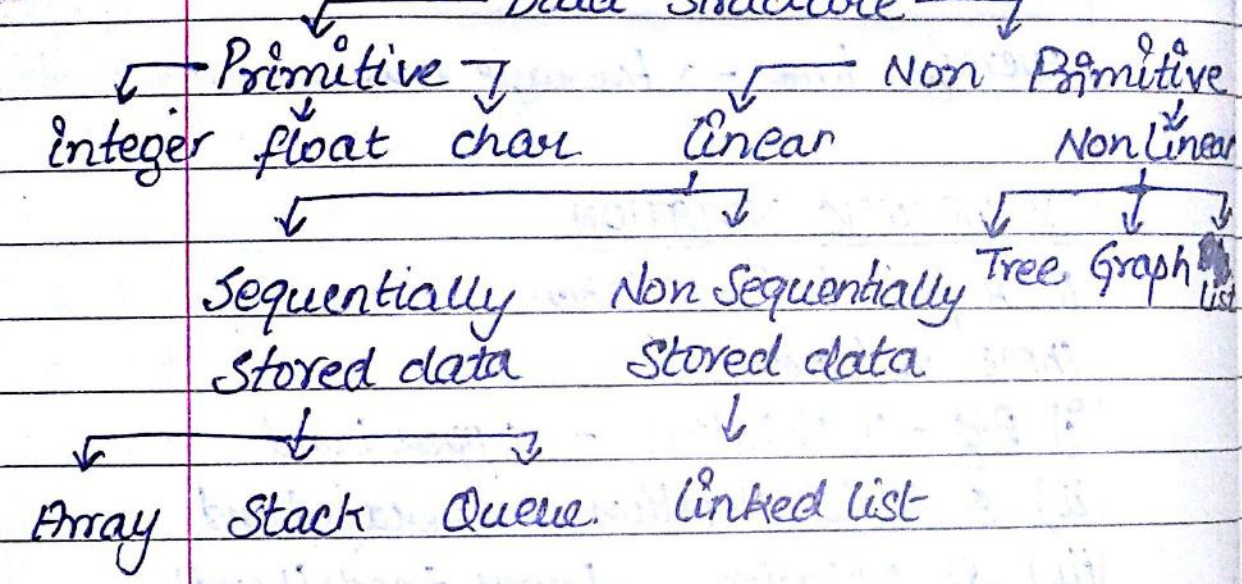


COMPLEXICITY OF AN ALGORITHM.

DATA STRUCTURE

1) It is the way of organising data and storing it into a format to get the desired information as and when required. or
2) The logical or mathematical modes of particular data organisation is called as data structure.

CLASSIFICATION OF DATA STRUCTURE



INT DATA TYPE

→ Int., Short Int, long Int, unsigned short Int, unsigned int, unsigned long int,

→

Int	2 bytes	-2^{15} to $2^{15}-1$
Short Int	2 bytes	-2^{15} to $2^{15}-1$
long Int	4 bytes	2^{31} to $2^{31}-1$
unsigned int	2 bytes	0 to $2^{16}-1$
unsigned short int	2 bytes	0 to $2^{16}-1$
unsigned long int	4 bytes	0 to $2^{32}-1$

CHAR DATA TYPE

→ Signed char −128 to 127 1 byte
unsigned char 0 to 255 1 byte.

FLOAT DATA TYPE

→ Float $3.4E^{-38}$ to $3.4E^{38}$ 4 bytes
Double $1.7E^{-308}$ to $1.7E^{+308}$ 8 bytes
Long Double. $3.4E^{-4932}$ to $3.4E^{4932}$

PRIMITIVE DATA STRUCTURES.

The data structure which are directly supported by the machine. Eg:- Int, float, char

NON PRIMITIVE DATA STRUCTURES

The data structure having no specific instruction to manipulate the individual data item

LINEAR DATA STRUCTURES

Data Structure using sequential memory location for the data elements. Eg :- Array, stack, Queue.

NON LINEAR DATA STRUCTURES

Data Structure using linked allocation for the data elements are called as non linear data structures. Eg:- List, tree, Graph.

SEQUENTIALLY STORED DATA

Data Structure where the consecutive

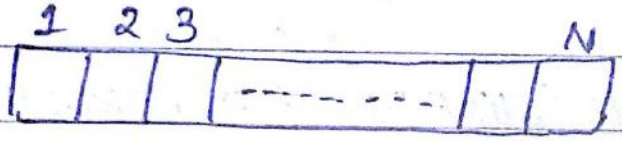
memory is allocated to the data element.

NON SEQUENTIALLY STORED DATA STRUCTURE

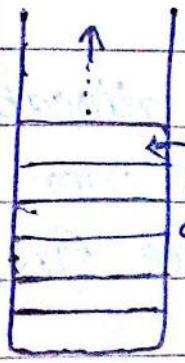
Data Structure where consecutive memory is not necessarily allocated to data elements.

ARRAY.

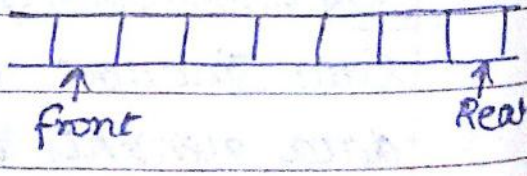
It is the ordered set of homogenous elements sharing a common name and consists of a fixed no: of elements



Stack



Queue

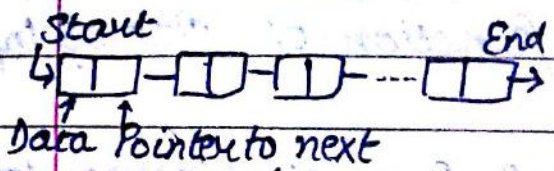


Insertion take place at rear. Deletion take place at.

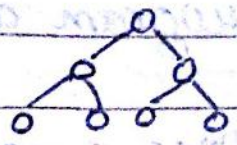
Linke
Start
Data Pointer
OPERATI
1) Transve
Structu
the da
2) Search
3) Sortin
or des
4) Insertio
5) Deletio
ALGORIT
An alg
well de
ular p
Types :-
COMPLEXI
The com
which
in term

Date:

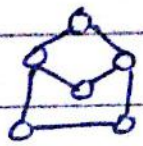
Linked Lists:



Tree



Graph



OPERATION ON DATA STRUCTURE

1) Transversing :- Accessing each element of data structure, once and doing some operation on the data
2) Searching :- Searching a specified data
3) Sorting :- Sorting the data in ascending or descending order.
4) Insertion :- Inserting a new data element
5) Deletion :- Deleting a specified data element

ALGORITHM

An algorithm is a finite step by step list of well defined instruction for solving a particular problem

Types :- 1) Sequential 3) Iterative
2) conditional

COMPLEXITY OF ALGORITHM :-

The complexity of algorithm is the function which gives the running time and/or space in terms of the input size.

DELTA Page No. Date:

TIME COMPLEXITY:- Running time of the Program as a function of size of input

SPACE COMPLEXITY:- Amount of computer memory required during program execution, as a function of size of input

LINEAR DATA STRUCTURE / LINEAR LIST

→ In this data can be processed in linear fashion i.e. one by one, sequentially
→ It includes following types of data structure
a) Array b) Linked list c) Stack & Queue.

ARRAY

→ Array is a linear data structure
→ It is a collection of elements of similar data type.
→ It is a list of a number n of homogenous data elements. eg:- A[10].

LINKED LIST

→ It is collection of nodes
→ A node has two field i.e. information or data and address field or link field.
→ The data field stores the actual piece of information.
→ Link/address field is used to store address

of the

STACK

A stack
out sy
insertio
one end

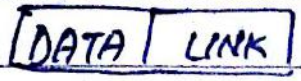
QUEUE

It is a li
takes pl
deletion
end. It
first ou

NON LIN

A data
is not
are n
a) Tree
In this
hierar
elemen
b) Gra
Data
pairs
hiera

of the next node. i.e. pointers to the next node.



STACK

A stack is also called as LIFO i.e. last in first out system. It is a linear list in which insertion and deletion can take place only at one end called top

QUEUE

It is a linear data structure in which insertion takes place at one end called the rear end and deletion takes place at other end called front end. It works on basis of FIFO (first in first out) or FCFS (First come first serve)

NON LINEAR DATA STRUCTURES

A data structure in which insertion & deletion is not possible in a linear fashion. Following are non linear data structures

a) Tree

In this data structure, data contain a hierarchical relationship between various elements.

b) Graph

Data sometimes contains relationship b/w pairs of elements which is not necessarily hierarchical in nature. Eg:- air line flow

between cities connected by lines.

SPECIAL OPERATIONS ON DATA STRUCTURES

1) Merging :- Combining the records in two different sorted files into a single sorted file
2) Copying :- Copying contents of one file to another
3) Concatenation :- Combining the ordered or unordered data of two files.

TERMS

1) DATA :- Data are simple values or set of values.
2) DATA ITEMS :- It is single unit of values

Data Items
Group Items Elementary Items

3) GROUP ITEMS :- These are those data item that are divided into sub item eg :- Name may be divided into first, middle and last name.
4) ELEMENTARY ITEMS :- These are those data items which are not sub divide
5) ENTITY :- Entity has certain attribute or properties which may be assigned values
6) ENTITY SET :- Entities with similar attribute form an entity set. Collection of data

are organi
→ records –
7) FIELD :- field
information
an entity
or column.
8) RECORD/ROW
values of a
9) PRIMARY KEY
uniquely a
file may
field (K) is
i.e K_1, K_2 ...
key values.
Example :-

Rol
1
2
3.

1) Rollno. Na
2) Each Stu
3) collection
4) Field valu
attributes
5) Collection
record /t
6)

are organised into a hierarchy of fields → records → files.

7) **FIELD** :- field is a single elementary unit of information representing an attribute of an entity. Fields are also known as attribute or column.

8) **RECORD/ROW/TUPLE** :- Record is the collection of field values of a given entity.

9) **PRIMARY KEY** :- The value in a certain field uniquely data record as each record in a file may contain many field items. Such a field (K) is called primary key and values i.e. K_1, K_2 ---- in such field are called keys or key values.

Example :-

Student

Rollno	Name
1	John
2	Willams
3.	Antony

1) Rollno, Name are attribute, field, column.
2) Each student is an entity.
3) Collection of all entities is entity set.
4) Field values are values assigned to attributes like 1, 2, 3, John, willams, antony.
5) Collection of field values of each student is record / tuple / row
6) Collection of all records make file

7) Primary key is null no

ANALYZING ALGORITHMS

An algorithm analysis measures the efficiency of the algorithm.
The efficiency of algorithm is checked by
i) The correctness of algo.
ii) The implementation of an algo.
iii) The simplicity of an algo.
iv) The execution time and memory (space req.) of algo
v) The new ways of doing same task even faster.

SPACE TIME TRADE OFF ALGORITHM.

The space time trade off refers to a choice between algorithmic solution of a data processing problem that allow one to decrease the running time of an algorithmic solution by increasing the space to store the data and vice versa.

ALGORITHM ANALYSIS

i) worst case :- The maximum value of f(n) for any possible input.
ii) Average case :- The expected value of f(n)
iii) Best case :- Minimum possible value of f(n)

GROWT
- Suppose size of M inc
- It is that is $\log_2 n$
- Algori to ex

CATEGO
1. Const
2. Logar
3. Linea

ASYMP
The r runn terms an o of o the perf
i) BIG-
The ther suc

GROWTH OF ORDER

- Suppose. M is an algorithm and suppose n is the size of input data, then complexity f(n) of M increases as n increases.
- It is usually the rate of increase of f(n) that is required to examine
  $\log_2 n < n < n\log n < n^2 < n^3 < 2^n$ .....
- Algorithm function grow slowly as compared to experimental terms.

CATEGORIES OF ALGORITHM

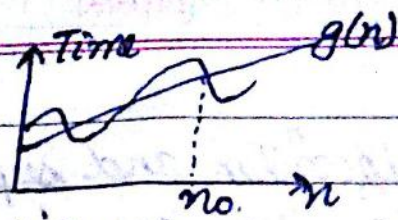
1. Constant time. algo.
2. Logarithmic algo
3. Linear time algo.

ASYMPTOTIC. NOTATION (O, θ, Ω)

The notation use to describe the asymptatic running time of an algorithm is defined in terms of function. Performance evolution of an algo is obtained by totalling the number of occurence of each operation when running the algo. Following notation are used in performance analysis

i) BIG-OH (O) NOTATION (Upper bound)

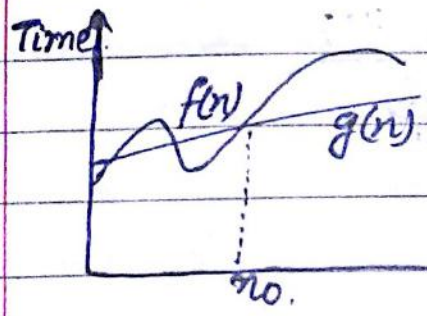
The function $f(n) = O(g(n))$ iff (if & only if) there exists positive constant c and n_0. such that $f(n) \leq c \cdot g(n)$ for all $n, n \geq n_0$



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

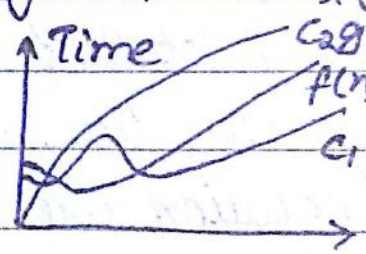
Generally it is used to find worst running time. $f(n)$ computes the actual time to execute the algo.

ii) Ω (OMEGA) NOTATION (lower bound) :-
The function $f(n) = \Omega(g(n))$ iff there exists +ve constant c and no such that $f(n) \geq c \cdot g(n)$ for all $n \geq n_0$.



$f(n) = \Omega(g(n))$

iii) Θ (THETA)- NOTATION (Average bound)
The function $f(n) = \Theta(g(n))$ iff there exists +ve constants c_1, c_2 & n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all $n, n \geq n_0$



$f(n) = \Theta(g(n))$

ALGORITHM COMPLEXITY

The choice of particular algo depends
1) Performance requirement i.e. time complex
2) Memory requirement i.e. space complexity
3) Programming Requirement-

→ Performa
critical
→ It is not
unless
require
→ Therefor
only on
runnin

SPACE C
The space
of mem
Space
followin
i) Instruct
table
ii) Data
constan
the c
- Space
this sp
- Space
such
- Dyna
usual
iii) Envir
Space
to re

→ Performance requirements are usually more critical than memory requirements.
→ It is not necessary to worry about memory unless they grow faster than performance requirements.
→ Therefore, in general, the algo are analysed only on basis of performance requirement i.e. running time efficiency.

SPACE COMPLEXITY

The space complexity of an algo is the amount of memory it needs to run to completion.
Space needed by a program consists of following components:

i) Instruction Space :- Space needed to store executable version of the program
ii) Data Space :- Space needed to store all constants, variable values and has further functions, the components.
- Space needed by constants and simple variable this space is fixed
- Space needed by fixed sized structured variable such as arrays and structures
- Dynamically allocated space. This space is usually varying.

iii) Environment Stack Space
Space needed to store the information needed to resume the suspended (partially completed)

functions. Each time a function is invoked the following data is saved on environment stack:-

- Return address i.e. from where it has to resume after completion of called fn.
- Values of all local variables and values of formal parameters in the function being invoked.

iv) Reverse Stack Space:-

The amount of space needed by recursive function. This space depends on space needed by the local variables & formal parameters. In addition, this space depends on the maximum depth of recursion

In general, total space needed by a program can be divided into 2 parts

i) A fixed part - that is independent of particular problem and includes instruction space, space for constants, simple variables and fixed size structure variable

ii) A variable part - that includes structured variables whose size depends on particular problem being solved dynamically allocated space and the recursion stack space

TIME COMPLEXITY

The time complexity of an algo is the amount of time it needs to run to completion.
The reason for knowing time complexity are following :-

i) It is interesting to know in advance that whether the program will provide a satisfactory real time response for eg :- an interactive program such as an editor, must provide such a response.- if it takes even few sec to move the cursor at page up to down, it will not be acceptable to the user.

BIG O NOTATION

Big O Notation is the formal method for expressing upper bound of an algorithm's running time. It is measure of the largest amount of time it could possibly take for the algorithm to complete. It is generally not possible to perform a simple analysis of algorithm to determine the exact time required to execute it, reasons are :-

i) Exact time depends on the implementation of algorithm and on the machine we are executing on.

ii) Time requirement to be found in time analysis

depends on amount of input.

* Complexity expressed in O-notation is on an upper bound and the actual comp may be much lower.
* So this complexity can be treated as worst case complexity
* The constants c and n_c are known & are not necessarily small.

Different effects of constants on f(n) are shown.

n \	log n	n	n log n	n^2	n^3	2^n
5	3	5	15	25	125	32
10	4	10	40	100	10^3	1024
100	7	100	700	10^4	10^6	$(1024)^{10}$
1000	10	1000	10^4	10^6	10^9	$(1024)^{100}$

In General, the most computating times algorithms written as

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2)$
$< O(n^3) < O(2^n)$

For example:

1) Statement;

This statement is constant O(1) The running time of the statement will not change in relation to N.

2) for(i=0; i<N; i++)
Statement;

3) for(i=
{ for
Stat
}

This s
time
squar

4) while
{ m
if
hi
else
lo
el

This
The ru
propor
by d
with
with
this

5) For(i
{ m
whil
{ m
}

This statement is linear. The running to of the loop is directly proportional to (N)

3) for(i=0; i<N; i++)
{ for (j=0; j<N; j++)
Statement;
}

This statement is quadratic. The running time of the two loops is proportional to the square of N i.e. N^2.

4) while (low< high)
{ mid= (low+high)/2
if target < list [mid]
high= mid-1;
else if (target > list[mid])
low = mid+1;
else break; }

This statement is of log statement
The running time for this statement is proportional to the number of time N can by divided by 2. This is because the algorithm divides the work area in half with each iteration. The running time of this algo is O(nlogn)

5) For (i=0; i<n; i++)
{ m=n
while (m>1)
{m= m/2; }
}

This type of algo solve a problem by breaking it up into smaller point problems and than solve the problem.

——X——X——X——

8 August 2012

Date:

ARRAYS

Array :- Collection of Homogenous data

Address formula

Address of element = Base (A) + W * (K - LB)

K in array → Base Address; W → width

Types of Arrays

1) One dimensional / Single Array
2) Multidimensional Array → 2d (Row / Column) / (Subscript / Row / col)

9 August 2012

FORMULAS

1) Column Major

2d :- $Loc(J,K) = Base(A) + W(E_2L_1 + E_1)$

3d :- $Loc(J,K,M) = Base(A) + W(E_3L_2L_1 + E_2L_1 + E_1)$

4d :- $Loc(J,K,M,N) = Base(A) + W(E_4L_3L_2L_1 + E_3L_2L_1 + E_2L_1 + E_1)$

2) Row Major.

2d :- $= Base(A) + W(E_1L_2 + E_2)$

3d :- $= Base(A) + W(E_1L_2L_3 + E_2L_3 + E_3)$

4d :- $= Base(A) + W(E_1L_2L_3L_4 + E_2L_3L_4 + E_3L_4 + E_4)$

3) Column major

= Base(A) + W[M(K - LB$_2$) + (J - LB$_1$)]

M → No. of rows (ub - lb + 1)

K → lower bond of col

J → lower bond of row

A = (-2:2, 2:22)

Ques Base(A) = 400 W = 4. A(1, 3) = ?

Sol:- A(1,3) = 400 + 4[(2 - (-2) + 1)(3 - 2) + (1 + 2)]

= 432.

4) Row major.

= Base(A) + W[(K - LB$_2$) + N(J - LB$_1$)]

Sol:- A(1,3) = 400 + 4[(3 - 2) + (22 - 2 + 1)(1 + 2)]

= 400 + 4[63 + 1]

= 656.

Ques:- Base (A) = 500, W = 2.

A (1:8, 6:6)

Find A(2, 3)

Sol:- CM :- = 500 + 2[(8 - 1 + 1)(3 - 6) + (2 - 1)]

= 500 + 2[-24 + 1] = 500 + 2(-23)

= 454.

(*) [L$_1$, L$_2$, L$_3$ = ub - lb + 1]

[E$_1$ = J - lb$_{L_1}$] E$_3$ = M - lb$_{L_3}$]

[E$_2$ = K - lb$_{L_2}$.]

Ques A (3:30,

Base (A) =

Sol:- L$_1$ = 30 - 3

L$_2$ = 15 - (

L$_3$ = 20 - 1

∴ CM

= 1000 +

= 1000 +

= 3528

Ques A : (2:8,

A

Sol:- L$_1$ = 8 - 2 +

L$_2$ = 1 - (-

L$_3$ = 10 - 6

= 900

= 1900

= 200

= 632

Ques A (3:30, -5:15, 10:20)
Base (A) = 1000, w=4. Find A(5,10,15)

Sol:- $L_1 = 30-3+1 = 28$ $E_1 = 5-3 = 2$

$L_2 = 15-(-5)+1 = 21$ $E_2 = 10-(-5) = 15$

$L_3 = 20-10+1 = 11.$ $E_3 = 15-(10) = 5.$

$\therefore$ CM

$= 1000 + 4[2\times21\times11 + 15\times11 + 5]$

$= 1000 + 2528$

$= 3528.$

Ques A: (2:8, -4:1, 6:10) Base (A) = 200 w=4

A: (5, -1, 8) = ? (CM)

Sol:- $L_1 = 8-2+1 = 7$ $E_1 = 3$ $E_2 = 3$ $E_3 = 2$

$L_2 = 1-(-4)+1 = 6$

$L_3 = 10-6+1 = 5.$

$= 200 + 4[2\times6\times7 + 3\times7 + 3]$

$= 200 + 4[84 + 21 + 3]$

$= 200 + 4[108]$

$= 632$

NOTES

ARRAYS

- Array is a linear data structure. It is a collection of elements of similar data types.
- It is a list of a finite number n of homogenous data element (i.e. data elements of same type) is called array
- The element of the array are referenced respectively by an index set consisting of consecutive numbers.
- The elements of array are stored respectively in successive memory locations.

TYPES OF ARRAYS.

1) Single dimensional array (1D)
2) Multidimensional array (2D) (3D) (N^{th}D)

SINGLE DIMENSIONAL ARRAY

Syntax:
Data type name of array [size]
Eg:- int A[10], char S[3], float F[3] etc.

LENGTH OF ARRAY.

length = upper bond − lower bond + 1.

REPRESENTATION OF LINEAR ARRAY IN MEMORY

A[0]	A[1]	A[2]	A[3]	A[4]	A[5]
11	12	13	14	15	16
202	204	206	208	210	212

W = Number of element

CALCULATION OF

Computer do address of need to keep first element

Loc(A[K])

where Base

W → No

LB → Lo

A → Ar

OPERATION ON

1) TRANSVERSING element fro Transverse very easy a

Algo:-

Transverse

A → Array t

LB → Lower

UB → Upper

Process → Ope

a) Initalize Co

b) Repeat step

c) [Visit Elem

ctore. It is a
ar data

n of homo-
elements of

referenced
consisting of

ed respecti-
cations.

D)
(3D) (N^mD)

ze]
[3] etc.

d+1.

N MEMORY

Date:

W = Number of word/bytes allotated for each element

CALCULATION OF ADDRESS.

Computer doesn't need to keep track of the address of every element of array but need to keep track only of address of first element of array i.e. Base Address

$$Loc(A[K]) = Base(A) + W*(K-LB)$$

where Base(A) → address of 1st element of array
W → No. of words allocated for each ele.
LB → Lower bond loc → Location
A → Array Name.

OPERATION ON SINGLE DIMENSIONAL ARRAY

1) TRANSVERSING:- Accessing or processing each element from given array exactly once. Transverse in linear data structure is very easy as compared to non linear DS.

Algo:-
Transverse (A, LB, UB)
A → Array to be transversed
LB → Lower bound
UB → Upper bound.
Process → operation to be performed.
a) [Initalize Counter] let K = LB.
b) Repeat steps © & ⓓ while K ≤ UB.
c) [Visit element] Apply process to A[K]

d) [Increase counter] K = K+1
e) Exit.

2) INSERTION :-
- Inserting an element at end of a linear array can be easily done if memory space is available in array
- It is difficult to add or insert elemen in middle of array because it involves the overhead of shifting the elements downwards.

Algo :-
Insert (A, N, K, item)
A → Array to be manipulated.
N → No. of element
K → location of new element
item → new element
a) [Initalize counter] set J = N-1.
b) Repeat © & d) while J >= K
c) [Move element downwards] set A[J+1] = A[J]
d) [Decrease counter] Set J = J-1.
[End of step 2 loop]
e) [Insert Element] Set A[K] = item.
f) [Reset N] set N = N+1.
g) Exit.

3.) DELETION :-
Algo :-

Delete(A,
A → Arra
N → No. o
K → Inde
item = A
a) Set ite
b) Repeat
c) [Move
set LA[J
[End of
d) [Reset n
Set N =
e) Exit.

GENERAL M
An n-dim
B is a c
elements
specified
$K_1, K_2 \ldots K_n$
$0 \leq K_i <$

Represen
Row major : In this
to last

Column major : In th
rapidl

Delete(A, N, K, item)

A → Array to be manipulate

N → No. of element

K → Index of element to be drawn with / deleted

item = A[K]

a) Set item = LA[K]

b) Repeat for J=K to N-1

c) [move J to lst element upward]

set LA[J] = LA[J+1]

[End of loop]

d) [Reset number N of element]

Set N=N-1

e) Exit.

GENERAL MULTIDIMENSIONAL ARRAY

An n-dimensional $m_1 \times m_2 \times \dots \times m_n$ array B is a collection of $m_1, m_2 \dots m_n$ data elements in which each element is specified by a list of n integer... such as $K_1 \cdot K_2 \dots K_n$ called subscripts with property

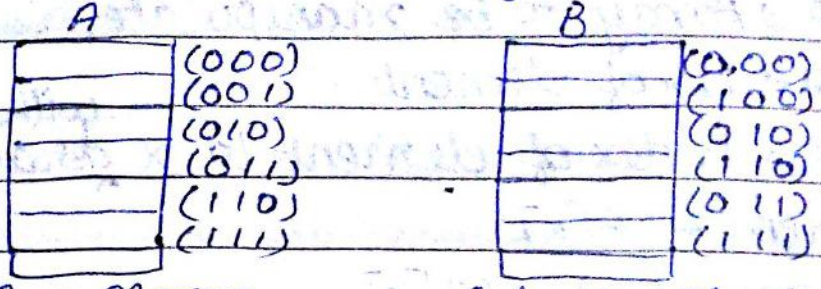
$0 \leq K_1 < m_1$, $0 \leq K_2 < m_2$ ---- $1 \leq K_n < m_n$

Representation of array in memory

Row Major:- In this last subscript vary first, the next to last subscript vary second & so on.

Column Major:- In this first subscript vary first (most rapidly), the next subscript varies

second (less rapidly) and so on.



Row Major Column Major

COLUMN MAJOR GENERAL FORMULA

$$Loc[K_1, K_2, K_3 \cdots K_n] = Base(C) + W[(((\cdots(E_n L_{n-1} + E_{N-1})L_{N-2}) + \cdots E_3)L_2 + E_2)L_1 + E_1]$$

↳ Effective Index

where $E_i = K_i$ - lower bound.

TWO DIMENSIONAL ARRAY

A two dimensional m×n array A is a collection of m·n data elements such that each element is specified by a pair of integers (such as J, K) called subscripts, with the property that $1 \le J \le m$ and $1 \le K \le n$ A[m][n]

Row Column

Element of A with first subscript j and second subscript K will be denoted by $A_{J,K}$ or A[J,K]

→ A [0,0 0,1
1,0 1,1
Column
second sub

→ The pair of

→ Non Regula
array w

→ Regular
array w

→ The length
number
obtained f
length =

REPRESENTA

i) Column

ii) Row ma

A [0,0 0,1
1,0 1,1
2,0 2,1

A

Column

→ $A\begin{bmatrix} 0,0 & 0,1 \\ 1,0 & 1,1 \end{bmatrix}$ Row contains those elements with the same 1st subscripts

Column contains those elements with same second subscripts.

→ The pair of length m×n is called size of array

→ Non Regular Array:- Those multidimensional array whose lower bound are not 0 or 1.

→ Regular Array:- Those multidimensional array whose lower bound are 0 or 1.

→ The length of given dimension (i.e. the number of integers in its index set) can be obtained from formula

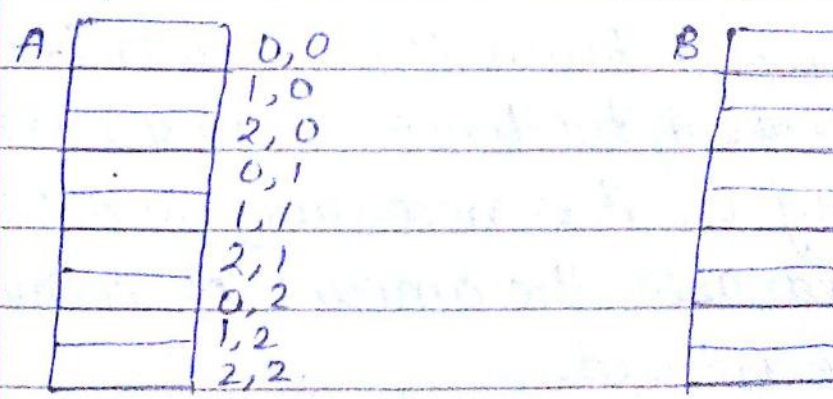
Length = upper bound − lower bound + 1

REPRESENTATION OF 2D ARRAY IN MEMORY

i) Column major order (column by column)

ii) Row major order (row by row)

$$A\begin{bmatrix} 0,0 & 0,1 & 0,2 \\ 1,0 & 1,1 & 1,2 \\ 2,0 & 2,1 & 2,2 \end{bmatrix}$$



Column Major Row major

CALCULATION OF ADDRESS OF 2D ARRAY

i) Row Major Order:

$$Loc(A[J,K]) = Base(A) + W[N(J - LB) + (K - LB)]$$

(J: No. of Row, K: No. of col, N: No. of column)

ii) Column Major Order:

$$Loc(A[J,K]) = Base(A) + W[M(K - LB) + (J - LB)]$$

(K: No. of col, J: No. of Row, M: No. of Rows)

APPLICATION OF LINEAR ARRAY.

1) These are used to represent all sort of lists
2) These are also used to represent other data structures such as stacks, queue heaps etc.

ADVANTAGES OF ARRAYS

Array makes whole task of organising & manipulating data easier & more efficient

DISADVANTAGE OF ARRAYS

1) The prior knowledge of number of elements in the linear array is necessary i.e. it is necessary to declare in advance the amount of memory to be utilized.
2) Array are static structure. static mean that whether memory is allocated at

compilation.
used by a
extended
3) The insertion
are time.

11 August 2012

LINEAR SEAR

1) Data[N+1]
2) Set Loc = 1
3) Repeat wh
Set Loc=Loc
4) If Loc = N+
5) exit.

BINARY SEA

1) Set BEG=LB
2) Repeat Step
[mid] + IT
3) If ITEM <
Else set
4) Set MID =
5) If Data[
Else Set
6) Exit.

compilation time or run time, thus memory used by arrays cannot be reduced or extended.

3) The insertion and deletion in these arrays are time consuming.

11 August 2012

LINEAR SEARCH (ALGO)

1) Data[N+1] = ITEM.
2) Set LOC = 1
3) Repeat while Data[LOC] ≠ ITEM.
   Set LOC = LOC + 1
4) If LOC = N+1 then set LOC = 0
5) exit.

BINARY SEARCH (ALGO)

1) Set BEG = LB, END = UB and mid = INT(BEG + END/2)
2) Repeat Step 3 & 4 while BEG ≤ END and DATA [mid] ≠ ITEM
3) If ITEM < Data[MID], then set END = MID - 1.
   Else set BEG = MID + 1
4) Set MID = INT(BEG + END/2)
5) If Data[mid] = ITEM, then set LOC = MID
   Else set LOC = NULL
6) Exit.

SPARSE MATRIX.

Values of matrix is zero. Memory wastage

four types

1) Diagonal SM
2) Tridiagonal SM
3) Triangular SM
4) Irregular SM

Diagonal SM

$$\begin{bmatrix} 1 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 3 \end{bmatrix}_{3\times3}$$

A[1,1] [2,2] [3,3] are non zero

No. of non-zero element = n = 3

zero elements = n(n-1) = 6

No. of non zero elements before particular location = Index number − LB

= J − LB

Eg:- A : [I, J] (I = K) LB = 0 ∴ before 2 nonzero

A[1,1] = 1 − 0 = 1

Tridiagonal SM.

Half of matrix (above diagonal/ below diagonal) are zero.

Eg:- Lo

No. o

No. o

No. of
inclu

if L

These
For abo

Trian
Diago

Eg:- Lower Tridiagonal sm

$$\begin{bmatrix} 10 & 0 & 0 & 0 & 0 \\ 2 & 3 & 0 & 0 & 0 \\ 4 & 5 & 6 & 0 & 0 \\ 7 & 8 & 9 & 10 & 0 \\ 10 & 12 & 13 & 14 & 15 \end{bmatrix}$$

No. of zero $= \frac{n(n-1)}{2}$

No. of non zero $= \frac{n(n+1)}{2}$

No. of non zero elements above and including $A[I,K] = \frac{I(I-1)}{2} + K$. when LB = 1

if LB = 0 than $\Rightarrow \frac{I(I+1)}{2} + (K+1)$

These matrices are converted to 1-D matrix.
For above example.

Row (R)	Column (C)	Element
0	0	1
1	0	2
1	1	3
⋮	⋮	⋮
5	4	14
5	5	15

Triangular sm

Diagonal along ±1 elements are non zero.

Date:

16 Aug

$$\begin{bmatrix} x & x & 0 & 0 & 0 \\ x & x & x & 0 & 0 \\ 0 & x & x & x & 0 \\ 0 & 0 & x & x & x \\ 0 & 0 & 0 & x & x \end{bmatrix}$$

No. of Non zero elements = $3n-1$.

above & including $A[J,K]$ LB=0 $\downarrow$

$2J+K+1$

if LB=1 $\Rightarrow$ $2J+K-2$.

1.
2.
3.
4
5

1.
2
3
6

LINKED LISTS.

Collection of nodes.

Traversing:-

1. Set PTR = START.
2. Repeat step 3 & 4 while PTR ≠ NULL.
3. Apply PROCESS to INFO[PTR]
4. Set PTR = LINK[PTR] [End of step 2 loop]
5. Exit.

Searching (unsorted)

1. Set PTR = START.
2. Repeat step 3 while PTR ≠ NULL
3. If ITEM = INFO[PTR] then set LOC = PTR and Exit
   Else PTR = LINK[PTR] (End of step 2)
4. Set LOC = NULL
5. Exit.

for sorted ITEM < INFO[PTR] => LOC = NULL.

Garbage collection :- collection of free nodes
Avail link list :- list having empty nodes
Start link list :- list having nodes along with data

(A) INSERTION AT START

1. If AVAIL = NULL than write overflow and Return
2. Set NEW = AVAIL and AVAIL = LINK[NEW]
3. Set INFO[NEW] = ITEM.
4. Set LINK[NEW] = START
5. Set START = NEW
6. Exit.

(B) INSERTION AT MID

1. If AVAIL = NULL and write overflow and return
2. Set NEW = AVAIL and AVAIL = LINK[NEW]
3. Set INFO[NEW] = ITEM.
4. If LOC = NULL, than
   Set LINK[NEW] = NULL
   and START = NEW
   Else
   Set LINK[NEW] = LINK[LOC]
   and LINK[LOC] = NEW
5. Exit

(C) INSERTION AT END

1. If AVAIL = NULL than write overflow and exit (return)
2. Set NEW = AVAIL and AVAIL = LINK[NEW]
3. Set INF
4. If Repe
5. PTR
6. LINK
7. Exit.

(D) DELETION

1. If LOCP=
   Set ST
   Else.
   Set L
2. Set LIN
3. Exit

18 August 2012

(E) SEARCHI

1. If STA
   set
2. If IN
   Set
3. Save =
4. Repeat
5. If IN
   LOCP
6. Set
7. Set

3. Set INFO[NEW] = ITEM.
4. Repeat from PTR = START and PTR = NULL
5. PTR = LINK[PTR]
LINK[PTR] = NEW.
6. LINK[NEW] = NULL
7. Exit.

(D) DELETION.

1. If LOCP = NULL then
   Set START = LINK[START]
   Else.
   Set LINK[LOCP] = LINK[LOC] & AVAIL = LOC.
2. Set LINK[LOC] = AVAIL and AVAIL = LOC.
3. Exit.

12 August 2012

(E) SEARCHING AND DELETING ELEMENT IN A LINK LIST

1) If START = NULL then
   Set LOC = NULL and LOCP = NULL
2) If INFO[START] = ITEM, then
   Set LOC = START and LOCP = NULL and Return
3) Save = START and PTR = LINK[PTR]
4) Repeat Step 5 & 6 while PTR ≠ NULL.
5) If INFO[PTR] = ITEM then LOC = PTR and
   LOCP = Save and return
6) Set SAVE = PTR and PTR = LINK[PTR]
7) Set LOC = NULL
8) Exit.

24 Aug 2012

~~Same as previous steps 1 to 7~~

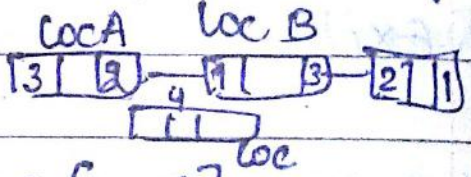
b) If LOC = NULL, then ITEM not in list

c) If LOCP = NULL then
   Set START = LINK[START]
   Else set LINK[LOCP] = LINK[LOC]

d) Set LINK[LOC] = AVAIL
   and AVAIL = LOC

e) Exit

Doubly Link list, a node contain address of both, address of previous & next element.



Insertion

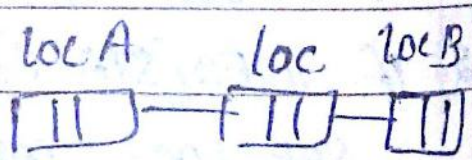
Forw[LOC] = Forw[LOCA] / LOC B

Forw[LOC A] = LOC

Backw[LOC] = Backw[LOC B] / LOC A.

Backw[LOC B] = LOC.

Deletion



Forw[LOC A] = LOC B

Backw[LOC B] = LOC A

(F) Inse
1. If
2. Set
3. Se
4. S
5.
6.
7. S
8. E

at
1.
2., 3.,
4.

24 Aug 2012

CIRCULAR LINK LIST

Replace till PTR = NULL by till PTR = START in singly link list

and for doubly list.

(F) Insertion at start

1. If AVAIL = NULL than write overflow & return
2. Set NEW = AVAIL and AVAIL = LINK[NEW]
3. Set INFO[NEW] = ITEM
4. Set LINK[NEW] = START
5. Set START = NEW
6. PTR = START to LINK[PTR] = START. and PTR = LINK[PTR]
7. Set LINK[PTR] = NEW.
8. Exit

at Mid

1. Same as in (B)
2, 3, 4, " " " "
5. If loc = Same as in Step 6.
6. Exit

at End

1,2,3. - (C)

4. PTR = Start to LINK[PTR] = START
   and PTR = LINK[PTR]
5. LINK[PTR] = NEW
6. LINK[NEW] = START
7. Exit

Deletion

1. Same (a) in (E), step (4) LINK [PTR] ≠ Start
2. Same in (E) (6 step)

at Start

than Start = link (Start)
PTR = Start to LINK [PTR] = Start
    PTR = LINK(PTR)
  & LINK[PTR] = Link [Start]

link [Start] = Avail
   Avail = Start

Doubly

Insertion

1,2,3 (A)
4. Set Forw[NEW] = Forw [START]
   Forw[START] = NEW

Back [New] = Back [start]

Ptr = back [start]

forw [Ptr] = New

Back [New] = Ptr

Back [start] = New

start = new

(2) At Mid :-

i) LOC → is position after which insertion is carried out, passed as argument i.e. given by user

ii) iii) Same as above

iv) LOCP = FORW[LOC]

v) Set FORW[LOC] = NEW

vi) BACK[NEW] = LOC

vii) Set FORW[NEW] = LOCP

viii) Set BACKW[LOCP] = NEW

ix) Exit

(3) At End :- LOC → last element after which insertion to done

i) ii) iii) Same as above

iv) FORW[LOC] = NEW

v) BACKW[NEW] = LOC

vi) FORW[NEW] = NULL

vii) Exit

(Deletion) ⟶ LOC → To be deleted
LOCA → Before LOC
LOCB → After LOC

i) At Start

1. LOC = START
2. Now LOCB = FORW[LOC]
3. BACKW[LOCB] = NULL
4. START = LOCB
5. BACKW[LOC] = NULL, BACKW[AVAIL] = LOC
6. FORW[LOC] = AVAIL, AVAIL = LOC
7. Exit

ii) At Mid

LOC is

1. LOCB =
2. FORW[
3. 4. Steps
5. Exit

iii) At End

1. for PTR
2. PTR = LO
3. LOCA = A
4. FORW[L
5. 6. Steps
7. Exit

2 WAY C

Insertio

(i) At beg

i) If AV

ii) NEW

iii) BAC

iv) FOR

v) INFO

BACK

vi) LOC

vii) A

viii) A

ii) At Mid

LOC is given by user or found using search algo.

1. LOCB = FORW[LOC] , LOCA = BACKW[LOC]
2. FORW[LOCA] = LOCB, BACKW[LOCB] = LOCA
3. 4. Steps 5 & 6 of above algo.
5. Exit

iii) At End

1. for PTR = START to LINK[PTR] = NULL and PTR = LINK[PTR]
2. PTR = LOC
3. LOCA = BACKW[LOC]
4. FORW[LOCA] = NULL
5. 6. Steps 5 & 6 of algo - deletion at start
7. Exit

2 WAY CIRCULAR

Insertion

i) At beg

i) If AVAIL = NULL, overflow & return
ii) NEW = AVAIL and AVAIL = FORW[NEW]
iii) BACK[FORW[NEW]] = BACK[NEW]
iv) FORW[BACK[NEW]] = FORW[NEW]
v) INFO[NEW] = ITEM, FORW[NEW] = FORW[START], BACK[NEW] = BACK[START]
vi) LOCA = BACK[START], LOCB = FORW[START]
vii) FORW(START) = NEW, BACK[LOCB] = NEW
~~viii)~~ FORW[LOCA] = NEW
viii) Exit

② At Mid

i) ii) iii) iv) Same

v) LOCB = FORW[LOCB], INFO[NEW] = ITEM.
FORW[NEW] = LOCB, BACK[NEW] = LOCA.

vi) FORW[LOCA] = NEW, BACK[LOCB] = NEW

vii) Exit.

③ At End

i) ii) iii) iv) - Same.

v) LOCB = FORW[START] LOCA = BACK[START]

vi) BACK[START] = NEW, BACK[LOCB] = NEW,
FORW[LOCA] = NEW,

vii) BACK[NEW] = LOCA, FORW[NEW] = LOCB

viii) Exit.

Deletion

① At beg.

i) LOC = FORW[START]

ii) LOCA = BACK[LOC], LOCB = FORW[LOC]

iii) FORW[START] = LOCB

iv) BACK[LOCB] = LOCA.

v) FORW[LOCA] = LOCB

vi) FORW[LOC] = FORW[AVAIL]
BACK[LOC] = BACK[FORW[AVAIL]]
BACK[FORW[AVAIL]] = LOC
FORW[BACK[AVAIL]] = LOC

vii) Exit.

② At Mid.

i) 2nd, 4

ii) Exit

③ At End

i) LOC = B

ii) 2nd

iii) BACK(

iv) 4th,

vi) Exit.

3 Aug 2012

HEADER

similar

The 1st n

an infor

link lis

operatio

(2) At Mid.

i) 2nd, 4th, 5th step of above.

ii) Exit

(3) At End

i) LOC = BACK[START]

ii) 2nd step of del. at beg

iii) BACK[START] = LOCB

iv) 4th, 5th step of del. at beg

vi) Exit.

28 Aug 2012

HEADER LINK LIST

Similar to one way link list

The 1st node only contains address not only data or information by user but contains info. about link list. We don't use header node for any operation.

STACK & QUEUE

STACK :-

Stack is a non primitive data structure where insertion and deletion take place at top position.
Stack is also known as LIFO
It is also known as push down list
→ Push means insertion
→ Pop means deletion
→ maxsize indicate max. limit
If TOP = maxsize ⇒ overflow
If TOP = Null / 0 ⇒ underflow

Implementation of Stack

⇒ Stack is implemented in 2 forms
i) Static implemention (using Arrays)
ii) Dynamic " (using linked

Dynamic ~~Static~~ Implementation

Algo for stack from arrays is similar to that of link list. Only difference is that start is namely top. and link is changed by stack and concept of AVAIL is ~~not~~ present

Static ~~Dynamic~~ Implementation

Insertion

1) If TOP = maxsize, overflow
2) TOP = TOP + 1
3) STACK [TOP] = ITEM
4) Exit

Deletion

1) If TOP = 0,
2) Set ITEM =
3) TOP = TOP -
4) EXIT

POLISH NOTA

1) Add right
2) Scan P le
step of eac
is encount
3) If operan
stack
4) If operato
operand f
next to t
b) Evalute
c) Place
5) Set va
6) Exit

POSTFIX – E