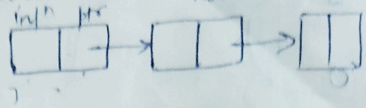
Length / size of array = The no. of elements in array

= UB - LB + 1

(n - 1 + 1 = n 1, 2 ... n
n - 0 + 1 = n + 1 0, 1, 2 ... n)

There are 3 nodes linked with each other.

Linked List :-



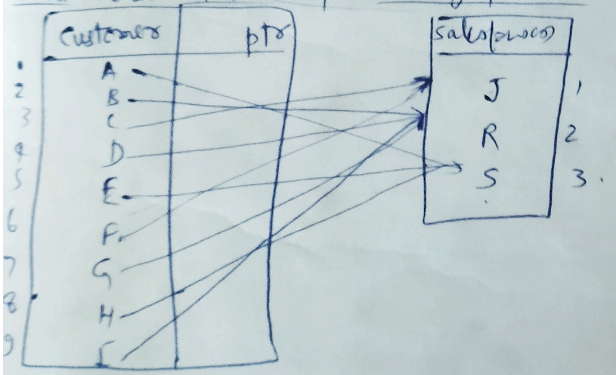
list → cust. [A, B, C, D ...]
→ salesperson = [S, R, J]

Linked list → provides method to link these 2 lists

"A linked list is a collection of nodes. A node has 2 fields < info / address. Address field contains address of next node i.e. points to next node

	Customer	Salesperson
1	A	S
2	B	R
3	C	J
4	D	R
	E	S
5	F	J
6	G	R
7	H	S
8, 9	I	R

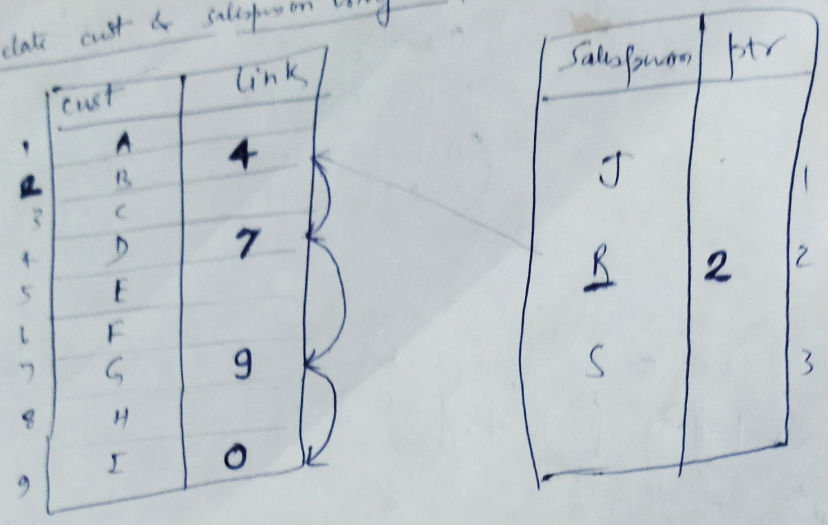
Relate customer & salesperson using pointers :—



≡

Salesperson	ptr
J	3, 6
R	2 4, 7, 9
S	1, 5, 8

Relate cust & salesperson using link -



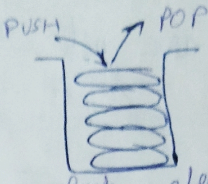
Similar case for J & S ——— ; start from 2
R - 2, 4, 7, 9 start from 3
J - 3, 6 start from 1
S - 1, 5, 8

Points - when an element in one list points to an element in different list

Link - ——— same list

stack : LIFO (It is a LIFO structure type of DS)

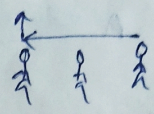
Ex. stack of dishes



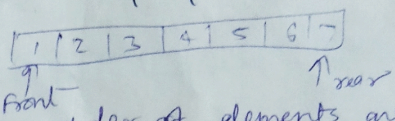
A stack is a DS in which elements are inserted & deleted at one end which is the top of the stack

Queue : FIFO (It follow FIFO type of structure)

Ex. ① A Queue waiting for tickets



② A Que with 7 elements

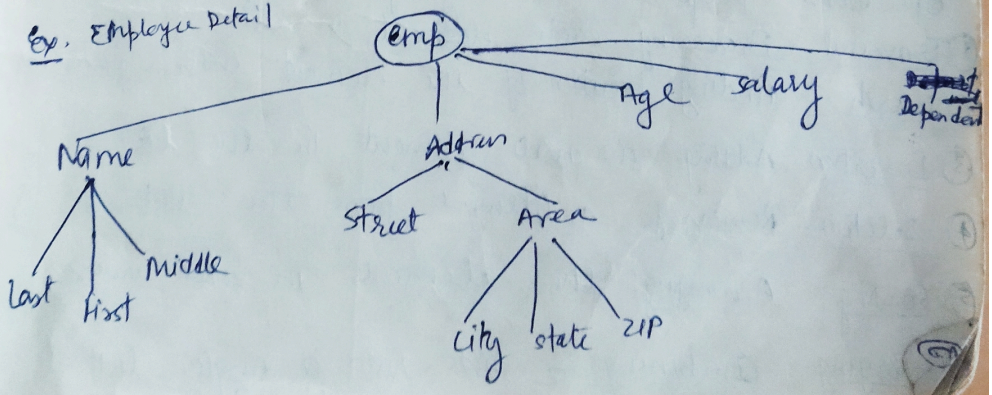


A Queue is a DS in which the order of elements are moved to the rear and removed from the front of the list

Tree :

Data frequently contain a hierarchical relationship b/w various elements. The ~~ds~~ reflecting this relationship is called a ~~Tree~~ Rooted Tree Graph or a Tree

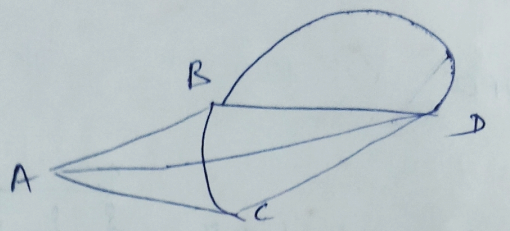
Ex. Employee Detail



Graph :-

Data sometimes contains a relationship b/w pairs of elements which are not necessarily hierarchical in nature

Ex. Airline flights



Structure operations :-

Operations performed on any of the structure are as below

1) Traversal - Processing each element in list
2) search - finding location of the element with a given key
3) Insertion - Adding a new element in the list
4) Deletion - Removing a element from the list
5) sorting - Arranging the elements in some order
6) Merging - Combining 2 lists into a single list

Algorithm for DS :-

study of a DS is a study of algorithms that control them.

Algo is a collection of statements that can be used for programming by any language.

* An algo is a medium b/w a natural language & a pgmg language.

→ Algo must be expressed in a fashion that is completely free of ambiguity. *

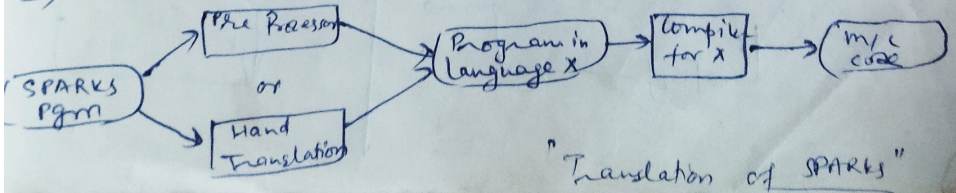
→ Algo should be efficient.
They should not unnecessarily use memory location.
They should not require an excessive no: of logical operations

→ Algo should be concise & compact to facilitate verification of their correctness.
Several test cases are used for verificat" of correctness.

Algo is also known as SPARKS language. ~~is also~~ :

→ Smart Programmers Are Required to Know SPARKS

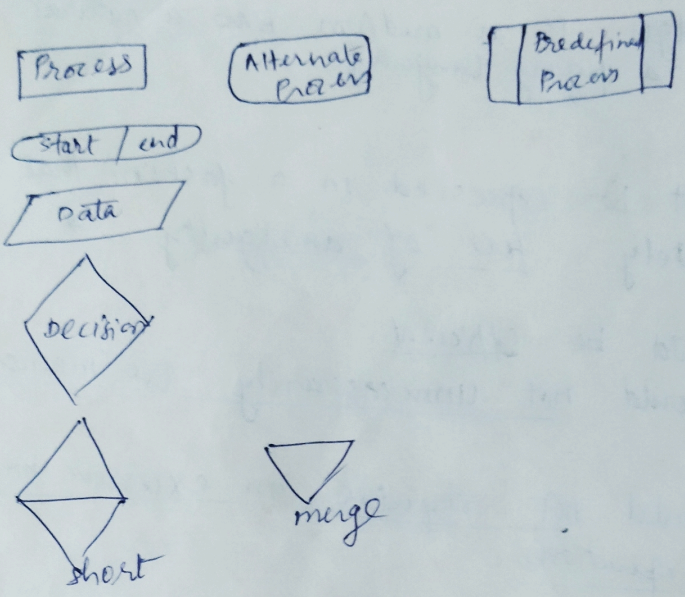
→ Structured Programming : A Reasonably komplete set



"Translation of SPARKS"

→ Procedure should carefully specify its i/p - o/p variable
→ Documentation should be short & meaningful.
→ Use subroutines.

Notations used for creating a flow chart :—

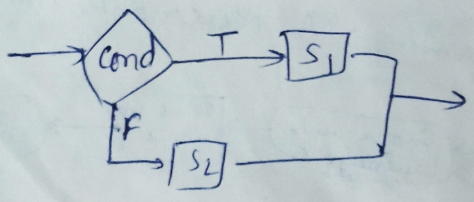


Things to know before going to Algorithms

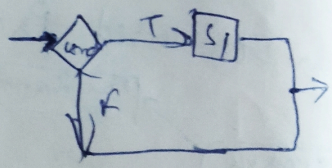
Assignment : variable ← expression ("var := exp" Schaum's)

Conditional statements / Iteration statements :

① if cond then S_1 else S_2

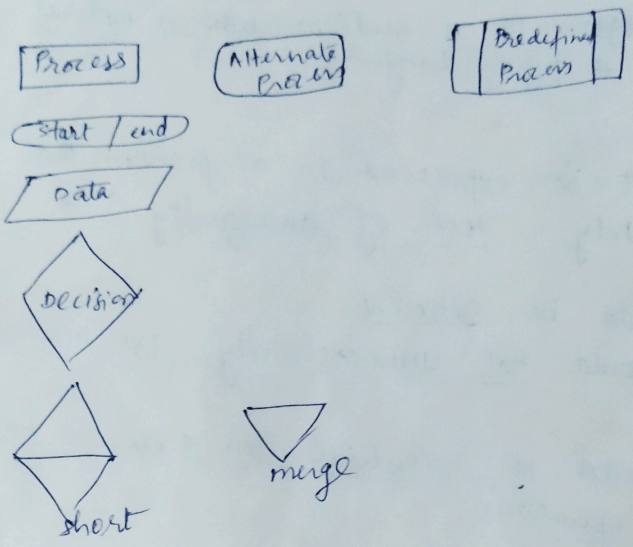


if cond then S



→ Procedure should carefully specify its i/p - o/p variable
→ Documentation should be short & meaningful.
→ Use subroutines.

Notations used for creating a flow chart :—

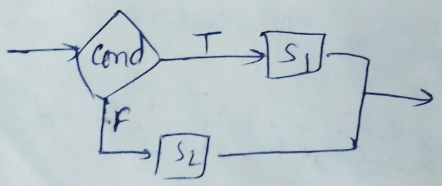


Things to know before going to Algorithms

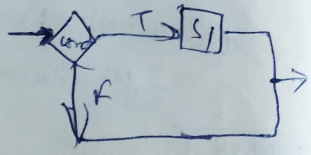
Assignment : variable ← expression ("var := exp" Schaum's)

conditional statements / Iteration statements.

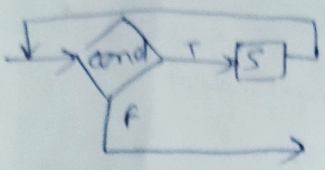
① if cond then S_1 else S_2



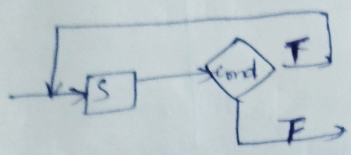
if cond then S_1



② while cond do
S
end

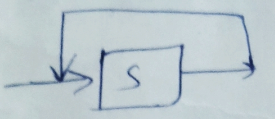


③ repeat
S
until cond



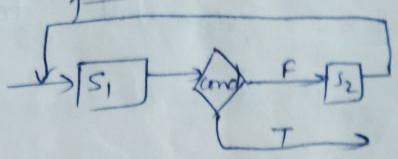
* (S will be executed at least once)

④ loop
S
forever



** (infinite loop)

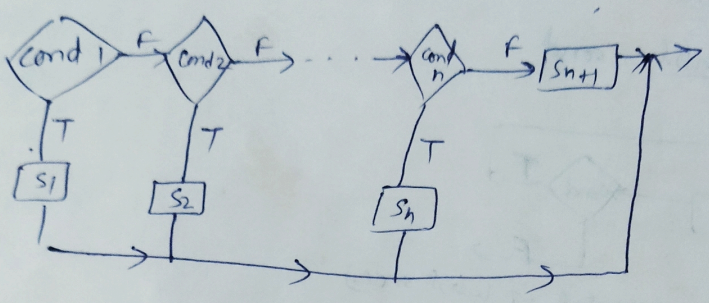
loop
S_1
** if cond then exit
S_2
forever



** (To avoid infinite loop we can also use "goto" "label" at the place of "exit".

⑤ case
: cond 1 : S_1
: cond 2 : S_2
⋮
: cond n : S_n
: else : S_{n+1} // optional clause

end



⑥ ~~while start~~

for vble ← start to finish by increment, do
S
end

Complete SPARKS procedure has the form:

Procedure NAME (parameter list)
S
end

→ A procedure can be used as a fn by using the stmt
return (expr)
The execution of an end implies a return at the end of procedure

→ A procedure may be invoked by using a call stmt
call NAME (parameter list)

→ For input/output we assume the existence of 2 fns.
read (argument list)
print (argument list)

Read: var name
Write: message/var name (schaums)

→ comments may appear anywhere on a line enclosed by double slashes. Ex:
// this is a comment //

→ All the variables are assumed to be of type INTEGER unless stated.

→ SPARKS language is rich enough so that one can create a good looking pgm by applying some simple rules —

① Every procedure should carefully specify its i/p and o/p variables

② The meaning of variables should be defined.

③ The flow of the pgm should generally be forward except for normal looping or unavoidable instances.

④ Indentation rules should be established & followed so that computational units of pgm text can more easily be identified

⑤ Documentation should be short, but meaningful Avoid sentences like "i is increased by one".

⑥ Use subroutines where appropriate

How to create pgms

1. Requirements
2. Design
3. Analysis (compare which algo is better etc)
4. Refinement & coding
5. Verification

Algorithm –

An algo is a finite set of instructions which, if followed, accomplish a particular task

Every algo must satisfy following things –

① i/p - (0 or more) quantity should be supplied ext. i/p
② o/p - (at least one)
③ definiteness - (each inst^n should be clear & unambiguous)
④ finiteness - (while tracing out the inst^{ns} for all cases the algo will terminate after a finite no. of steps)
⑤ effectiveness - (every inst^n must be sufficiently basic that it can be understood by a person using only pencil & paper)

Note - Algo & its associated DS form a pgm. (Algo + DS = pgm)

Page 22 Schaum's

Algorithms : Complexity, time-space trade-off

An algo should be efficient which can be decided by time & space used by the algo execution.

The complexity of an algo is the function which gives the running time &/or space in terms of the i/p size

Each of algo uses a particular DS. Choice of DS depends on many things like, type of data & frequency of the operations applied so we may not always use the most efficient algo.

Sometimes the choice of DS involves a time-space trade off.

"By increasing the amount of space for storing the data, one may be able to reduce the time needed for processing the data, or vice versa."

Thus this tradeoff of space for time (more space comes to reduce the time complexity) is not worth the expense (memory location method). So "hashing fn." is used as its alternative

→ Time complexity: no of key operations (in sorting/searching algo)
(running time) or no of comparisons

~~Space~~

→ Space complexity: max of memory needed by algo.
(storage space)

→ Unless stated or implied, the term "complexity" shall refer to the running time of algo.

→ complexity { worst case / Avg case

The complexity of the avg case of an algo is usually much more complicated to analyze than that of the worst case

---

→ unless stated or implied, the complexity of an algo shall mean the function which gives the running time of the worst case in terms of i/p size

---

Note complexity of avg case for many algos is proportional to the worst case.

Big O Notation (capital O)

complexity $f(n)$ can be any of the following-
$(\log_2 n, n\log_2 n, n, n^2, n^3, 2^n)$
where n is the size of the i/p data

Depends on Rate of growth of the curve formed by passing the values in algo

The complexity of certain well known searching & sorting algo's is given below —

1. Linear search : $O(n)$
2. Binary : $O(\log_2 n)$
3. Bubble sort : $O(n^2)$
4. Merge sort : $O(n\log_2 n)$

time complexity

n \ f(n)	log n	n log n	n	n^2	n^3	2^n
5	3	15	5	25	125	32
10	4	40	10	100	10^3	~~10^3~~ 2^{10}

→ Constant time algo's — $O(1)$
→ Logarithmic time algo's — $O(\log n)$
→ Linear time algo's — $O(n)$
→ Polynomial time algo's — $O(n^k)$ for $k \geq 1$
→ Exponential time algo's — $O(k^n)$ for $k > 1$

Note- Many algo's are $O(n \log n)$

Arrays

(no bound checking concept in 'C' it is upto pgmr to check the UB)

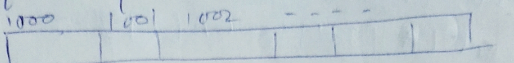
→ Array is a consecutive set of memory locations (not always)

Definition :→ It is a list of finite no. of similar data elements

1 Darray / Linear Array

Representation of linear array in memory :-

Memory = sequence of addressed locations

= 

Address calculation

LOC (A[K]) = addres of element A[K] of the array A

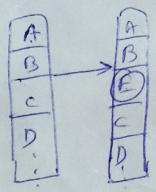
Base (A) = Base address of A (Array name is actually a pointer allocated to the name of the array) to the 1st locat^n of memory block

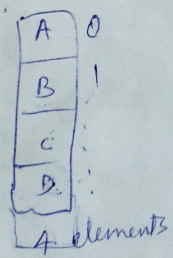
$$LOC(A[K]) = Base(A) + w(K - LB)$$

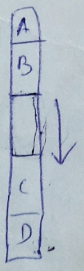
w = no of words per memory cell of array

Analysis regarding Insertion & deletion -

Insertion







Create space to insert E

(Move C, D downward by 1 position)

5 elements

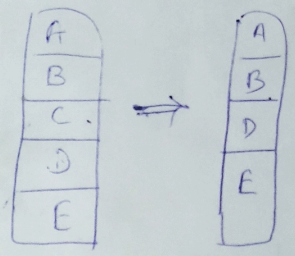
(Insert E & increment the array length by 1)

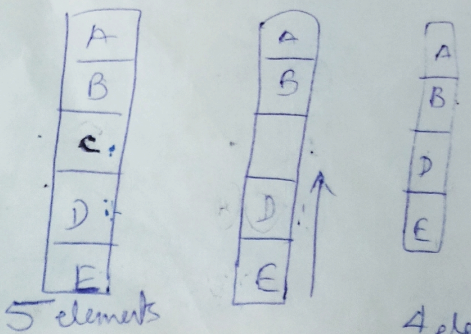
INSERT (LA, N, K, ITEM)

LA – Linear array with N elements
K – Positive integer such that K≤N
ITEM is inserted into K^{th} position in LA

1. set J := N-1 // Initialize counter
2. Repeat 3 & 4 while J ≥ K
3. set LA[J+1] := LA[J] // Move J^{th} element downward
4. set J = J-1 // Decrease counter / end of loop
5. set LA[K] := ITEM // Insert element
6. set N := N+1 // Reset N
7. Exit

Deletion





5 elements

4 elements

DELETE (LA, N, K, ITEM)

1. set ITEM := LA[K]
2. Repeat for J = K to N-1.
3. set LA[J] := LA[J+1] // (Move J+1st element upward) end of loop
4. N := N-1 // Reset the no N of elements in LA
5. Exit

Traversal

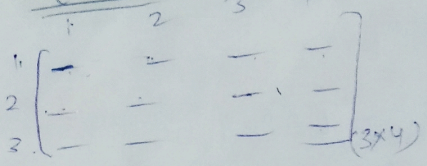
TRAVERSE (LA, LB, UB)

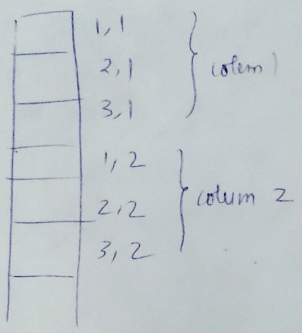
LA - Linear array
LB - lower bound
UB - upper bound

1. Repeat for K = LB to UB
   Apply PROCESS to LA[K]
   (End of loop)
2. Exit

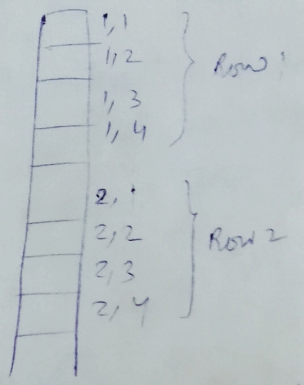
Multidimensional Array

2D Array :





Column-major order



Row-major order

Matrix Programs

Traversal

```
for (i = 1 to n)
  {
    for (j = 1 to m)
      printf ("%A[i][j]);
  }
```

Matrix Addition

```
for (i = 1 to n)
  {
    for (j = 1 to m)
      C[i][j] = A[i][j] + B[i][j]
  }
```

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix} + \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \end{bmatrix} = \begin{bmatrix} 8 & 10 & 12 \\ 14 & 16 & 18 \end{bmatrix}$$

Matrix Multiplication

$$\begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \end{bmatrix}_{2\times 3} \begin{bmatrix} 7 & 8 & 9 \\ 10 & 11 & 12 \\ 13 & 14 & 15 \end{bmatrix}_{3\times 3} = \begin{bmatrix} 1+20+39 & 8+22+42 & \\ & & \\ & & \end{bmatrix}$$

m p p n

Mat(A, B, C, M, P, N)

A - M x P Matrix array
B - P x N Matrix array
C - Product of A & B is stored in this M x N matrix

→ ① Repeat ② to ④ for I = 1 to M
→ ② Repeat ③ to ④ for J = 1 to N
→ ③ set C[I,J] := 0 // initialise C[I,J]
→ ④ Repeat for K = 1 to P
C[I,J] := C[I,J] + A[I,K] * B[K,J]

[end of inner loop]

[end of step ② loop]

[end of step ① loop]

→ ⑤ exit

INITIALIZATION OF ARRAYS :-

Initializing an 1D array -

```
int A[5] = {1, 2, 3, 4, 5};
char A[6] = {'h', 'e', 'l', 'l', 'o', '\0'};
char A[] = "Hello";
```

Initializing a 2D array

```
int A[2][2] = {1, 2, 3, 4};
int A[3][3] = {1, 2, 3, 4, 5, 6, 7, 8, 9};
```

Initializing a 3D array

```
int A[3][3][3] = {
                  ...
                  ...
                  ...
                 };
```

A[1][2][0] = K

A[0][0][0] = 0
A[0][2][2] = 0

Application of arrays –

→ To list some elements of a record.
→ Matrices in Mathematics
→ Tables in Business app^n

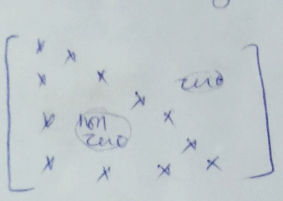
SPARSE Matrices & Vectors –

Matrices with a relatively high proportion of zero entries are called SPARSE Matrices.

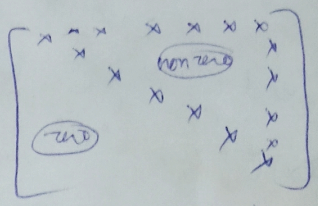
There are 2 types of n square sparse matrix

1) Triangular Matrix –

Non-zero entries can only occur (on) or (below/above) the main diagonal.



lower triangular



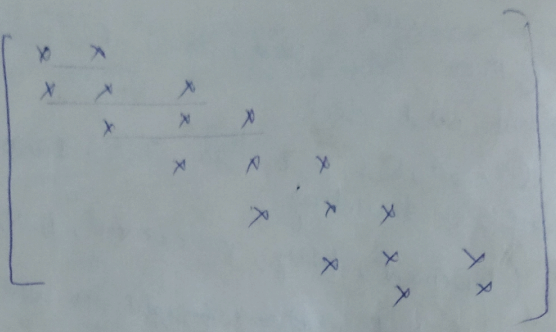
upper triangular

Ex.

$$\begin{bmatrix} 1 & & & \\ 2 & 5 & & \\ 3 & 6 & 8 & \\ 4 & 7 & 9 & 10 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 2 & 5 & 0 & 0 \\ 3 & 6 & 8 & 0 \\ 4 & 7 & 9 & 12 \end{bmatrix}$$

② Tridiagonal Matrix

non-zero entries can only occur on the diagonal or on elements immediately above or below the diagonal.



Ex.

$$\begin{bmatrix} 1 & 2 & & & \\ 3 & 4 & 5 & & \\ & 6 & 7 & 8 & \\ & & 9 & 10 & \end{bmatrix} \equiv \begin{bmatrix} 1 & 2 & 0 & 0 \\ 3 & 4 & 5 & 0 \\ 0 & 6 & 7 & 8 \\ 0 & 0 & 9 & 10 \end{bmatrix}$$

NOTE:- Natural method of representing matrices in memory as 2D array may not be suitable for sparse matrices i.e. one may save space by storing only nonzero entries. (to avoid wastage of memory space which will caused by storing zeros in memory)

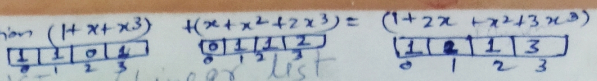
Array Representation of polynomials

$1+x+2x^2+3x^3+4x^4$

1	1	2	3	4	5
0	1	2	3	4	5

5x5

Polynomial Addition $(1+x+x^3)$ $+(x+x^2+2x^3) = (1+2x+x^2+3x^3)$



ORDERED LIST – / Linear List

It is one of the simplest & most commonly found data object.

Ex –

① days of the week
(Mon, Tue, Wed, Thu, Fri, Sat, Sun)

② values in card deck
(2, 3, 4, 5, 6, 7, 8, 9, 10, Jack, Queen, King, Ace)

③ floors of a building
(basement, lobby, mezzanine, first, second, third)

④ Years the United states fought in World war II
(1941, 1942, 1943, 1944, 1945)

It may be empty or can be written as $(a_1, a_2 \cdots a_n)$ where a_i are atoms of some set S.

Difference b/w Array & List :—

Array	List
It is a set consistent of a fixed no. of data items	It is a ordered set consisting of a variable no. of data items
Insertion & deletion operations are performed	can be performed using pointers

Character string in C – (%s) (%c)

"Length of a string = no of characters of the string + 1"

↓ to store the null char ('\0')

A string of char's is stored in successive elements of a char array & terminated by the Null char

char msg[5] = "hello" : means

msg[0] = 'h'
msg[1] = 'e'
msg[2] = 'l'
msg[3] = 'l'
msg[4] = 'o'
msg[5] = '\0' = NULL = 0

ASCII value of NULL is 0

Library functions for String Manipulation –

strcpy – To copy a string, including the null char

string.h

```
char A[10], B[10];
gets(A);
strcpy(B, A);
printf("A: %s and B: %s \n", A, B);
```

strncpy – Similar to strcpy, but it allows the no of char's to be copied.
– If the source is shorter than the destination dest'n is padded with null char upto the length specified.

```
char *strncpy(char *dst, const char *src,
              size_t len);
```

strcat – used to append a src to the end of dest" str

→ char *strcat (char *dst, const char *src)

→ strcat (A, B); // where char A[20] = "; char B[20];

strncat –

char *strncat (char *dst, const char *src, size_t N);

strcmp – To compare 2 strings, it returns no >0 (if string1 > string2), <0 (if string1 < string2), =0

if (strcmp (first, second) == 0)

puts ("The 2 strings are equal");

strncmp – It compares first N char of each string

int strncmp (const char *first, const char *, size_t N);

strlen – It returns the length of the string (not counting the null char)

→ size_t strlen (const char *str);

→ int length;
gets (name);
length = strlen (name);

strrev – To reverse the string

- strrev (str);

strlwr – convert a string into lower case

strlwr (str);

- It can be used by ASCII values

Difference = 32

(lower case ASCII value = upper case ASCII value − 32)

A

WAP

Arrays of strings :-

It is just a 2D array of chars.

```
char A[3][6] = { "Ryan",
                 "Tom",
                 "Jackie" };
```

3 → no of strings

6 → no of char per string.
(If no is not mentioned, compiler will take the longest length of the name as the max length)

WAP to check a string as a palindrome — Page 4 of 25 of Book 2

~~STACK~~

Array as Parameter

Ptr 2 parameters

```
display (int *, int);
main()
{
  int num[] = {1,2,3,4,5,6};
  display (&num[0], 6);
}
display (int *j, int n)
{
  int i;
  for (i=0; i<=(n-1); i++)
  {
    printf ("\n element = %d ", *j);
    j++;
  }
}
```

---

single parameter

```
void display (int *)
main()
{ int n[] = {1,2,3}
  display (&n[0]);
}
void display (int *a)
{
  for (int i=0; i<=2; i++)
  { printf ("%d", *a); a++; }
}
```

Length of string

```
void length (char *);
main()
{
  char a[10] = "computer";
  length (&a[0]);
}
void length (char *a)
{ for (int c=0; *a!='\0'; c++) a++;
  printf ("%d", c);
}
```