PROGRAMMING METHODOLOGY

&

DATA STRUCTURES

Q:

```
main()
{
 int a=5;
 int b=6;
 printf("%d %d %d", a==b, a<b, a>b);
}
```

Sol: In the 'printf' statement

if a==b, it will return True
else it will return False.

Here 5 ≠ 6. So it will print 0 to indicate false.

a<b ⇒ True, so print 1

a>b ⇒ False, so print 0.

∴ Output: 010

Q:

```
main()
{
 int a=5;
 int b=6;
 printf("%d", a=b);
 printf("%d", a==b);
}
```

Sol: Given that a=5, b=6.

in the first printf statement, b's value is copying to a.

So a=6, b=6.

This printf statement will print 6

In next printf statement, values of both a and b are compared. Value of both are 6 now. a==b is true. So it will print 1.

Output :- 6 1

Q: main()
{
int a=5;
int b=6;
printf("%d %d %d", a==b, a=b, a<b);
}

Sol: a=5, b=6.

In 1st statement, ie., a==b, a not eql b so it should print 0 as false.

In 2nd statement a=b, b's value copied to 'a' and it should print 6.

In the last statement a<b, 6<6 so it should print 0 as false.

BUT THIS IS NOT THE CASE

This proogram is not going to return 060 as O/P.

If we compile this pgm in Mac compiler

O/P ⇒ 060

in Dev C++ ⇒ 161

in some online compiler ⇒ 160

There will be different outputs.

There is a reason why it gives different O/Ps. Reason is "Undefined Behaviour".

It says whenever in a single statement change the value of ~~the value of~~ variable and use of that variable in the same statement will leads to undefined behaviour.

Any one of the 3 statements may occur first.

One more example

```
main()
{
  int a=0;
  printf("%d %d %d", a++, a++, ++a);
}.
```

It should ~~have~~ print 0 1 3, but it may not.

So to solve this, use separate printf statements for each statement to get the desired output.

What is the output of the following programs?

a) main()
{
int a=032;
printf("%od",a);
}

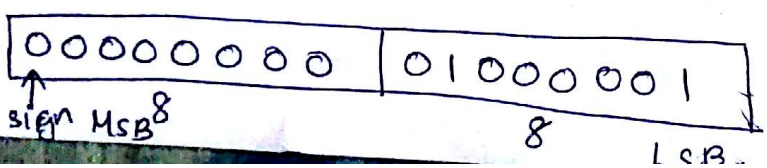
Sol: printf statement printing the statement as %od, ie., as decimal value.
Whenever we start an integer number with 0, the C compiler by default take this number in octal format. Here printing as decimal.
Convert the octal number to decimal

$032 \Rightarrow 3\times8^1+2\times8^0 = 26.$

∴ Output : 26.

b) main(){
int a=65;
printf("%oc",a);
}

Sol: Here we have an integer number and we have to print the character value of it.
Size of integer variable in C language is 2 bytes in 32-bits s/ms and 4 bytes in 64 bits s/ms.
Consider as 2 bytes.



It is the 2 bytes (16 bits) form of 65 in binary.

Size of char is 1 byte in C language

so char will take Least significant 1 byte and print the ASCII value.

∴ ASCII value corresponding to 65 is 'A'

output:- A

c) main () {

int a = 0101;

printf ("%d", a);

printf

~~Scanf~~ ("%i", a)

printf ("%c", a);

}

Sol: C compiler will take this number as octal no.

In 1st printf, the no. will print as decimal no. corresponding to octal no.

$0101 \rightarrow 1 \times 8^0 + 0 \times 8^1 + 1 \times 8^2 \Rightarrow 65$

%i used for integers

%o i and %d behave same when they are using in printf but behave different when use in Scanf.

∴ 2nd printf ⇒ 65

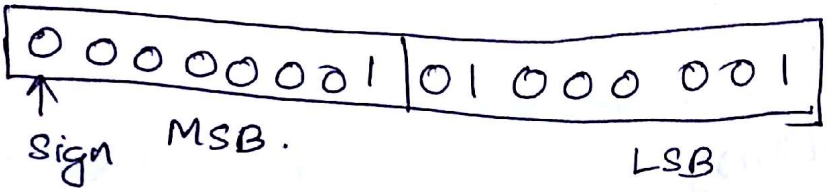
3rd printf ⇒ A.

∴ output : 65 65 A

d) main(){
int a = 321;
printf("%c", a);
}.

Sol: Here 321 is a character variable. Here we are printing the character value of a.

binary format of 321



2 | 321
2 | 160 → 1
2 | 80 → 0
2 | 40 → 0
2 | 20 → 0
2 | 10 → 0
2 | 5 → 0
2 | 2 → 1
1 → 0

LSB value is 65, Ascii value of 65 = 'A'

Output :- A

e) main()
{
int a;
1. scanf("%i", &a); // Suppose user entered 0x56
2. printf("a %i %d", a, a);
3. scanf("%d", &a); // Suppose user entered 0x56 again.
4. printf("a %i %d", a, a);
}.

(1)

Sol: %i in scanf:- 0x56 is entered. Since it contains 'x', the scanf will take it as hexadecimal value

decimal equivalent of 0x56 $\Rightarrow 5\times16^1+6\times16^0$

$= 65$

in (2) both %i and %d behave same.

∴ output 65 65

in (3) scanf(%d, &a)

since %d only accept the value before x, 0 is stored. Because %d in scanf will consider x as character and avoid anything comes after it

∴ o/p of (4):- 0 0.

Instead of 0x56, user has entered 0101 (octal)

%i in scanf assume as octal number

%d in scanf assume as decimal number.

∴ 1st printf will print 65 65 (decimal equivalent)

2nd printf will print 101 101

- what is the output of the following programs?

a) main()

if (0){printf("gate lectures.com");}

if (1){printf("ugcnet lectures.com");}

}

Sol: If (0) means if false

∴ That statement is not going to execute.

If (1) means true.

∴ That statement will execute.

Output :-

ugcnetlectures.com.

b) main(){

if(-1){printf("best online coaching");}

if(0){printf("subscribe");}

if(-86){printf("you will clear the GATE Exam");}

if(86){printf("you will clear the UGC NET exam");}

}.

Sol: Whenever the user types any numbers other than

0 it is true.

output :-

best online coaching

you will clear the GATE exam.

you will clear the UGC NET exam.

c) main(){

int a=5, b=6;

if (a=b) { printf("Do you think it will print this statement");}

if (a==b){printf("Dost you really think you are right?");}

}

~~d) main(){~~

Sol: If (a=b) means copying b's value to a

∴ it becomes if (6) ⇒ if true

∴ That statement will be executed.

if (a==b) ⇒ True.

It will executed.

d) main(){

float a=5.66;

printf("%d", a);

}

Sol: %d is used for integers

It will print 5.

• what are the outputs of the following pgms?

1) main() {
int a=128;
printf("%d", 1+a++);
}

(a) 126 (b) 127 (c) 128 (d) 129 (e) 130

Sol: a=128.
a++ is post incrementation
1+128 = 129

2) main() {
int a=0;
if(a++){printf("do you really think this is the output %d", a);}
if(a){printf("really are you that mad? %d", a);}
if(++a){printf("it is magic %d", a);}}

Sol: a=0.
In 1st print, a++ is post increment
∴ if(0), it is false, so not executed.
next print if(a) ⇒ if(1) ⇒ print 1
So it will execute.
Next will print 2.

3) main()
{
printf("hi %d \n", printf("hello"));}

Sol: 1st it will execute inner printf

hello.

next outer

hi

next execute %d, ie; no. of characters in inner printf ⇒ hello
1 2345

∴ output:- hello hi 5

4) main(){
if (printf("gelloo")){
printf("it printed");}}

Sol: "gelloo" is printed and 'if' statement will become
if (6) ⇒ if (true)
so the printf will executed.

∴ output:-
gelloo it printed.

Q: #include <stdio.h>

```
void main()
{ int i=5;
  Printf("%d %d %d",++i, i, i++);
}
```

what is the output of the above pgm?

(A) 5,6,7 (B) 6,6,7 (C) 7,6,5 (D) 6,5,6.

(E) UNDEFINED

Sol: E

NB Q: The minimum number of temporary variables needed to swap the contents of two variables is

(a) 1 (b) 2 (c) 3 (d) 0

Sol: classically we use the following method.

a = 5, b = 6.

```
temp = a
b = a
a = temp.
```

Here no. of temporary variable = 1

But we can also do swapping without temp.

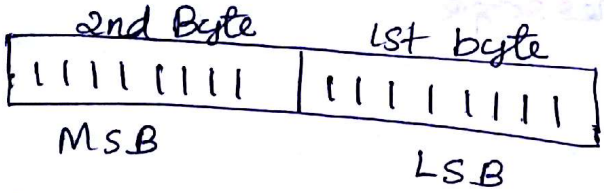
```
a = a+b;
b = a-b;
a = a-b;
```

∴ option (d)

Q: If integers needs two bytes of storage, then maximum value of an unsigned integer is ?

(a) $2^{16}-1$ (b) $2^{15}-1$ (c) 2^{16} (d) 2^{15}

Sol:



when all the bits are 1's, the value is maximum

∴ The value is $2^{16}-1$

[1000000000000000 $\Rightarrow 2^{16}$.

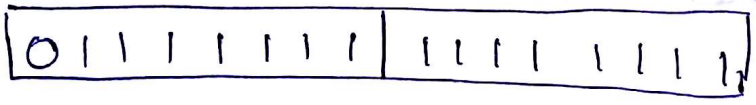
$2^{16}-1 \Rightarrow$ 1111 1111 1111 1111]

Q: If integers needs two bytes of storage, the maximum value of an signed integers is ?

a) $2^{16}-1$ (b) $2^{15}-1$ (c) 2^{16} (d) 2^{15}

Sol: In signed integers MSB is the sign bit.

0 ⇒ +ve, 1 ⇒ -ve.



$\Rightarrow 2^{15}-1$

Maximum value of signed integer is when all 1's except MSB.

∴ $2^{15}-1$.

Q: The following program fragment

```
int a=5, b=6;
printf("%d", a==b);
```

a) Outputs an error message

b) Prints 0

c) Prints 1

d) None of the above.

Sol: a==b is a conditional statement.

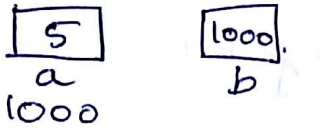
O/P is 0 ⇒ False.

∴ option b.

Q: The operator used to get ~~the~~ value at address stored in a pointer variable is:

(a) * (b) & (c) && (d) ||

Sol:



int a; → variable.

int *b;

b=&a;

printf("%d", *b); / prints 5

printf("%u", b);

~~%~~u is used to print the address stored in a variable.

∴ option (a)

Q: The purpose of the following program fragment

```
b = s+b
s = b-s
b = b-s;
```

where s, b are two integers is to:

(a) transfers the contents of s to b.

(b) transfers the contents of b to s.

(c) Exchange (swap) the contents of ~~s~~ and b.

(d) negate the contents of s and b.

Sol: (c)

Suppose

s = 5, b = 6.

b = s+b = 5+6 = 11.

b = 11 now.

s = b-s = 11-5 = 6.

s = 6.

b = b-s = 11-6 = 5

b = 5

Now values are swapped.

Q: The following program fragment

```
int a=5, b=6;
printf("%od", a!=b);
```

a) outputs an error message.

b) prints 0

c) prints 1

d) None of the above.

Sol: c

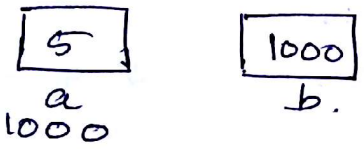
Q: The follow program fragment

```
int a=5;
int *b=&a; *b= *b+2;
printf("%od",a);
```

a) output as an error message b) prints 5

c) prints 7 d) None of the above

Sol: c



*b = *b+2.

it is equivalent to

a = a+2 = 5+2 = 7

in printf, value of a is printed, ie 7.

Q: Consider the following program fragment.

```
if (a>b)
if (b>c)
        S1;
else S2;
```

S2 will be executed if

a) a<=b b) b>c c) b<=c and a<=b

d) a>b and b<=c

Sol: d

The else statement is of inner if's. So the else statement is executed if 1st 'if' is true and 2nd 'if' is false.

ie:, a>b and b<=c

Q: a<<1 is equivalent to

a) Multiplying 'a' by 2

b) dividing a by 2.

c) Adding 2 to a

d) None of the above.

Sol: a), d) [because not all cases are true]

a<< is used as left shift.

Suppose 192 as 1 byte

11000000.

a<<1 means shift by 1 position

10000000.

last bit will be lost.

Suppose a=4 ⇒00000100

a<<1 ⇒ 00001000 ⇒8.

ie; Multiplying by 2.

Q: The most significant bit will be lost in which of the following operations.

a) >> b) << c) Complementation d) None of the above.

Sol: b.

Q: The following program.

```
main()
{
float a=.5, b=.7;
if (b<=7)
if (a<.5) printf("IES");
    else
        printf("PSU");
    else
        printf("GIATE");
}
```

Output

a) PSU (b) IES (c) GIATE (d) None of these

Sol: a.

a	b
.5	.7

Q: The statement ~~Pscan~~

printf("%d", 10?0?5:11:12);

points

(a) 10 (b) 0 (c) 12 (d) 11

Ans: d

The statement can be coded as

```
1. if (10)
   {
2.   if (0)
3.     then 5;
4.   else
5.     it is 11;
   }
6. else
7.   it is 12;
```


line 1 will be executed since if(10) ⇒ if (true).
when line 2 is ~~else~~ executed, ie; if (0) ⇒ if (false).
So it does not execute. else part will be executed.
That is 11.

Q: Consider the following declaration

int a, *b = &a, **c = &b;

The following program fragment

a = 4;

**c = 5;

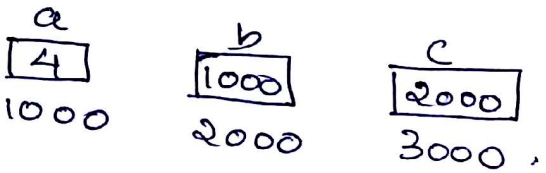
a) Does not change the value of a

b) assigns address of c to a

c) assigns the value of b to a

d) assigns 5 to a.

Sol: d



**c = 5 ⇒ *(*c) = 5

i.e., *c = 1000.

**c = *(1000) = a

i.e., a = 5

∴ assigns 5 to a.

Q: The following statement

printf("%f", a/5);

Prints

a) 1.8 b) 1.0 c) 2.0 d) None of these

Sol: b

a and 5 are integers so 1.0

Q. printf("%f", (float) a/5);
prints

a) 1.8 b) 1.0 c) 2.0 d) None of these

Sol: a

Q: if (a=0)
printf("a is zero");
else
printf("a is not zero");
result in the printing of

a) a is zero b) a is not zero
c) Nothing d) Garbage.

Sol: b.

if (a=0) ⇒ if (0) ⇒ false will not execute.

Q: The following loop

for(i=1,j=10; i<6; ++i,--j)
printf("%d %d", i,j);
prints

a) 1 10 2 9 3 8 4 7 5 6
b) 1 2 3 4 5 10 9 8 7 9
c) 1 1 1 1 1 9 9 9 9 9
d) None of these.

Sol: a)

i=1, j=10.

Print 1 10

++i , --j
↓ ↓
2 9

Print 2 9

∴ O/P 1102938756.

Q: The following Program fragment

int i=107, x=5;

Printf((x>7)? "%od": "%oc", i);

results in

a) An execution error

b) A syntax error.

c) Printing of k.

d) None of these.

Sol: C

if (x>7)
{
%od
}
else {
%oc
}.

x>7 is false so else part will execute.

printf becomes

printf("%oc", i);

a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p
97	98	99	100	101	102	103	104	105	106	107					

character value of 107 is printed. ie, k.

Q: The following statements

```
for(i=3; i<15; i+=3)
{ printf("%d", i);
  ++i;}
```

will result in printing of

a) 3 6 9 12 b) 3 6 9 12 15

c) 3 7 11 d) 3 7 11 15

Sol: c

3 7 11 ~~15~~

Q: If a=9, b=5 and c=13, then the expression (a-a/b*b%c)>a%b%c evaluates to

a) true b) false

c) Invalid d) 0.

Sol: b

/, * and % have equal precedence. So they calculated from left to right.

9-9/5*5%13 > 9%5%13.

9-1*5 > 4%13

4 > 4 ⇒ false.

Q: Consider the following program fragment

```
if (a>b) printf ("a>b");
else
    printf ("else part");
    printf ("a<=b");
```

then a<=b will be printed if

a) a>b b) a<b

c) a==b d) all of the above.

Sol: d

The printf ("a<=b") does not associated with 'if' statement. It is printed always.

Q: The following statement

```
if (a>b)
if (c>b)
printf ("one");
else
if (c==a) printf ("two");
else print ("three");
else print ("four");
```

if a=5, b=6, c=7, then the expression prints _____

Sol: four.

Operator precedence and Associativity

Q: main() {

```
int x;
x = -3+4*5-6; printf("%d\n", x);
x = 3+4%5-6; printf("%d\n", x);
x = -3*4%-6/5; printf("%d\n", x);
x = (7+6)%5/2; printf("%d\n", x);
```

Output:-

```
11
1
0
1
```

$-3+4*5-6 = -3+20-6 = 11$

$3+4\%5-6 = 3+4-6 = 1$

$(-3)*4\%(-6)/5 = (-3)*(4/5) = -3*0 = 0$. $(-3)*4\%(-6)/5 = -12\%-6/5 = 0/5 = 0$

→ unary minus has higher precedence than *, % and /

$(7+6)\%5/2 = 13\%5/2 = 3/2 = 1$

Q: main() {

```
int x=2, y, z;
x *= 3+2; printf("%d\n", x);
x *= y=2=4 ;
x = y==2;
x == (y=2);
```

Output:-

```
10
40
1
1
```

N.B

$\Rightarrow x *= 3+2 \Rightarrow x = x*(3+2) = 2*(3+2) = 6+2=8$ (10)

$\Rightarrow x *= y = z = 4.$

Assignment operation has higher precedence than multiplication. Also assignment operation is right associative.

4 is copied to z, z's value, ie; 4 is copied to y.

then $x *= y \Rightarrow x = x*y = 2*4 = 8$ (10).

$\Rightarrow x = y == z$

y==z is a comparison statement. its value is true. so 1 is assigned to x.

So 1 is printed

$\Rightarrow x == (y=z)$

y is assigned the value of z; ie., 4.

The value 4 is now compared with x

x's value is 1

1 == 4, So it is false.

But value printed is x's value. So 1 is printed.

Q: `#define pprint(int) printf("%d\n", int)`

```
main(){
int x, y, z;
x=2; y=1; z=9;
① x = x && y || z; pprint(x);
② pprint (x || !y && z);
③ x=y=1;
④ z = x++ -1; pprint(x); pprint(z);
⑤ z += -x ++ + ++y; pprint (x); pprint (z);
}.
```

Output:-

```
1
1
2
0
3
0
```

x = 2, y = 1, z = 9

x	y	x && y
T	T	T
T	F	F
F	T	F
F	F	F

① x = x && y || z

⇒ 2 && 1 || 9 ⇒ T || 9 ⇒ T ⇒ 1

&& is having more precedence than ||

② (x || !y && z) ! > && > ||

value of x is 1 now.

1 || !1 && 9 ⇒ 1 || 0 && 9 ⇒ 1 || 0 ⇒ 1

③ x = y = 1

④ z = x++ -1 ⇒ 1 - 1 = 0,

x = 2 (Printed) and z = 0 (Printed)

x = x ⑤

-2 + 2 = 0.

z = 2 + 0 = 0.

x =

⑤ ~~coil~~ can be rewritten as

NB. $z+=((-(x++))+(++y)$

$= -2+2=0$, $x=3$ & $y=2$ now.

$z = z+0 = 0$

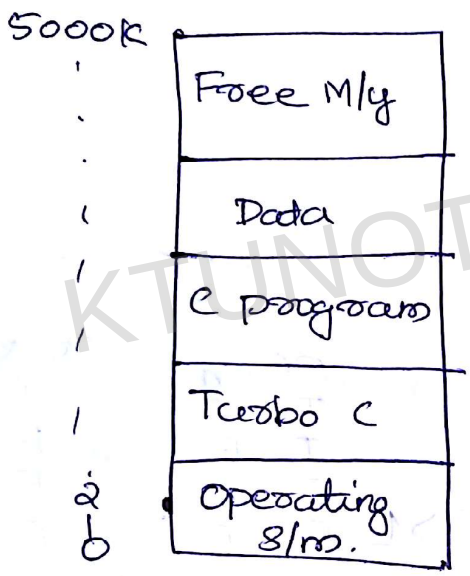
printed

printed.

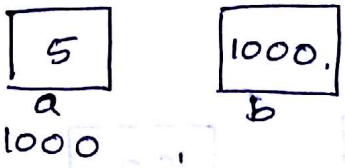
++ → unary -

Pointers

Pc's Memory Layout



Q: main()
{
int a=5;
int *b=&a;
printf("%op", b);
}



Sol: print 1000 in hexadecimal.
%op is used to print address of pointers.

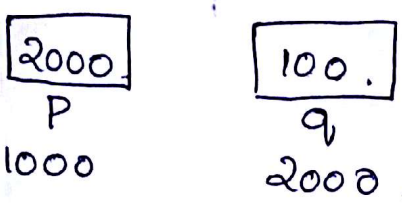
Q: main()
{
int a=5;
int *b=&a;
printf("%u", b);
printf("%d", b);
}

%u → used to print unsigned integers
%d → signed integers
Let size of int is 2 bytes.

Sol: will print address in decimal format.
the 2nd printf will not always give correct address since it is signed. If addresses are large and last bit is 1 it will indicate the last bit as -ve sign thus result in -ve address which cannot be possible.

Q: #include<stdio.h>
main()
{ int *p;
int q;
q = 100;
p = &q;
printf("%d", *p);
}

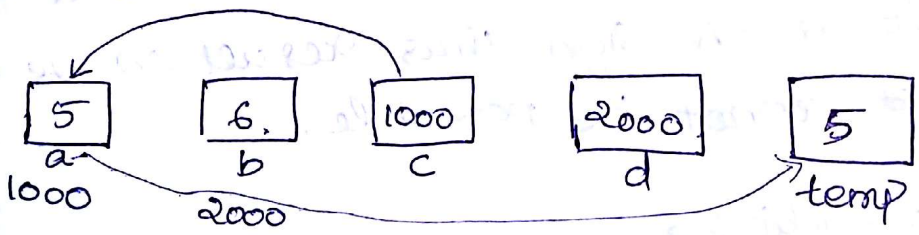
Sol: 100.



*P in printf will act as a dereferencer.

Q: main()
{
① int a=5;
② int b = 6
③ int *c = &a;
④ int *d = &b;
⑤ int temp = *c
⑥ *c = *d
⑦ *d = temp;
⑧ Printf("%d %d", a, b);
}.

Sol: 6 5



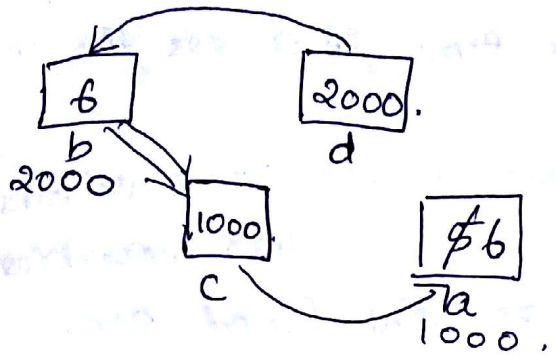
in int *c ⇒ we are declaring a pointer

in ③ & ④ ⇒ dereferencer.

i.e., when no data type associated with it.

*c means we will go to the location pointed by the pointer c. Then take/look on the value.

*c = *d.



*d = temp.

∴ 6 5 is the o/p.

Some legal and illegal example of pointer Initialization :

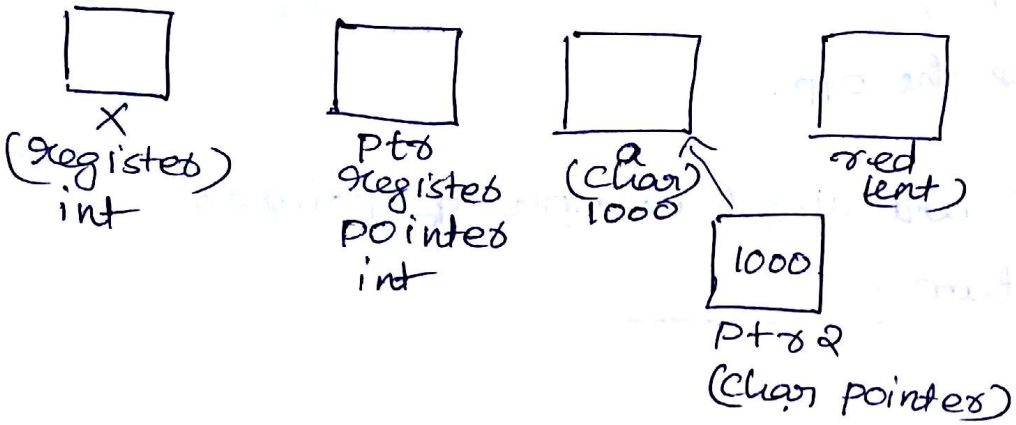
```
main()
{
① register int x;
② register int *ptr1;
③ char a;
④ char *ptr2;
⑤ int red;
⑥ ptr2 = &a;
⑦ ptr1 = &red;
⑧ ptr1 = &x;
⑨ ptr2 = &red;
⑩ ptr2 = &30;
⑪ ptr1 = &65;
⑫ ptr1 = &(a+0);
}
```

x is a register variable (stored in register).
ptr is a pointer & a stored in register.
Variables are stored in registers when they are again and again in pgms.

registers are closest to CPU and its speed is fastest. It takes less time to access the registers.

If a pointer is of char, it will store address of char variables.

Same in the case of int, float etc.



ptr1 = &red ⇒ also allowed

ptr1 = &x;

but x is a register variable. There is no need to find the address of register variable. So it is invalid.

(9) red is int but ptr2 is char. So invalid.

(10) ptr2 = &30.

Here 30 is a constant not a variable. We cannot store the address of a number. So invalid

(11) invalid

(12) ptr = &(a+3)

↳ it is an eqn

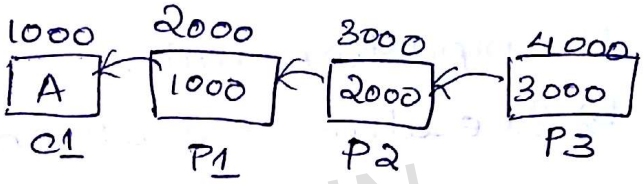
we cannot store the address of an eqn.

So what we cannot do in pointers?

- Store the address of a register variable.
- " " " " integer variable to char
- " " " " eqn
- " " " " constant number.

Levels of Indirection in pointers

```
/* A Pgm to illustrate the levels of indirection */
main()
{
char c1 = 'A';
char *P1;
char **P2;
char ***P3;
P1 = &c1;
P2 = &P1;
P3 = &P2;
}
```



The ANCI C Std says all compilers must handle atleast 12 levels.

Tip:- Having two levels of indirection is common. Any more than that gets a bit harder to think about easily; don't do it unless the alternative would be worse.

GATE - 2014 (1 Marks)

Q: consider the following program in C language:

```
#include <stdio.h>
main()
{int i;
int *pi=&i;
scanf("%d", pi);
printf("%d\n", i+5);
}
```

which one of the following statements is TRUE?

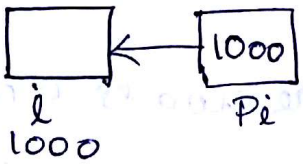
a) Compilation fails.

b) Execution results in run time error.

c) On execution, the value printed is 5 more than the address of variable i.

d) On execution the value printed is 5 more than the integer value entered.

Sol: d.



in scanf statement instead of giving address of i, they are given pi.

usually scanf statement is like

scanf("%d", &i);

Here pi is a pointer which stores the address of i. so the scanf statement in the pgm is a valid statement.

next statement is printf("%d", i+5);

we are adding 5 to the value taken from the user.

Q: what is printed by the following C Pgm? Gate-2008 (2 marks)

```
int f(int x, int *py, int **ppz)
{ int y,z;
  **ppz += 1; z = **ppz;
  *py += 2; y = *py;
  x += 3;
  return x+y+z;
}
void main()
{ int c, *b, **a;
  c = 4; b = &c; a = &b;
  printf("%d", f(c,b,a));
}
```

z = x
y = 7
x = 4

a) 18 b) 19 c) 21 d) 22.

Sol; b

Pgm execution starts from main().

4	1000	2000
c	b	a.
1000	2000	3000.

f(c, b, a) is called.

4	1000	2000		
x	py	ppz.	y	z.

**PP2+=1 ⇒ **PP2 = (*(*PP2))+1

⇒ *(*2000)+1 ⇒ 4+1=5

ie., **PP2=5

ie., c=5 now.

z=**PP2 ⇒ **2000 ⇒ *1000 = 5

z=5

*Py+=2 ⇒ *Py = *Py+2 ⇒ *1000+2 ⇒ *5+2=7

*Py=7 ⇒

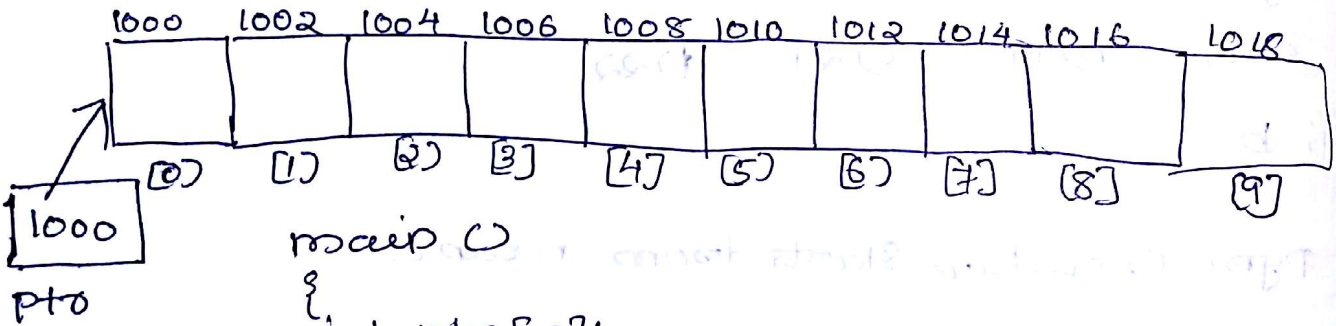
c=7 now.

y = *Py = 7

x=x+3 ⇒ 4+3=7

return (x+y+z) ⇒ 7+7+5 = 19

Relationship b/w Arrays & Pointers




```
main()
{
int ptr[10];
  :
}
```

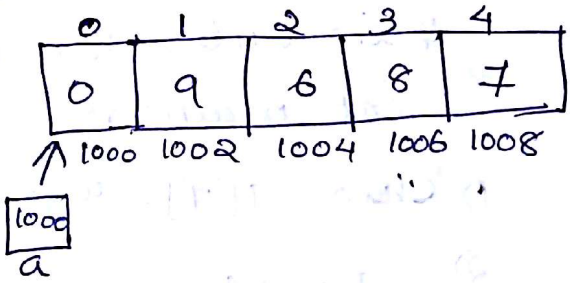
a[2]=3 means 3 is storing at index location 2.

In the pgm an array pto[10] of integer type is declared. Here pto is not the name of the array. pto is a pointer which is pointing to the base address or the starting address of these mly locations (contiguous mly locations).

Here pto is pointing to the base address of array. Every mly location is of 2 bytes since it is int. If size of int is 4 Bytes, each mly location is of 4 bytes.

- consider the program

```
main()
{
int a[5]= {0,9,6,7,8};
 int *p=a;
 printf("%u", a); //prints 1000.
① printf("%u", a+2); //
② printf("%u", *a+2);
③ printf("%d", a[2]);
④ printf("%u", a+4);
⑤ printf("%u", *(a+2));
⑥ p=p+2;
⑦ printf("%u", *p);
⑧ printf("%u", *p+1);
⑨ printf("%u", *(p+1));
}
```



storing a's value in *p.

① is equivalent to saying a+2*size of int
[Char → 1B, float → 4B]

⇒ 1000+2*2 = 1004 is printed

② *a+2 ⇒ 0+2 = 2

③ a[2] = 6

④ a+4 ⇒ 1000 + 4*2 = 1008

⑤ *(a+2) ⇒ *(1004) = 6

⑥ P=P+2 = 1000 + 2x2 = 1004.

⑦ *P = 6.

⑧ *P+1 = 6+1 = 7

⑨ *(P+1) = *(1004 + 1*2) = *1006 = 8

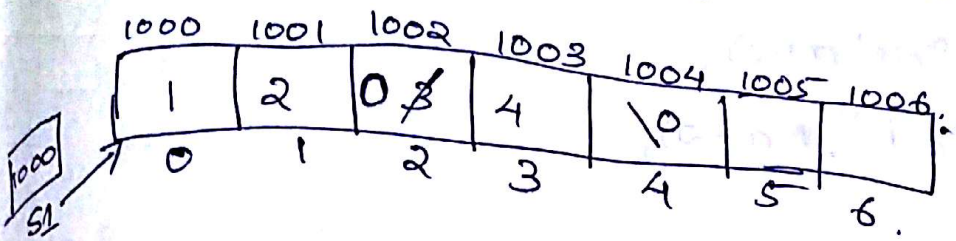
Q: Consider the following C program segment (Gate-2015)

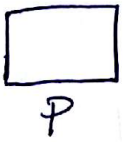
```
#include <stdio.h>
int main(){
1) Char S1[7] = "1234", *P;
2)    P = S1+2;
3)    *P = '0';
4)    printf("%s", S1);
}
```

what will be printed by the pgm?

a) 12
b) 120 400
c) 1204
d) 1034.

sol: s1 [7] is a char array and "1234" is a string.





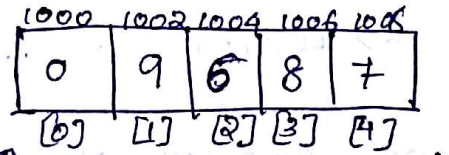
① p = s1+2. ⇒ 1000+2*1 = 1002

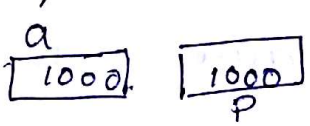
② *p = '0'. ('0' is a string)
p is storing the value 1002, value at 1002 is 3.
this 3 is changed to '0'.

③ The string till the '\0' is printed.
ie., 1204
option (c)

Q: what is valid and what is not valid?
Consider the pgm.

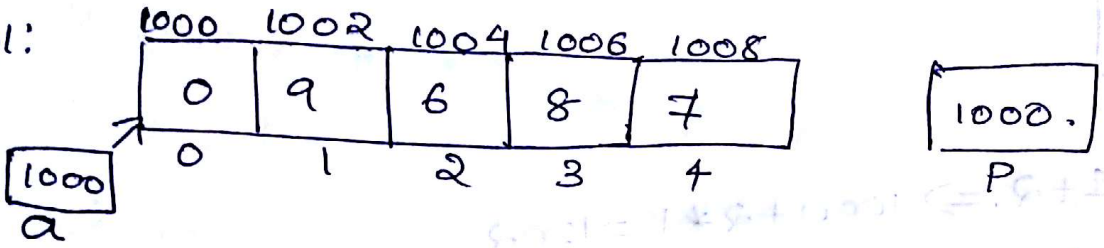
```
main()
{
1) int a[5] = {0,9,6,8,7};
2) int *p = a;
3) printf("%u", a);
4) printf("%u", a+2);
5) printf("%d", *a+2);
6) p = p+2;
7) printf("%u", *p);
8) printf("%u", *p+1);
9) printf("%u", *(p+1));
10) a = a+2;
```





11) printf("%u", a);
12) printf("%u", a+2);
13) printf("%d", *a+2);
}

Sol:



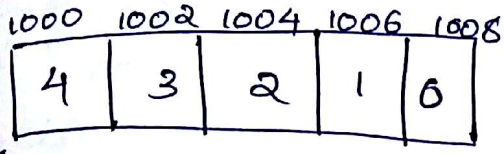
2) *P=a
3) print 1000.
4) a+2 = a+2*2 = 1004
5) *a+2 = *1000+2 = 0+2 = 2.
6) P = P+2 = 1000+2*2 = 1004
7) 6 is printed
8) *P+1 = ~~6++~~ 6+1 = 7.
9) *(P+1) = *(1004+2) = 8.
10) a=a+2 ⇒ 1000+2*2 = 1004 X but not valid

we cannot change/update the address stored in 'a' because 'a' is a mnemonic but not a variable. (Here but we can change/update the address stored in the pointer 'P' because 'P' is a pointer variable).

∴ all the ⑪, ⑫, ⑬ are invalid.

Q: main()
{ int a[5] = {4,3,2,1,0};
① printf("%od", *a);
② printf("%od", a[0]);
③ printf("%od", a[2]);
④ printf("%od", *(a+2));
}

Sol:



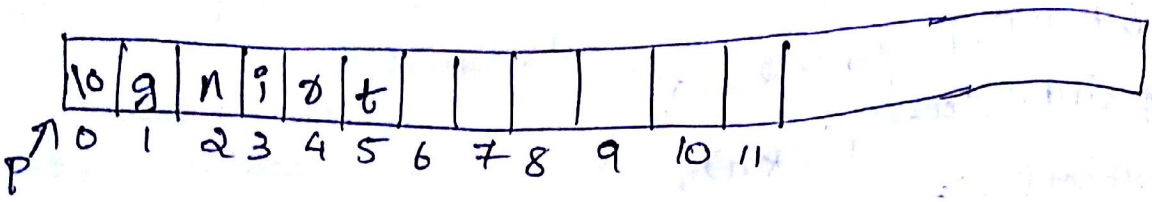
a
1000

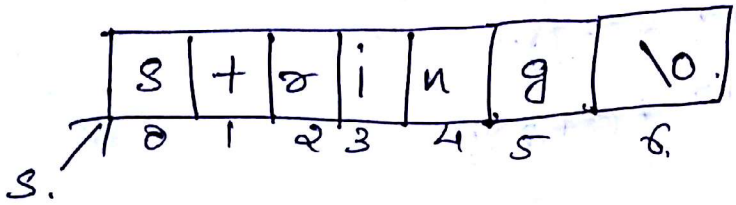
① → 4
② → 4
③ → 2.
④ *(a + 2*2) = *(000+4)
= *(004) = 2

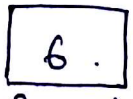
*a == a[0]
a[2] == *(a+2)
*(a+3) == a[3].

Q: consider the following C pgm segment (Gate 2004: 2 marks)
1) char p[20]
2) char *s = "string";
3) int length = strlen(s);
4) for(i=0; i<length; i++)
5) p[i] = s[length-i];
6) printf("%s", p);
The output of the program is
a) gnirts b) string
c) gniot d) no o/p is printed.

Sol: d







for(i=0; i<6; i++)

P[0] = s[6-0]

P[0] = s[6] ⇒ \0

P[1] = s[6-1] = s[5] g

P[2] = s[6-2] = s[4] n

P[3] = s[6-3] = s[3] i

P[4] = s[6-4] = s[2] r

P[5] = s[6-5] = s[1] t

i < length is false ⇒ loop stops.

But in the printf statement first '\0' is encountered. So compiler will assume that string is ended. So NO OUTPUT.

Q: Consider the following C program

```
main()
{ int x, y, m, n;
  scanf("%od %od", &x, &y);
  /* assume x>0 and y>0 */
  m=x; n=y;
  while (m!=n)
  {
    if (m>n)
      m=m-n;
    else
      n=n-m;
  }
printf("%od", n);
}
```

The program computes

a) x/y using repeated subtraction

b) x mod y using repeated subtraction

c) The GCD of x and y

d) LCM of x and y.

Ans: C

x	y	m	n
40	32	40	32.

assume x=40 & y=32

m>n ⇒ m=m-n = 40-32 = 8, m=8.

m<n ⇒ n=n-m = 24, n=24

m<n ⇒ 24-8 = 16.

m<n ⇒ 16-8 = 8.

m=n now. So come out of while loop.

now print the value of n, ie., 8.

a) $\frac{40}{32} = 1$ false.

b) x mod y = 40 mod 32 = 8

c) GCD of 40 & 32 ⇒ 8.

d) LCM of x & y ⇒ not 8.

b and c may be true.

Proceed with another values.

x = 15, y = 5

m = 15, n = 5

m > n

m = 15 - 5 = 10.

m > n

m = 10 - 5 = 5

m = n come out of loop.

print 5

x mod y = 15 mod 5 = 3.

∴ option c is correct.

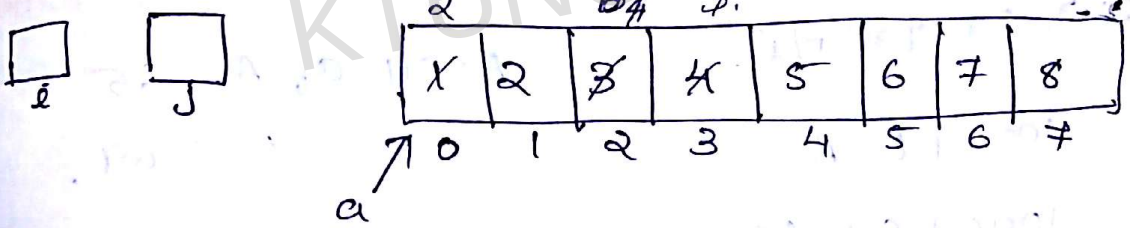
Q: Consider the C pgm given below. what does it print? (Gate 2008 : 2 Marks)

```
#include <stdio.h>
int main()
{
① int i,j;
② int a[8] = {1,2,3,4,5,6,7,8};
```

```
① for (i=0; i<3; i++)
   {a[i] = a[i]+1;
②   i++;
③  }
④ i--;
⑤ for (j=7; j>4; j--)
   { int i=j/2;
⑥
⑦    a[i] = a[i]-1;
   }
⑧. printf("%od %d", i, a[i]);
 }
```

a) 2,3 b) 2,4 c) 3,2 d) 3,3.

Sol: c



④ ⇒ a[0] = a[0]+1 = 2.
a[2] = a[2]+1 = 3 4
i=4 so come out of loop

⑥ ⇒ i-- ⇒ i=3.

⑦ j=7
i = j/2 = 7/2 = 3.
a[i] = a[j]-1 ⇒ a[3] = a[3]-1 = 4-1 = 3 ∴
j=6
i = j/2 = 6/2 = 3.
a[3] = a[3]-1 = 3-1 = 2.
j=5
i = j/2 = 5/2 = 2; a[2] = a[2]-1 = 4-1 = 3

a[j=4. so come out of loop.
i = 3, a[i] = a[3] = 2

Q: what does the following fragment of C Pgm print? (Gate 2011 : 1 Mark)

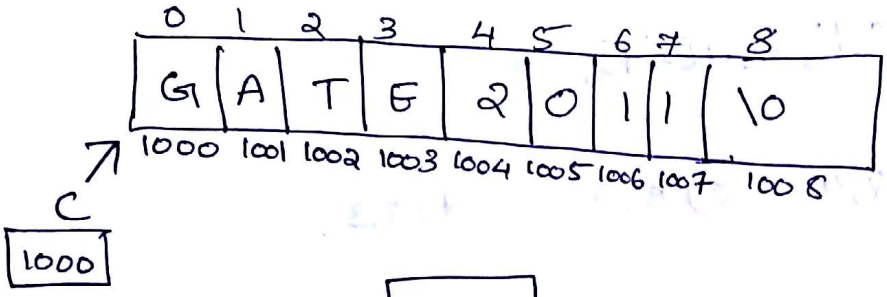
① Char c[] = "GATE2011";

② Char *P = c

③ .Printf("%s", P+P[3]-P[1]);

a) GATE 2011 b) E2011 c) 2011 d) 011

Sol: c



③ → P+P[3]-P[1].

ASCII of A = 65

E = 69.

1000+E-A

1000+69-65 = 1004

Here we are printing everything from 1004 since we are printing with %s.

[If %c is used, only the string/character at location 1004 is printed].

∴ ~~200~~ 2011 is printed

2D ARRAYS

Eg:- int a[2][2]

There are 2 ways in which m/y is allocated to the 2D array.

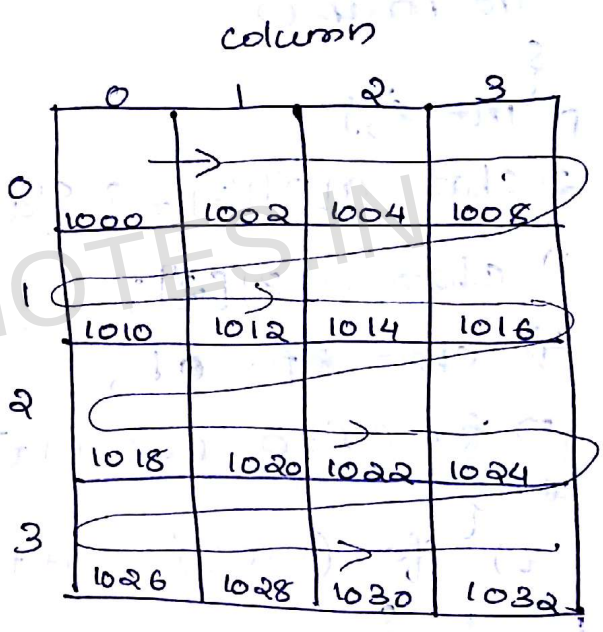
1. Row major order (used in C, C++, java, .Net)
2. Column major order. (Fortran, MATLAB)

Using row Major ordering in 2D Array

int ptr[4][4]; ↓Rows ←Columns

(assuming size of int is 2 bytes)

all the data are stored in row by row wise.



Q: If you are having a 2D array int a[19][20] let the base address is 1000, what will be the address of location a[17][16]? (Row major order)

Sol

number of rows crossed = 16×20 + 16 = 336

multiply it with size of int ⇒ 336×2 = 672.

add with base address ⇒ 1000 + 672 = 1672

$(i * c + j) *$ size of (int) + Base Address.

c → no. of columns in the array.

$(17 * 20 + 16) * 2 + 1000 = 1712$

$(i * c + j)$ gives the number of rows crossed

Q: A C program is given below. (Gate 2008; 2 marks)

```
#include <stdio.h>
int main()
{
1) int i,j;
2) char a[2][3] = { {'a','b','c'},{'d','e','f'} };
3) char b[3][2];
4) char *p = *b;
5) for (i=0; i<2; i++)
6) {
7)     for (j=0;j<3;j++)
       {
           *(p+2*j+i) = a[i][j];
       }
}
```

What should be the contents of the array b at the end of the Pgm?

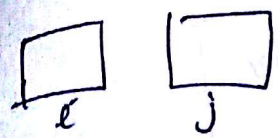
a) a b
   c d
   e f

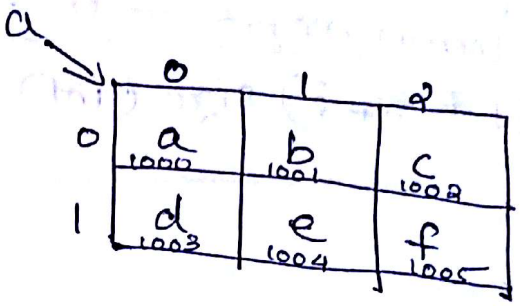
b) a d
   b e
   c f

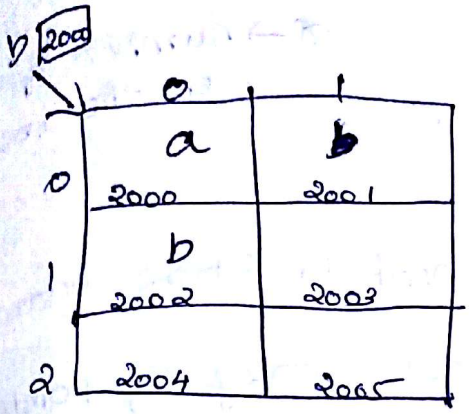
c) a c
   e b
   d f

d) a e
   d c
   b f

and b







5) i=0, j=0.

*(P + 2*j + i) = a[i][j]

*(2000 + 2*0 + 0) = a[0][0]

*2000 = a[0][0]

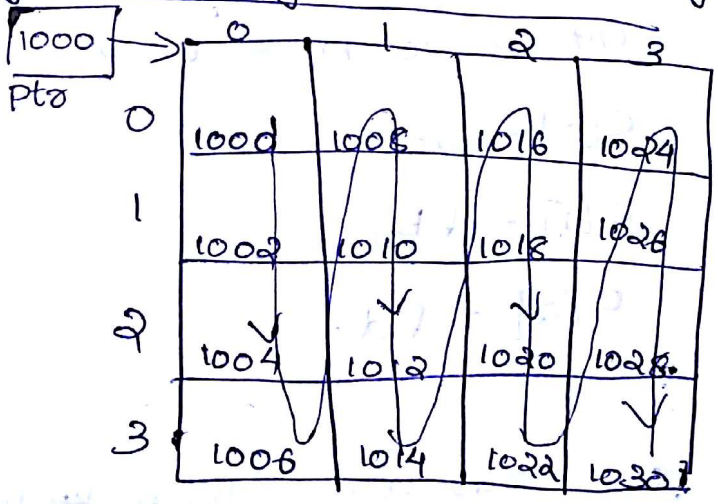
i.e., at the location 2000, a is stored.

j++ ⇒ j=1; j<3 now.

*(P + 2*1 + 0) = a[0][1] ⇒ *2002 = a[0][1]

and so on.

Using Column Major Ordering in 2D Array



To find the address of a given location in 2D array (Column major order).

$$(j * r + i)\ sizeof(int) + Base\ address$$

r → number of rows in the array.

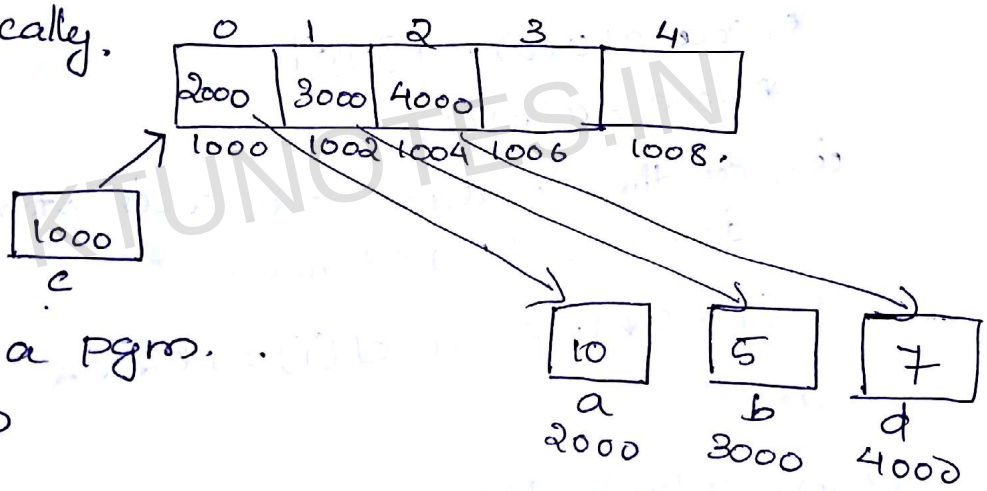
Array of Pointers

In some cases we may need to store addresses of multiple variables. For this array of pointers is used.

declaration

Eg:- `int *c[5];`

Graphically.

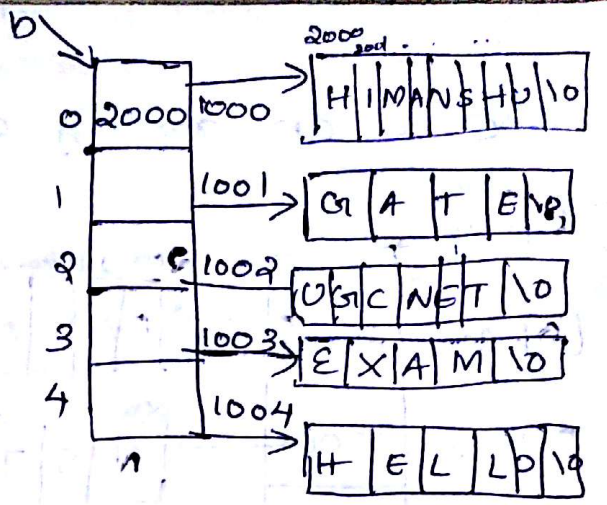


Consider a pgm.

```
main()
{
  int *c[5];
  int a=10, b=5, d=7;
  c[0] = &a;
  c[1] = &b;
  c[2] = &d.
}
```

we can access them just like the way we access normal arrays.

```
main()
{
char *b[5]
b[0] = "Hiranshu";
b[1] = "Gate";
b[2] = "UGC NET";
b[3] = "Exam";
b[4] = "Hello";
printf("%s", b[1]); // to print "GATE".
printf("%s", b[0]+3); // print ANSHU
printf("%s", b[3]+2); // AM is printed.
}
```



This is the way the strings are stored.

Q: #include <stdio.h>

1) char *c[] = {"ENTNG", "NST", "AMAZI", "FIRBE"};

2) char **cp[] = {c+3, c+2, c+1, c};

3) char ***cpp = cp;

4) void main()

{

5) printf("%s", **++cpp);

6) printf("%s", *((*++cpp)-1)+3);

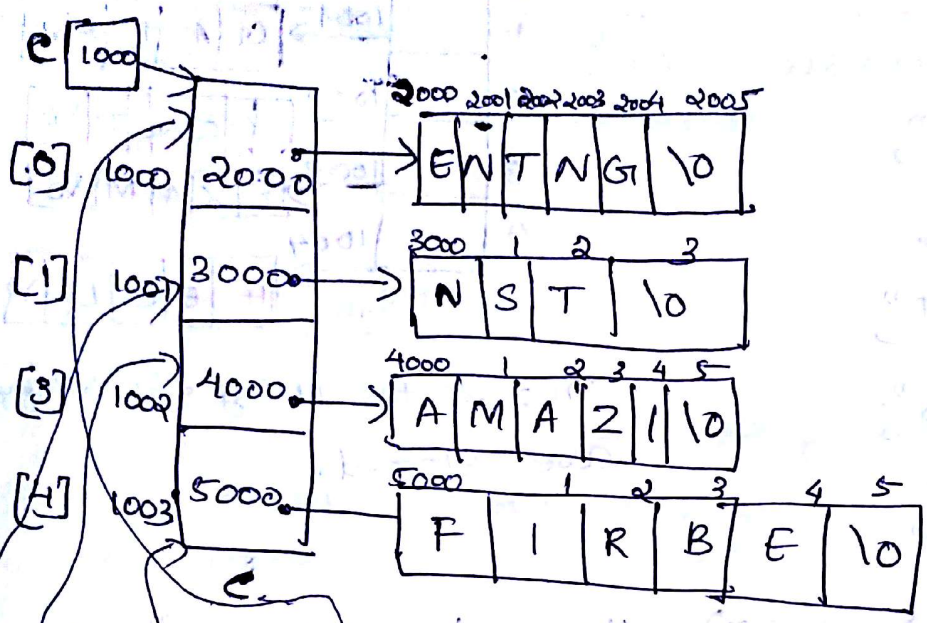
7) printf("%s", *cpp[-2]+3);

8) printf("%s", cpp[-1][-1]+1);

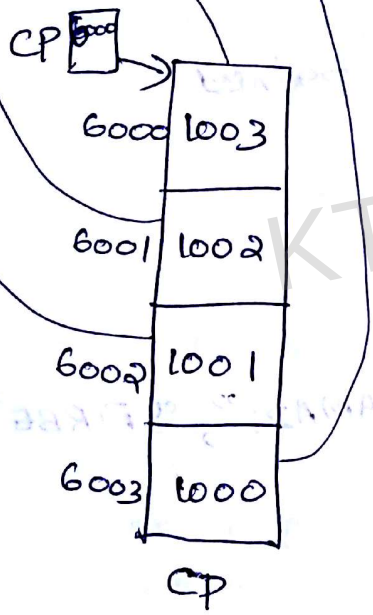
}

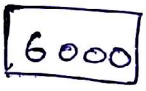
Explain the output.

c is an array of pointers.



② ⇒ having an array of pointers which is pointing to pointers.





⑤ ⇒ **++cpp. ⇒ **6001 ⇒ *1002 (4000)
⇒ 4000.
print from 4000 till \0
AMA21

⑥ *((**++cpp)-1)+3) ⇒ *((*6002)-1+3)
*((1001)-1+3) = *1003
*1000+3 = 2000+3 = 2003.

so NG will be printed

4) *CPP[-2]+3 this can be written as

((CPP-2))+3

((6002-2))+3 = *1003+3 = 5000+3 = 5003

BE is printed

5) CPP[1][-1]+1 ⇒*(*(CPP-1)-1)+1

((6002-1-1)+1

*(*6001-1)+1 = ~~*(1002) = 3000~~

*(1001)+1 ⇒ 3000+1 = 3001

ST will be printed

∴ Final output AMAZINGBEST

Functions

function declaration :-

returntype functionname(i/p parameters)

{

// Statements.

returndata;

}

Not every functions needs to have return type and return data.

Eg:- mainfunction.

- Area of a square using function

```
int area(int s)  // function will return integer no.
{                    → parameters / Arguments.
  int a = s*s;
  return a;
}
main()
{
 int side=5;
 int k;
 k = area(side);
 printf("%d", k);
}
```

- Function to find the area of a rectangle.

```
int  area(int a, int b)
{
   int aro;
   aro = a*b;
   return aro;
}
main()
{
 int a=5; b=6;
 int ari
 ari = area(a,b);
 printf("%d", ari);
}
```

Call by Value and Call by Reference

2 ways in passing parameters to a function.

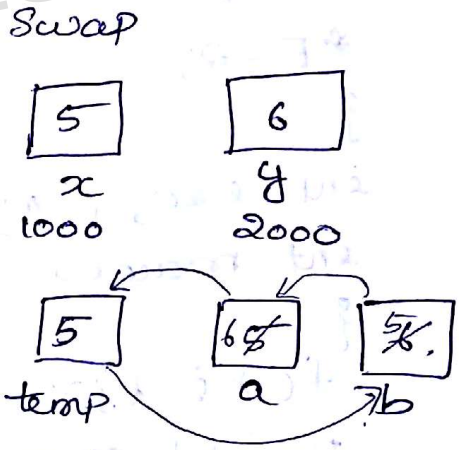
1. call by value
2. call by reference.

consider a function to swap the values of 2 variables.

```
swap (int a, int b)
{
int temp;
temp=a;
 a=b;
 b=temp;
}
```

```
swap2 (int *a, int *b)
{
int temp;
temp= *a;
 *a= *b;
 *b=temp;
}
```

```
main()
{
int  x=5, y=6;
 swap(x, y);
 printf("%d %d", x, y);
 swap2 (&x, &y);
 printf("%d %d", x, y);
}
```

a=6, b=5 now.

There is no return value in swap(), so even after swap() is called, the 1st printf() will print 5, 6 (x & y respectively).

swap2() ⇒ *a = &x, *b = &y.

a = 1000, b = 2000.

temp = *a = *1000 = 5 ⇒ temp = 5

*a = *b ⇒ *1000 = *2000.

x = 6 now.

*b = temp = *2000 = 5

new values, x = 6, y = 5

2nd printf() coill print 6,5

swap() ⇒ call by value

swap2() ⇒ call by reference.

Q: what does the following pgm print?

Gate 2010 : 1 marks

```
#include <stdio.h>
void f(int *p, int *q)
{
p=q;
*p=2;
}
int i=0, j=1;
int main()
{
f(&i, &j);
printf("%d %d \n", i,j);
return 0;
}
```

a) 2 2 b) 2 1 c) 0 1 d) 0 2.

Sol: d.

i and j are global variables.

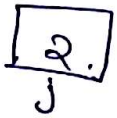
p = &i, q = &j

p = q

*p = 2.

*2000 = 2.

j = 2.



printf("%d %d", i, j);

i = 0, j = 2 now.

Q: The output of the following program is _____ (Gate 2015 : 1 marks)

```
void f1(int a, int b){
 int c;
 c=a; a=b; b=c;
}
void f2(int *a, int *b){
 int c;
 c= *a; *a= *b; *b=c;
}
int main(){
int a=4, b=5, c=6;
f1(a,b);
f2(&b,&c);
printf("%d", c-a-b);
}
```

Sol: -5

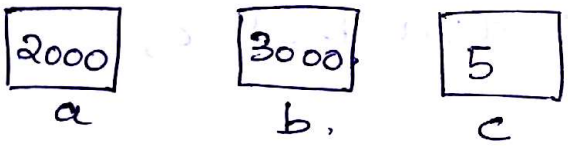
f1 is called (call by value)

[a, b in f1 is separate variables. Their scope is only limited to f1].

after completing f1 control goes back to main. Since f1 does not return any value, no change in original variable.

f2(&b, &c)

a = &b b = &c.



c = *a; ⇒ c = *2000 ⇒ c = 5

*a = *b ⇒ *2000 = *3000.

*b = c
*3000 = 5

& b = &

*b = c
*3000 = 5

point c-a-b. ⇒ 5-6-4 = -5

c = 5
a = &b
b = 5
b = 4

Q: consider the C pgm shown below. Gate 2003 : 2 marks

```
#include<stdio.h>
#define pr(x) printf("%d",x)
int x;
void Q(int z){
z+=x; pr(z);
}
void P(int *y){
int x=*y+2;
Q(x); *y=x-1;
pr(x);
}
main(){
x=5;
P(&x);
pr(x);
}
```

The output of the program is

a) 12 7 6

b) 22 12 11

c) 14 6 6

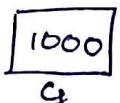
d) 7 6 6.

Sol: a.

main()

x is a global variable.

P(&x) ⇒ y = &x.



int x = *y+2 = *1000+2 (Here x is local variable)

Q(x) ⇒ Q(7)

z = z + x ⇒ 7 + 5 = 12

↓ global variable

Print (z) ⇒ 12.

* again go to PC)

* y = x − 1 ⇒ *1000 = 7 − 1 = 6.

↓ local.

Print(x) ⇒ Print local value of x ⇒ 7

Come back to main

Print (x) ⇒ global x.

∴ Print (6)

Use of Static Variables in C

```
fun()
{
  int a=0;
  Printf("%d", a++);
}
```

```
fun2()
{
  Static int a=0
  Printf("%d", a++);
```

```
main{
 fun();
 fun2();
  fun();
  fun2();
  fun();
  fun2();
 }
```

① fun() 

print 0, a=1 now.

but no return value for fun() so a value destroyed after printing

② fun22() 

print 0, a=1 now.

As soon as the fun2() ends ~~val~~ 'a' will not destroyed since it is static.

fun 1().

a=0,

print 0, destroyed

fun 2()

it will not initialize with a=0, it will go with the previous value.

print 1, a=2 now.

fun()

Print 0, destroy.

fun2()

Print 2, a=3 now.

O/P will be 0 0 0 1 0 2.

Gate 2015 : 2 marks

Q:
```
#include<stdio.h>
int x=10;
int f1() { int x=25; x++; return x; }
int f2() { static int x=50; x++; return x; }
int f3() { x*=10; x++; return x; }
int main() {
  int x=1;
  x+= f1()+ f2()+f3()+f2();
  Printf("%d", x);
  return 0;
}
```

Sol: 231

x is a global variable with value 10.

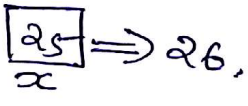
main()

x = 1, is local.

f1+f2+f3+f2 ⇒ which one will execute 1st will decided by associativity. Associativity of + is from left to right.

So f1

int x=25 // local.



x++

f1 return 26

f2

static int x=50/ local

x++

151 will return

f3

x = x * 10 | Here x is global.

= 10 * 10 = 100.

x++ ⇒ 101 is returned.

f2()

x++ ⇒ ~~52~~ 52 (since x is static here, no initialization this time)

~~26+5~~

∴ x = x + f1() + f2() + f3() + f2

= 1 + 26 + 51 + 101 + 52

= 231 is printed.

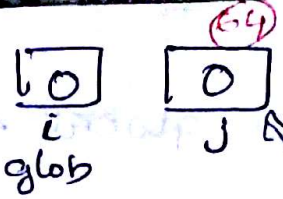
Q: The value of j at the end of the execution of the following C pgm is _____ Gate 2000; 2 Marks

```
int incr(int i)
{
  static int count = 0
   count = count + i;
   return (count);
}
main() {
  int i, j;
  for (i = 0; i <= 4; i++)
      j = incr(i);
}
```

a) 10 b) 4 c) 6 d) 7

Sol: (a)

main()



Now local i of incr() will destroyed but ccount will not destroyed.

i = 1,

j = incr(1) ⇒ ccount = 0+1=1

return (1)

j = 1

i = 2.

j = incr (2)

ccount = 1+2=3.

return(3)

j=3.

i=3.

j = incr (3)

ccount = 3+3=6.

return (6)

j=6.

i = 4

j = incr (4)

ccount = 6+4=10.

return (10), j=10.

MODULE 1

Introduction to programming methodologies

Programming methodology deals with the analysis, design and implementation of programs.

Algorithm

Algorithm is a step-by-step finite sequence of instruction, to solve a well-defined computational problem.

That is, in practice to solve any complex real life problems; first we have to define the problems. Second step is to design the algorithm to solve that problem.

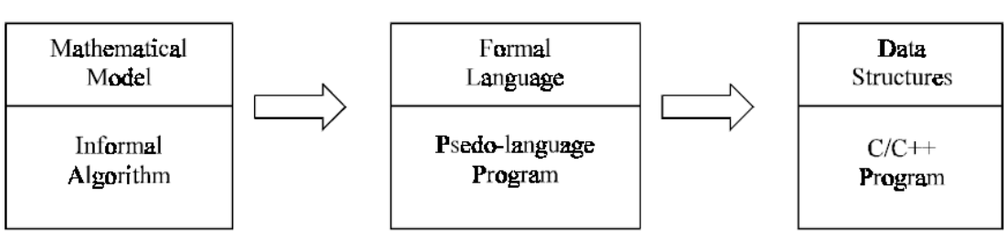
Writing and executing programs and then optimizing them may be effective for small programs. Optimization of a program is directly concerned with algorithm design. But for a large program, each part of the program must be well organized before writing the program. There are few steps of refinement involved when a problem is converted to program; this method is called ***stepwise refinement method***. There are two approaches for algorithm design; they are ***top-down*** and ***bottom-up*** algorithm design.

Stepwise Refinement Techniques

We can write an informal algorithm, if we have an appropriate mathematical model for a problem. The initial version of the algorithm will contain general statements, *i.e.*, informal instructions. Then we convert this informal algorithm to formal algorithm, that is, more definite instructions by applying any programming language syntax and semantics partially. Finally a program can be developed by converting the formal algorithm by a programming language manual. From the above discussion we have understood that there are several steps to reach a program from a mathematical model. In every step there is a refinement (or conversion).

That is to convert an informal algorithm to a program, we must go through several stages of formalization until we arrive at a program — whose meaning is formally defined by a programming language manual — is called stepwise refinement techniques.

There are three steps in refinement process, which is illustrated in Figure



1. In the first stage, modeling, we try to represent the problem using an appropriate mathematical model such as a graph, tree etc. At this stage, the solution to the problem is an algorithm expressed very informally.
2. At the next stage, the algorithm is written in pseudo-language (or formal algorithm) that is, a mixture of any programming language constructs and less formal English statements. The operations to be performed on the various types of data become fixed.
3. In the final stage we choose an implementation for each abstract data type and write the procedures for the various operations on that type. The remaining informal statements in the pseudo-language algorithm are replaced by (or any programming language) C/C++ code.

Programming style

Following sections will discuss different programming methodologies to design a program.

1. Procedural
2. Modular
3. Structured
4. Object oriented

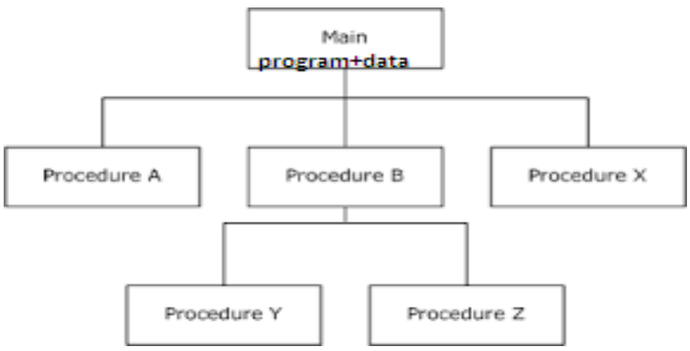
1.Procedural Programming

Procedural programming is a paradigm based on the concept of using procedures. Procedure (sometimes also called subprogram, routine or method) is a sequence of commands to be executed. Any procedure can be called from any point within the general program, including other procedures or even itself (resulting in a recursion).

Procedural programming is widely used in large-scale projects, when the following benefits are important:

- re-usability of pieces code designed as procedures
- ease of following the logic of program;
- Maintainability of code.
- Emphasis is on doing things (algorithms).
- Most of the functions share global data.
- Data move openly around the system from function to function.
- Functions transform data from one form to another.

Procedural programming is a sub-paradigm of imperative programming, since each step of computation is described explicitly, even if by the means of defining procedures.



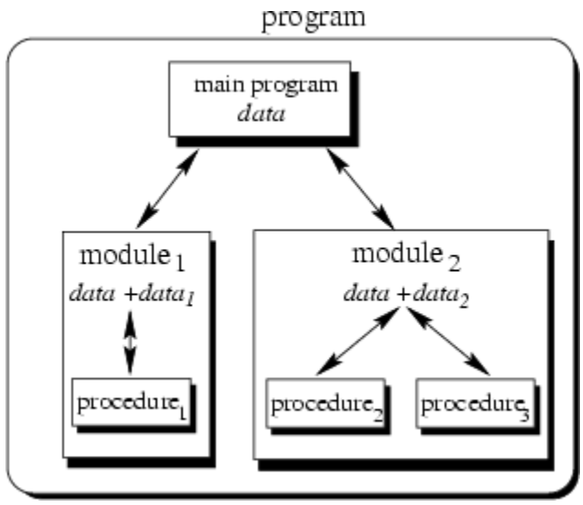
2.Modular Programming

The program is progressively decomposed into smaller partition called modules. The program can easily be written in modular form, thus allowing an overall problem to be decomposed into a sequence of individual sub programs. Thus we can consider, a module decomposed into successively subordinate module. Conversely, a number of modules can combined together to form a superior module.

A sub-module, are located elsewhere in the program and the superior module, whenever necessary make a reference to subordinate module and call for its execution. This activity on part of the superior module is known as a calling, and this module is referred as calling module, and sub module referred as called module. The sub module may be subprograms such as function or procedures.

The following are the steps needed to develop a modular program

1. Define the problem
2. Outline the steps needed to achieve the goal
3. Decompose the problem into subtasks
4. Prototype a subprogram for each sub task
5. Repeat step 3 and 4 for each subprogram until further decomposition seems counter productive



Modular Programming is heavily procedural. The focus is entirely on writing code (functions). Data is passive in Modular Programming. Modular Programming discourages the use of control variables and flags in parameters; their presence tends to indicate that the caller needs to know too much about how the function is implemented.

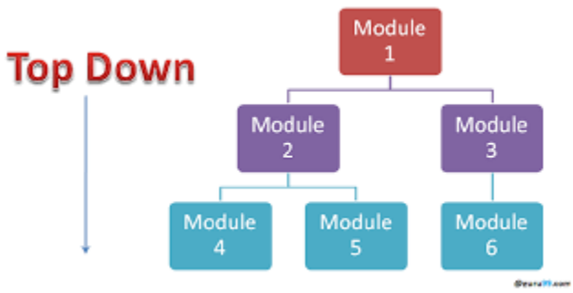
Two methods may be used for modular programming. They are known as **top-down and bottom-up**. Regardless of whether the top-down or bottom-up method is used, the end result is a modular program. This end result is important, because not all errors may be detected at the time of the initial testing. It is possible that there are still bugs in the program. If an error is discovered after the program supposedly has been fully tested, then the modules concerned can be isolated and retested by them. Regardless of the design method used, if a program has been written in modular form, it is easier to detect the source of the error and to test it in isolation, than if the program were written as one function.

Advantages of modular programming

1. Reduce the complexity of the entire problem
2. Avoid the duplication of code
3. debugging program is easier and reliable
4. Improves the performance
5. Modular program hides the use of data structure
6. Global data also hidden in module
7. Reusability- modules can be used in other program without rewriting and retesting
8. Modular program improves the portability of program
9. It reduces the development work

Top- down modular programming

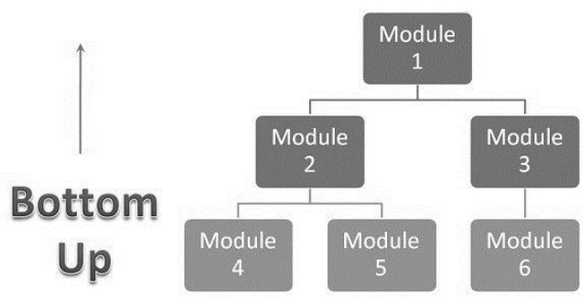
The principles of top-down design dictate that a program should be divided into a main module and its related modules. Each module should also be divided into sub modules according to software engineering and programming style. The division of modules processes until the module consists only of elementary process that are intrinsically understood and cannot be further subdivided.



Top-down algorithm design is a technique for organizing and coding programs in which a hierarchy of modules is used, and breaking the specification down into simpler and simpler pieces, each having a single entry and a single exit point, and in which control is passed downward through the structure without unconditional branches to higher levels of the structure. That is top-down programming tends to generate modules that are based on functionality, usually in the form of functions or procedures or methods.

Bottom-Up modular programming

Bottom-up algorithm design is the opposite of top-down design. It refers to a style of programming where an application is constructed starting with existing primitives of the programming language, and constructing gradually more and more complicated features, until the all of the application has been written. That is, starting the design with specific modules and build them into more complex structures, ending at the top. The bottom-up method is widely used for testing, because each of the lowest-level functions is written and tested first. This testing is done by special test functions that call the low-level functions, providing them with different parameters and examining the results for correctness. Once lowest-level functions have been tested and verified to be correct, the next level of functions may be tested. Since the lowest-level functions already have been tested, any detected errors are probably due to the higher-level functions. This process continues, moving up the levels, until finally the *main* function is tested.



3.Structured Programming

It is a programming style; and this style of programming is known by several names: Procedural decomposition, Structured programming, etc. Structured programming is not programming with structures but by using following types of code structures to write programs:

1. Sequence of sequentially executed statements
2. Conditional execution of statements (*i.e.*, "if" statements)
3. Looping or iteration (*i.e.*, "for, do...while, and while" statements)
4. Structured subroutine calls (*i.e.*, functions)

In particular, the following language usage is forbidden:

- "GoTo" statements
- "Break" or "continue" out of the middle of loops
- Multiple exit points to a function/procedure/subroutine (*i.e.*, multiple "return" statements)
- Multiple entry points to a function/procedure/subroutine/method

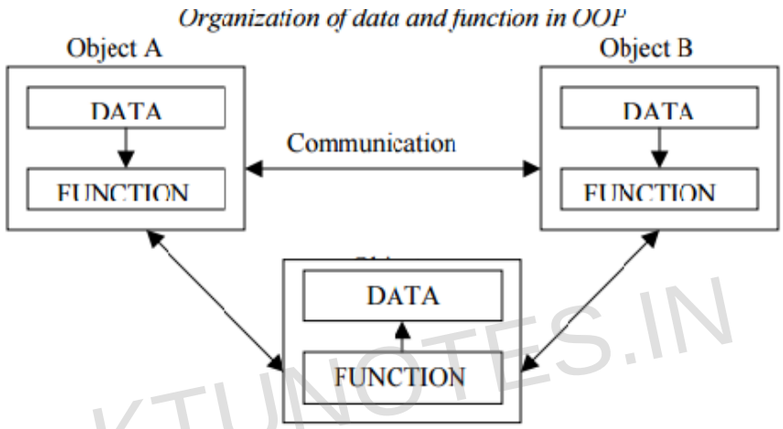
In this style of programming there is a great risk that implementation details of many data structures have to be shared between functions, and thus globally exposed. This in turn tempts other functions to use these implementation details; thereby creating unwanted dependencies between different parts of the program. The main disadvantage is that all decisions made from the start of the project depend directly or indirectly on the high-level specification of the application. It is a well known fact that this specification tends to change over a time. When that happens, there is a great risk that large parts of the application need to be rewritten.

Advantages of structured programming

1. clarity: structured programming has a clarity and logical pattern to their control structure and due to this tremendous increase in programming productivity
2. another key to structured programming is that each block of code has a single entry point and single exit point.so we can break up long sequence of code into modules
3. Maintenance: the clarity and modularity inherent in structured programming is of great help in finding an error and redesigning the required section of code.

4.Obect oriented programming

The major motivating factor in the invention of object-oriented approach is to remove some of the flaws encountered in the procedural or modular approach. OOP treats data as a critical element in the program development and does not allow it to flow freely around the system. It ties data more closely to the function that operate on it, and protects it from accidental modification from outside function. OOP allows decomposition of a problem into a number of entities called objects and then builds data and function around these objects. The organization of data and function in object-oriented programs is shown in fig. The data of an object can be accessed only by the function associated with that object. However, function of one object can access the function of other objects.



Some of the features of object oriented programming are:

- Emphasis is on data rather than procedure.
- Programs are divided into what are known as objects.
- Data structures are designed such that they characterize the objects.
- Functions that operate on the data of an object are ties together in the data structure.
- Data is hidden and cannot be accessed by external function.
- Objects may communicate with each other through function.
- New data and functions can be easily added whenever necessary.
- Follows bottom up approach in program design.

Analysis of Algorithm

After designing an algorithm, it has to be checked and its correctness needs to be predicted; this is done by analyzing the algorithm. The algorithm can be analyzed by tracing all step-by-step instructions, reading the algorithm for logical correctness, and testing it on some data using mathematical techniques to prove it correct. Another type of analysis is to analyze the simplicity of the algorithm. That is, design the algorithm in a simple way so that it becomes easier to be implemented. However, the simplest and most Straight forward way of solving a problem may not be sometimes the best one. Moreover there may be more than one algorithm to solve a problem. The choice of a particular algorithm depends on following performance analysis and measurements:

1. Space complexity
2. Time complexity

Space Complexity

Analysis of space complexity of an algorithm or program is the amount of memory it needs to run to completion. Some of the reasons for studying space complexity are:

1. If the program is to run on multi user system, it may be required to specify the amount of memory to be allocated to the program.

2. We may be interested to know in advance that whether sufficient memory is available to run the program.

3. There may be several possible solutions with different space requirements.

4. Can be used to estimate the size of the largest problem that a program can solve.

The space needed by a program consists of following components.

• *Instruction space* : Space needed to store the executable version of the program and it is fixed.

• *Data space* : Space needed to store all constants, variable values and has further two components :

(*a*) Space needed by constants and simple variables. This space is fixed.

(*b*) Space needed by fixed sized structural variables, such as arrays and structures.

(*c*) Dynamically allocated space. This space usually varies.

• ***Environment stack space****:* This space is needed to store the information to resume the suspended (partially completed) functions. Each time a function is invoked the following data is saved on the environment stack :

(*a*) Return address : *i.e.*, from where it has to resume after completion of the Called function.

(*b*) Values of all lead variables and the values of formal parameters in the function being invoked.

The amount of space needed by recursive function is called the recursion stack space. For each recursive function, this space depends on the space needed by the local variables and the formal parameter. In addition, this space depends on the maximum depth of the recursion *i.e.*, maximum number of nested recursive calls.

Time Complexity

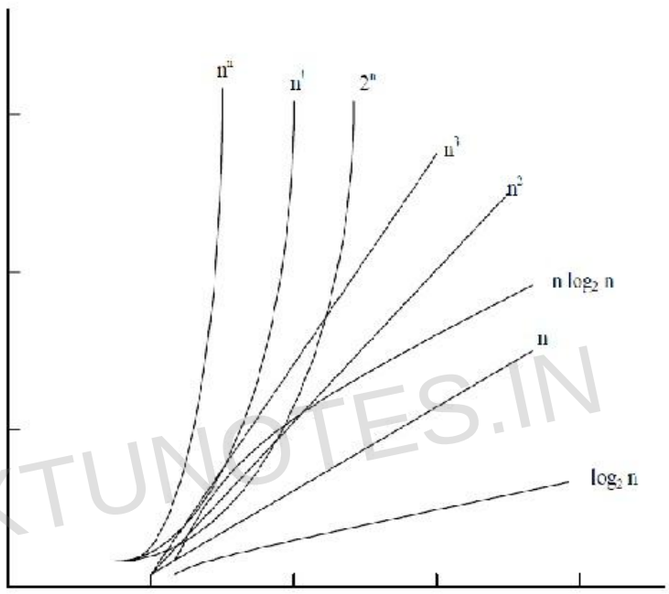
The time complexity of an algorithm or a program is the amount of time it needs to run to completion. The exact time will depend on the implementation of the algorithm, programming language, optimizing the capabilities of the compiler used, the CPU speed, other hardware characteristics/specifications and so on. To measure the time complexity accurately, we have to count all sorts of operations performed in an algorithm. If we know the time for each one of the primitive operations performed in a given computer, we can easily compute the time taken by an algorithm to complete its execution. This time will vary from machine to machine. By analyzing an algorithm, it is hard to come out with an exact time required. To find out exact time complexity, we need to know the exact instructions executed by the hardware and the time required for the instruction. The time complexity also depends on the amount of data inputted to an algorithm. But we can calculate the order of magnitude for the time required.

The time complexity also depends on the amount of data input to an algorithm, but we can calculate the order of magnitude for the time required. That is, our intention is to estimate the execution time of an algorithm irrespective of the computer machine on which it will be used.

Some of the reasons for studying time complexity are

a) We may be interest to know in advance that whether an algorithm or program will provide a satisfactory real time response
b) There may be several possible solutions with different time requirements

Here, the more sophisticated method is to identify the key operations and count such operations performed till the program completes its execution. A key operation in our algorithm is an operation that takes maximum time among all possible operations in the algorithm. Such an abstract, theoretical approach is not only useful for discussing and comparing algorithms, but also it is useful to improve solutions to practical problems. The time complexity can now be expressed as function of number of key operations performed. Before we go ahead with our discussions, it is important to understand the rate growth analysis of an algorithm, as shown in Figure.



The function that involves '*n*' as an exponent, *i.e.*, 2^n, n^n, $n!$ are called exponential functions, which is too slow except for small size input function where growth is less than or equal to n^c,(where '*c*' is a constant) *i.e.*; n^3, n^2, $n \log_2 n$, n, $\log_2 n$ are said to be polynomial. Algorithms with polynomial time can solve reasonable sized problems if the constant in the exponent is small.

When we analyze an algorithm it depends on the input data, there are three cases :

1. Best case
2. Average case
3. Worst case

In the best case, the amount of time a program might be expected to take on best possible input data.

In the average case, the amount of time a program might be expected to take on typical (or average) input data.

In the worst case, the amount of time a program would take on the worst possible input configuration.

Frequency Count

Frequency count method can be used to analyze a program .Here we assume that every statement takes the same constant amount of time for its execution. Hence the determination of time complexity of a given program is is the just matter of summing the frequency counts of all the statements of that program Consider the following examples

…….	for(i=0;I,n;i++)	for(i=0;i<n;i++)
…….	X++;	for(j=0;j<n;j++)
X++;	……..	x++;
(a)	(b)	(c)

In the example (a) the statement x++ is not contained within any loop either explicit or implicit. Then its frequency count is just one. In example (b) same element will be executed n times and in example (3) it is executed by n^2. From this frequency count we can analyze program

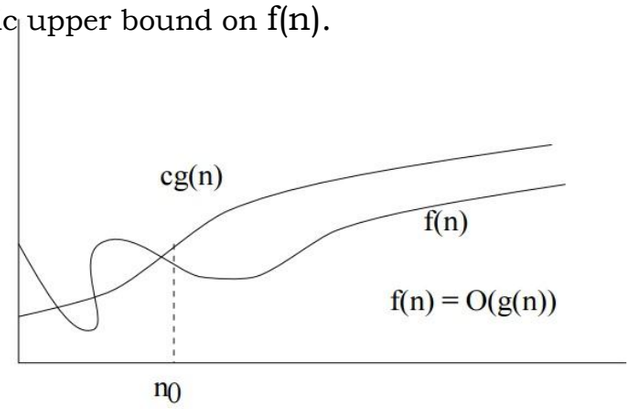
Growth of Functions and Asymptotic Notation

When we study algorithms, we are interested in characterizing them according to their efficiency. We are usually interesting in the order of growth of the running time of an algorithm, not in the exact running time. This is also referred to as the asymptotic running time. We need to develop a way to talk about rate of growth of functions so that we can compare algorithms. Asymptotic notation gives us a method for classifying functions according to their rate of growth.

Big-O Notation

Definition:

$f(n) = O(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \leq c\ |g(n)|$ for all $n \geq n_0$. If f(n) is nonnegative, we can simplify the last condition to $0 \leq f(n) \leq c\ g(n)$ for all $n \geq n_0$. then we say that **"f(n) is big-O of g(n)."** . As n increases, $f(n)$ grows n_o faster than $g(n)$. In other words, g(n) is an asymptotic upper bound on $f(n)$.



Example: $n^2 + n = O(n^3)$

Proof: • Here, we have $f(n) = n^2 + n$, and $g(n) = n^3$

• Notice that if $n \geq 1$, $n \leq n^3$ is clear.

• Also, notice that if $n \geq 1$, $n^2 \leq n^3$ is clear.

• In general, if $a \leq b$, then $n^a \leq n^b$ whenever $n \geq 1$. This fact is used often in these types of proofs.

• Therefore, $n^2 + n \leq n^3 + n^3 = 2n^3$

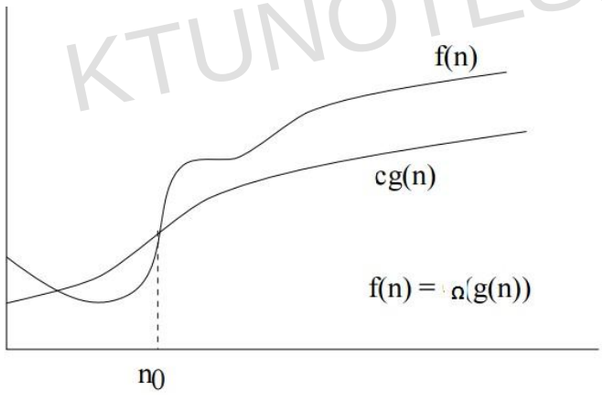
• We have just shown that $n^2 + n \leq 2n^3$ for all $n \geq 1$

• Thus, we have shown that $n^2 + n = O(n^3)$ (by definition of Big-O, with $n_0 = 1$, and $c = 2$.)

Big-Ω notation

Definition:

$f(n) = \Omega(g(n))$ iff there are two positive constants c and n_0 such that $|f(n)| \geq c\ |g(n)|$ for all $n \geq n0$ .If $f(n)$ is nonnegative, we can simplify the last condition to $0 \leq c\ g(n) \leq f(n)$ for all $n \geq n0$ • then we say that **"f(n) is omega of g(n)."** • As n increases, $f(n)$ grows no slower than $g(n)$. In other words, $g(n)$ is an asymptotic lower bound on $f(n)$



Example: $n^3 + 4n^2 = \Omega(n^2)$

Proof: • Here, we have $f(n) = n^3 + 4n^2$, and $g(n) = n^2$

• It is not too hard to see that if $n \geq 0$, $n^3 \leq n^3 + 4n^2$

• We have already seen that if $n \geq 1$, $n^2 \leq n^3$

• Thus when $n \geq 1$, $n^2 \leq n^3 \leq n^3 + 4n^2$

• Therefore,

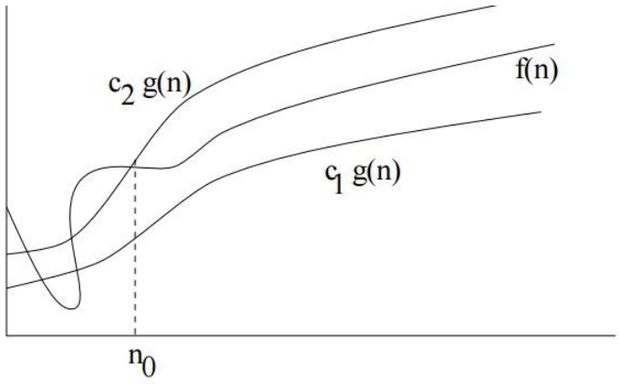
$1n^2 \leq n^3 + 4n^2$ for all $n \geq 1$

• Thus, we have shown that $n^3 + 4n^2 = \Omega(n^2)$ (by definition of Big-Ω, with $n_0 = 1$, and $c = 1$.)

Big-Θ notation

Definition:

$f(n) = \Theta(g(n))$ iff there are three positive constants c_1, c_2 and n_0 such that $c1|g(n)| \le |f(n)| \le c2|g(n)|$ for all $n \ge n_0$. If f(n) is nonnegative, we can simplify the last condition to $0 \le c1\ g(n) \le f(n) \le c2\ g(n)$ for all $n \ge n_0$ .then we say that **"f(n) is theta of g(n)."** . As n increases, f(n) grows at the same rate as g(n). In other words, g(n) is an asymptotically tight bound on f(n).



Example: $n^2 + 5n + 7 = \Theta(n^2)$

Proof: •

When $n \ge 1$, $n^2 + 5n + 7 \le n^2 + 5n^2 + 7n^2 \le 13n^2$

- When $n \ge 0$, $n^2 \le n^2 + 5n + 7$
- Thus, when $n \ge 1$

  $1n^2 \le n^2 + 5n + 7 \le 13n^2$

  Thus, we have shown that $n^2 + 5n + 7 = \Theta(n^2)$ (by definition of Big-Θ, with $n_0 = 1$, c1 = 1, and c2 = 13.)

Comparison of different Algorithm

Algorithm	Best case	Average case	Worst case
Quick sort	O(n log n)	O(n log n)	$O(n^2)$
Merge sort	O(n log n)	O(n log n)	O(n log n)
Heap sort	O(n log n)	O(n log n)	O(n log n)
Bubble sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	O(n)	$O(n^2)$	$O(n^2)$
Binary search	O(1)	O(log n)	O(log n)
Linear search	O(1)	O(n)	O(n)

MODULE II

Linear Data Structures (Linked list)

Data

Data means values or set of values. Data can be defined as representation of facts, concepts or instruction in a formalized manner which should be suitable for communication, interpretation or processing by human or machine.

Information

It is the organized or classified data . So that it has some meaningful values to the receiver. Information is the processed data on which decision and actions are performed.

Eg:-

Data	Information
16	Age of a person
	Distance between two places
28/9/1990	Age of a person
	Month of birth is September

Data Structure

In computer science data structure is a particular way of storing and organizing data in a computer, so that it can be use efficiently.

Data structure is an arrangement of data in computer memory or even disk storage (secondary memory).A data structure can be considered as a combination of data and a set of algorithms that stores and organize data in computer memory efficiently.

Thus data structure=Data +set of algorithms that stores and organize the data

Data structure perform the following

1. Storage representation of user data:-

   User data should be stored in a such a way that computer can understand it.

2. Retrieval of stored data

   The data stored in a computer should be retrieve in such a way that user can understand it.

3. Transformation of user data

   Various operations which require to be perform on user data, so that it can transform from one form to another

A data structure D is a triplet

D=(d,f,a)

Where d-→Domain

Domain is the range of values that a data made up. This domain is also term as data object.

f-→ functions

This is the set of operations which may be applied to the element of data object

a→Axioms

this is the set of rules with which different operations belongs to f actually can be implemented

Classification of Data Structure

Data structure can be classified into linear and nonlinear data structure

In case of linear data structure all the elements form a sequence and maintain a linear fashion. In case of nonlinear data structure elements are distributed over a plane, ie follow a nonlinear fashion.

Arrays

It is a collection of data of same data types stored in consecutive memory location that is referred by a common name

Linked List

It is a collection of data of same data type ,but the data items need not be stored in consecutive memory locations

Stack

A stack is Last in First Out(LIFO) data structure, here insertion and deletion takes place at one end called stack top. Here insertion is known as PUSH operation and deletion is known as POP operation

Queues

A queue is a First in First Out (FIFO) in which insertion takes place at real end and deletion takes place at front end. Insertion is known as **enqueue operation** and deletion is known as **dequeue operation.**

Trees

Trees are used to represent data that has some hierarchical relationship among data elements

Graphs

A graph is used to represent data that has relationship between pair of elements, not necessarily hierarchical

All trees are graphs, but all graphs are not trees

Tables

In table , data is arranged in no.of rows and columns

Sets

A set is a abstract data type that can store certain values without any particular order and without any repeatation.

Data types

→Data type is a term which refers to kind of data that may appear in computation.

→A data type is a classification for identifying one of the various type of data such as real numbers, integers, characters, Booleans etc

→Data types takes the possible values, the operation on that type, the way the value of that types are stored.

Built-in Data types

Data types which are built in with the programming language is called built-in data type/fundamental data type.

Eg:- the fundamental data type in C are int, float, char, double

Data type	**Number of bytes**	**Range of values**
char	1 byte	-2^7 to 2^7-1
int	2 byte	-2^{15} to $2^{15}-1$

unsigned int	2 byte	0 to 2^{16}-1
signed int	2 byte	-2^{15} to 2^{15} -1
short int	1 byte	-2^{7} to 2^{7}-1
long int	4 bytes	-2^{31} to 2^{31} -1
float	4 byte	3.4 e $^{-38}$ to 3.4e $^{+38}$
double	8 byte	1.7 e $^{-308}$ to 1.7e $^{+308}$
long double	10 bytes	3.4 e $^{-4932}$ to 1.1e^{+4932}

Fundamental data type in Fortran Language:-
integer, logical, character, complex, double precision, single precision

Fundamental data type in Pascal Language:-
real, character, boolean, array, record, file

Fundamental data type in Java Language:-
int, float, long, byte,etc

Scalar Data type

→A scalar data type is a single unit of data
→A scalar data type is a single valued data type that can be used for individual variable, constant etc.
→A scalar data type is atomic:-it is not made up of other variable components
→Hold a single value
→ Have no internal components
→The range of values depends on hardware architecture
Eg:- character, number, Boolean etc

Primitive Data type

In computer science primitive data type referred to either the following concepts

1. Basic type:- it is the data type provided by programming language as a built in block
   Most languages allow more complicated composite type to be recursively constructed starting from basic type.
2. Built-in type:- is a data type for which the programming language provides built-in support

→ In most programming languages, all basic data types are built-in.
→ In addition, many languages also provide a set of composite data types

→ Depending on the language and its implementation, primitive data types may or may not have a one-to-one correspondence with objects in the computer's memory

→Actual range of primitive data type depends upon the programming language

Typical primitive types are

1. Character
2. Integer
3. Floating point number
4. Fixed point number(fixed)
5. Reference(pointer,handle)

More complex built-in type include

→tuple in matlab or python

→linked list in LISP

→complex number in Fortran ,LISP,python

→rational number in LISP

Enumerated Data type

In computer programming, an enumerated type (also called enumeration or enum) is a data type consisting of a set of named values called elements, members, enumeral, or enumerators of the type. The enumerator names are usually identifiers that behave as constants in the language.

A variable that has been declared as having an enumerated type can be assigned any of the enumerators as a value. Some enumerator types may be built into the language. The Boolean type, for example is often a pre-defined enumeration of the values *FALSE* and *TRUE*. Many languages allow the user to define new enumerated types.

In C enumerators are created by keyword **enum**

Eg:-

enum fruit{ mango,apple,……};--→declaration of fruit data type

enum fruit x; --→declaration of variable x of data type fruit

x=mango;

x=apple; -→ assigning values to variable x

C also allows programmers to choose the value of enumeration constants explicitly.

enum fruit{ mango=4,apple,……};

Enumerated data type in Visual Basic can be defined as follows

```
Enum fruit
        Apple
        Mango
        .
        .
        .
End Enum
```

Subranges

Subrange facility of a programming language offers to assign range of values to variables of different data types

Eg:- subrange facility in PASCAL type

```
Tsmall:=0..9;
Var
    X:Tsmall;
begin
    x:=7;
    x:=25;
end
```


here the data type Tsmall takes the value from 0 to 9.The statement x:=7 works correctly. But x:=25 will not work, because 25 is out of these subrange. Here subrange is represented by **..**

Records

In computer science a record (structure, or tuple) is a data type consisting of 2 or more variables stored in consecutive memory locations. so that each components (called field or member of the record) can be accessed by applying different offset to a single physical address. A record type is a data type that describe such values and variables.

Most modern computer languages allow programmer to define new record types. The definition includes specification of the data type of each field, its position in the record and identifier(name or label) by which it can be accessed.

Records can exist in any storage medium, including main memory and mass storage devices such as magnetic tapes or hard disks. Records

are a fundamental component of most data structures, especially linked data structures. Many computer files are organized as arrays of logical records

A programming language that support record type usually provide the following

1. Declaration of a new record type, including the position, type, and (possibly) name of each field;
2. Declaration of variables and values as having a given record type
3. Construction of a record value from given field values
4. Selection of a field of a record with an explicit name
5. Assignment of a record value to a record variable
6. Comparison of two records for equality

Linked list

It is a data structure where the amount of memory required can be varied. A linked list is an ordered collection of finite, homogeneous data elements called nodes. Where the linear order is maintained by means of links or pointer. Ie, in a linked list adjacency is between the elements are maintained by means of pointers. A link or pointer actually stores the address of subsequent element.

An element in a linked list specially termed as nodes. A node consist of two fields. 1) data field:- to store actual information 2) link field:- to points to the next node.

The structure of node

The representation of a node can be coded as

```
struct node
{
        int data;
        struct node *link;
};
```

Consider the following example

START is a pointer which stores the address of the first node. START requires only 2 bytes because it stores only an address. Consider the first node, it contain its information 1 in data field and address of the next node (2000) in link field and so on. The last node contains only information field and link field contain NULL value. The NULL value indicates the end of the linked list.

Memory Representation

Advantages of Linked List

→Linked list is a dynamic data structure, which can grow and shrink during the execution of program

→Efficient memory utilization, ie memory is allocated whenever it is required and it is de allocated when it is not needed.

→Insertion and deletion are easier

→Many complex application can be easily carried out with linked list.

Disadvantages

→It is required more memory to store information

→Access to particular element is time consuming

→Extra care must needed for all the operation in linked list.

Operations

1)creation 2)insertion 3)searching 4)deletion 5)traversing 6) concatenation

1) Creation operation:- it is used to create a linked list
2) Insertion operation is used to insert a new node at specified location in a linked list. A new node may be inserted a) at beginning of linked list b)at the end of the linked list c) at any specified position of linked list.
3) Deletion is used to delete a particular node from any specified location, it may be a) from beginning b)from end c) from at any specified position.
4) Traversing is the process of going through all the nodes from one end to another end of the linked list exactly once.
5) Searching is the process of searching a particular key in the linked list.
6) Concatenation is the process of appending second linked list to the end of first linked list.

Classification of Linked list

1. Singly linked list
2. Doubly linked list
3. Circular linked list

Singly linked list

In singly linked list each node contains only one link field which points to the next node in linked list. Here START is a pointer which points to the first node of linked list. So , starting from first node one can reach to the last node whose link field does not contain any address rather than a NULL value. In a linked list one can move from left to right only. So it is also termed as one way list.

Insertion Operation

1)**Insert node at beginning of Linked list**

insert_begin_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the beginning of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print “node is not created, insertion is not possible”
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL

3. TEMP->link=START
4. START=TEMP

5. End If
6. stop

Explanation

Here we will first create an empty node by means of dynamic memory allocation. if node is created ,the memory allocation function will return the starting address of the node , otherwise returns a NULL pointer. Then we will fill data field of the newly created node as key and link field as the starting address of the old linked list.

2)Insert node at the end of Linked list

insert_end_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the end of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print "node is not created, insertion is not possible"
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL
    3. If START=NULL then
        1. START=TEMP
    4. Else
        1. TEMP1=START
        2. While TEMP1->link !=NULL do
            1. TEMP1=TEMP1->link
        3. End While
        4. TEMP1->link=TEMP
    5. End If
5. End If
6. stop

Explanation

The algorithm is used to insert a new node at the end of linked list. For this we have to create an empty node, then fill data field and link field of the newly created node and store its starting address to a pointer TEMP. Insertion at the end of linked list consist of two parts

1. for inserting a node into an empty linked list
2. inserting new node at the end of already existing linked list

for the first case, we simply assign the address of START as address of newly created node. For the second case we have to traverse linked list to last node using a pointer TEMP1. Then change the link field of the last node as address of the newly created node.

3)Insert node at the specified position of Linked list

insert_position_sll(START,key,position)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted at position

O/p: a new node is inserted at the specified position

Data Structure: singly linked list

Algorithm

1. Start
2. If position <= 0 then
    1. Print “position is invalied”
3. else
    1. Create a node and store its starting address to TEMP
        1. If TEMP=NULL then
            1. Print “node is not created, insertion is not possible”
        2. Else
            1. TEMP->data=Key
            2. TEMP->link=NULL
            3. If position=1 then
                1. TEMP->link=START
                2. START=TEMP
        3. Else
            1. TEMP1=START
            2. Count=1
            3. While count<position-1 and TEMP1!=NULL do
                1. TEMP1=TEMP1->link
                2. Count=count+1
            4. End While

5. If TEMP1=NULL then
    1. Print “linked list is out of range”
6. Else
    1. TEMP->link=TEMP1->link
    2. TEMP1->link=TEMP
7. End If

4. End if

4. End if
5. stop

Deletion Operation

1)Delete a node from the beginning of Linked list

delete_begin_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from the beginning of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print “ deletion is not possible, linked list is empty”
3. Else
    1. TEMP=START
    2. START=START->link
    3. Delete node pointed by TEMP
4. End if
5. Stop

2)Delete a node from the end of Linked list

delete_end_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from the end of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print “ deletion is not possible, linked list is empty”
3. Else

1. TEMP=START
2. PREVE=START
3. While TEMP->link!=NULL do
    1. PREVE=TEMP
    2. TEMP=TEMP->link
4. End while
5. If TEMP=PREVE
    1. START=NULL
6. Else
    1. PREVE->link=NULL
7. End if
8. Delete node pointed by TEMP

4. End if
5. Stop

3) Delete a node from any position of Linked list

delete_position_sll(START,position)

i/p: a singly linked list whose starting address is pointed by START, position where node is to be delete

O/p: a node is deleted from the specified position of linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If position<=0 then
    1. Print " position must be positive value"
3. Else
    1. If START=NULL then
        1. Print " linked list is empty, deletion is not possible"
    2. else
        1. if position=1 then
            1. TEMP=START
            2. START=START->link
        2. Else
            1. TEMP=START
            2. PREVE=START
            3. Count=1
            4. While count< position and TEMP !=NULL do
                1. PREVE=TEMP
                2. TEMP=TEMP->link

5. End while
6. If TEMP=NULL then
    1. Print “Linked list is out of range”
7. Else
    1. PREVE->link=TEM->link
8. End if
9. Delete node pointed by TEMP

3. End if

4. End if
5. Stop

Traversing of Linked List

traverse_sll(START)

i/p: a singly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: singly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print “Linked list is empty”
4. Else
    1. While TEMP!=NULL then
        1. print TEMP->data
        2. TEMP=TEMP->link
    2. End while
5. End If
6. stop

Searching a key in Linked List

search_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, key to be search

O/p: print present or not

Data Structure: singly linked list

Algorithm

1. Start

2. TEMP=START
3. Flag=0
4. If TEMP=NULL then
    1. Print “Linked list is empty”
5. Else
    1. While TEMP!=NULL and flag=0 then
        1. If TEMP->data=key then
            1. Flag=1
        2. End if
        3. TEMP=TEMP->link
    2. End while
    3. If flag=0 then
        1. Print “key is not found”
    4. Else
        1. Print “key is found”
    5. End If
6. End if
7. stop

concatenation of two linked list

concatinate_sll(START1,START2)

i/p: two singly linked lists, whose starting node are pointed by START1, START2 respectively

O/p: second linked list is concatenated at the end of first linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START1=NULL then
    1. START1=START2
3. Else
    1. TEMP=START1
    2. While TEMP->link !=NULL do
        1. TEMP=TEMP->link
    3. End while
4. TEMP->link=START2
5. stop

Insertion of nodes in ascending order

insert_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START, Key is the data to be inserted in ascending order

O/p: a new node is inserted in ascending order

Data Structure: singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print "node is not created, insertion is not possible"
4. Else
    1. TEMP->data=Key
    2. TEMP->link=NULL
    3. TEMP1=START
    4. PREV=START
    5. While TEMP1 !=NULL and key >TEMP1->data do
        1. PREV=TEMP1
        2. TEMP1=TEMP1->link
    6. End while
    7. If TEMP1=PREV then
        1. TEMP->link=START
        2. START=TEMP
    8. else
        1. TEMP->link=TEMP1
        2. PREV->link=TEMP
    9. End If
5. End if
6. stop

Deletion of a node from a ordered list

delete_sll(START,key)

i/p: a singly linked list whose starting address is pointed by START

O/p: a node is deleted from ordered singly linked list

Data Structure: singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print "Linked list is empty"

3. else
    1. TEMP=START
    2. PREV=START
    3. While TEMP !=NULL and TEMP1->data !=key do
        1. PREV=TEMP
        2. TEMP=TEMP->link
    4. End while
    5. If TEMP=NULL then
        1. Print “key is not found”
    6. else
        1. if PREV=TEMP then
            1. START=START->link
        2. Else
            1. PREV->link=TEMP->link
        3. End If
        4. Delete node pointed by TEMP
    7. End if
4. End if
5. stop

Doubly Linked List

a doubly linked list is one in which are nodes are linked together by multiple links which helps to access both successor (next) and predecessor (previous) nodes from any arbitrary position.

In a singly linked list one can move from first node to any node in one direction only(from left to right). So a singly linked list is also called one way list. In other hand a doubly linked list is a two way list because one can move in either direction from left to right or right to left. This is accomplished by maintaining two link field called NEXT and PREV

Node Structure:-

Here PREV points to the left side of the node(previous node) and NEXT points to the right side of the node (successor). Each node consisting of a data field where the actual information is stored.

Node structure can be coded as

```
struct node
{
        struct node *prev;
        int data;
        struct node *next;
};
```

Operations

1)creation 2)insertion 3)searching 4)deletion 5)traversing 6) concatenation

Insertion Operation

1)Insert node at beginning of doubly Linked list

insert_begin_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the beginning of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print “node is not created, insertion is not possible”
4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. If START=NULL then
        1. START=TEMP
    5. Else
        1. TEMP->NEXT=START
        2. START->PREV=TEMP
        3. START=TEMP
    6. End If
5. End if
6. stop

2)Insert node at end of doubly Linked list

insert_end_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted at the end of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print "node is not created, insertion is not possible"
4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. If START=NULL then
        1. START=TEMP
    5. Else
        1. TEMP1=START
        2. While TEMP1->NEXT !=NULL
            1. TEMP1=TEMP1->NEXT
        3. End while
        4. TEMP1->NEXT=TEMP
        5. TEMP->PREV=TEMP1
    6. End If
5. End if
6. stop

3)Insert node at specified position of doubly Linked list

insert_position_dll(START,key,pos)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted at pos

O/p: a new node is inserted at specified position of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If pos<=0 then
    1. Print “position must be positive”
3. Else
    1. Create a node and store its starting address to TEMP
    2. If TEMP=NULL then
        1. Print “node is not created, insertion is not possible”
    3. Else
        1. TEMP->PREV=NULL
        2. TEMP->data=Key
        3. TEMP->NEXT=NULL
        4. If pos=1 then
            1. TEMP->NEXT=START
            2. If START !=NULL
                1. START->PREV=TEMP
            3. End if
            4. START=TEMP
        5. Else
        6. TEMP1=START
        7. Count=1
        8. While count<pos-1 and TEMP1!=NULL do
            1. TEMP1=TEMP1->NEXT
            2. Count=count+1
        9. End while
        10. If TEMP1=NULL then
            1. Print “DLL is out of range”
        11. Else
            1. If TEMP1->NEXT=NULL then
                1. TEMP1->NEXT=TEMP
                2. TEMP->PREV=TEMP1
            2. Else
                1. TEMP->NEXT=TEMP1->NEXT
                2. TEMP1->NEXT->PREV=TEMP
                3. TEMP1->NEXT=TEMP
                4. TEMP->PREV=TEMP1
            3. End if
        12. End if
    4. End If
4. End if
5. stop

Deletion Operation

1)Delete a node from the beginning of doubly Linked list

delete_begin_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: a node is deleted from the beginning of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print "DLL is empty"
3. Else
    1. TEMP=START
    2. If START->NEXT=NULL then
        1. START=NULL
    3. Else
        1. START=START->NEXT
        2. START->PREV=NULL
    4. End if
    5. Delete node pointed by TEMP
4. End if
5. stop

2)Delete a node from the end of doubly Linked list

delete_end_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: a node is deleted from the end of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print "DLL is empty"
3. Else
    1. TEMP=START
    2. If START->NEXT=NULL then
        1. START=NULL
    3. Else
        1. While TEMP->NEXT !=NULL then
            1. TEMP=TEMP->NEXT
        2. End while
        3. TEMP->PREV->NEXT=NULL

4. Delete node pointed by TEMP

4. End if

4. End if

5. stop

3)Delete a node from the specified position of doubly Linked list

delete_position_dll(START,pos)

i/p: a doubly linked list whose starting address is pointed by START, pos from node to be deleted

O/p: a node is deleted from the specified position of doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print "DLL is empty"
3. Else
    1. If pos<=0 then
        1. Print "position must be positive"
    2. else
        1. TEMP=START
        2. If pos =1 then
            1. If START->NEXT=NULL then
                1. START=NULL
            2. Else
                1. START=START->NEXT
                2. START->PREV=NULL
            3. End if
        3. else
            1. TEMP=START
            2. Count=1
            3. While count <pos and TEMP !=NULL then
                1. TEMP=TEMP->NEXT
                2. Count=count+1
            4. End while
            5. If TEMP=NULL then
                1. Print "DLL is out of range"
            6. Else
                1. If TEMP->NEXT=NULL then
                    1. TEMP->PREV->NEXT=NULL

2. Else
    1. TEMP->PREV->NEXT=TEMP->NEXT
    2. TEMP->NEXT->PREV=TEMP->PREV
3. End if

7. End if

4. End if

5. Delete node pointed by TEMP

3. End if

4. End if

5. stop

traverse_dll(START)

i/p: a doubly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: doubly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print “DLL is empty”
4. Else
    1. While TEMP!=NULL then
        1. print TEMP->data
        2. TEMP=TEMP->NEXT
    2. End while
5. End If
6. stop

Insert a node to an ordered Doubly Linked List

insert_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: key inserted in ascending order

Data Structure: doubly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print “node is not created, insertion is not possible”

4. Else
    1. TEMP->PREV=NULL
    2. TEMP->data=Key
    3. TEMP->NEXT=NULL
    4. TEMP1=START
    5. While TEMP1->NEXT !=NULL and key >TEMP1->data do
        1. TEMP1=TEMP1->NEXT
    6. End while
    7. If TEMP1=START then
        1. If START=NULL then
            1. START=TEMP
        2. Else
            1. TEMP->NEXT=START
            2. START->PREV=TEMP
            3. START=TEMP
        3. End if
    8. else
        1. if TEMP1->NEXT=NULL
            1. TEMP1->NEXT=TEMP
            2. TEMP->PREV=TEMP1
        2. Else
            1. TEMP->PREV=TEMP1->PREV
            2. TEMP1->PREV->NEXT=TEMP
            3. TEMP->NEXT=TEMP1
            4. TEMP1->PREV=TEMP
        3. End if

    9. End If
5. End if
6. stop

Delete a node from an ordered Doubly Linked List

delete_dll(START,key)

i/p: a doubly linked list whose starting address is pointed by START, Key is the data to be deleted

O/p: key is deleted from ordered doubly linked list

Data Structure: doubly linked list

Algorithm

1. Start
2. If START=NULL then

1. Print “doubly linked list is empty”

3. Else
    1. TEMP1=START
    2. While TEMP1 !=NULL and key !=TEMP1->data do
        1. TEMP1=TEMP1->NEXT
    3. End while
    4. If TEMP1=NULL then
        1. Print “ key is not found”
    5. else
        1. If TEMP1=START then
            1. If START->NEXT=NULL then
                1. START=NULL
            2. Else
                1. START=START->NEXT
                2. START->PREV=NULL
            3. End if
        2. else
            1. if TEMP1->NEXT=NULL
                1. TEMP1->PREV->NEXT=NULL
            2. Else
                1. TEMP1->PREV->NEXT=TEMP1->NEXT
                2. TEMP->NEXT->PREV=TEMP1->PREV
            3. End if
        3. End if
    6. End if
    7. Delete node pointed by TEMP1
4. End if
5. stop

Circular Linked List

A circular linked list is one which has no beginning and end. A singly linked list can be changed to a circular linked list by simply storing the address of the first node in the link field of the last node.

A circular doubly linked list can be form by storing the address of the first node in the NEXT field of the last node and address of the last node in the PREV field of first node.

Operations

Traversing of Circular Linked List

traverse_cll(START)

i/p: a circular singly linked list whose starting address is pointed by START

O/p: visit all nodes exactly once

Data Structure: circular singly linked list

Algorithm

1. Start
2. TEMP=START
3. If TEMP=NULL then
    1. Print "Circular Linked list is empty"
4. Else
    1. While TEMP->link!=START then
        1. print TEMP->data
        2. TEMP=TEMP->link
    2. End while
    3. Print TEMP->data
5. End If
6. stop

Insert node to Circular Linked list

Insert_cll(START,key)

i/p: a circular singly linked list whose starting address is pointed by START, Key is the data to be inserted

O/p: a new node is inserted to circular linked list

Data Structure: circular singly linked list

Algorithm

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then

1. Print “node is not created, insertion is not possible”

4. Else
    1. TEMP->data=key
    2. TEMP->link=TEMP
    3. If START=NULL then
        1. START=TEMP
    4. Else
        1. TEMP1=START
        2. While TEMP1->link !=START then
            1. TEMP1=TEMP1->link
        3. End while
        4. TEMP1->link=TEMP
        5. TEMP->link=START
    5. End if
5. end if
6. stop

Delete node from Circular Linked list

delete_cll(START,key)

i/p: a circular singly linked list whose starting address is pointed by START, Key is the data to be delete

O/p: key is deleted from circular linked list

Data Structure: circular singly linked list

Algorithm

1. Start
2. If START=NULL then
    1. Print “node is not created, insertion is not possible”
3. Else
    1. TEMP=START
    2. While TEMP->data != key and TEMP->link != START then
        1. PREV=TEMP
        2. TEMP=TEMP->link
    3. End while
    4. If TEMP->data=key then
        1. If TEMP=START then

1. If START->link=START then
    1. START=NULL
2. Else
    1. TEMP1=START
    2. While TEMP1->link != START then
        1. TEMP1=TEMP1->link
    3. End while
    4. TEMP1->link=START->link
    5. START=START->link
3. End if

2. else
    1. PREV->link=TEMP->link
3. End if
4. Delete node pointed by TEMP

5. else
    1. print “key is not found”

6. end if

4. end if
5. stop

Application of Linked List

1. polynomial representation
2. sparse matrix representation

Polynomial Representation

An important application of linked list is to represent polynomial and their manipulations. Main advantages of linked list for polynomial representation is that it can accommodate number of polynomial of growing size. So that the combined size does not exceed a total memory available.

Consider general form of polynomial f(x)=

Where is the term in the polynomial , a_i is the non-zero coefficient and e_i is the exponent. Here we will arrange the terms in descending order of exponent. Ie, $e_n > e_{n-1} > e_{n-2}.....>e_1$. Therefore the node structure of term in a polynomial contains the following fields

1. coefficient
2. Exponent
3. Link field

Where coeff represent the coefficient of term and exp represent the exponent and link represent the address of the next term

Insert a term into Linked List representation of Polynomial

Insert_term_poly(START,c,e)

i/p: START represent the starting address of the polynomial(address of the first term),c represent the coefficient of new term, e represent the exponent of the new term

O/p: a new term is added to polynomial

Data Structure: singly linked list

Algorithm

1 start
2. if START=NULL then
    1. create a node and store its starting address to TEMP
    2. if TEMP=NULL then
        1. print "node is not created ,term cannot be inserted"
    3. else
        1. TEMP->coeff=c
        2. TEMP->exp=e
        3. TEMP->link=NULL
        4. START=TEMP
    4. End if
3. Else
    1. TEMP1=START
    2. While TEMP1->exp > e and TEMP1 !=NULL do
        1. PREV=TEMP1
        2. TEMP1=TEMP1->link
    3. End while
    4. If TEMP1=NULL then
        1. create a node and store its starting address to TEMP
        2. if TEMP=NULL then
            1. print "node is not created ,term cannot be inserted"
        3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=NULL
4. PREV->link=TEMP

4. End if

5. Else

1. If TEMP1->exp=e then

1. TEMP1->coeff=TEMP1->coeff+c

2. Else

1. If TEMP1=START then

1. create a node and store its starting address to TEMP
2. if TEMP=NULL then

1. print “node is not created ,term cannot be inserted”

3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=START
4. START=TEMP

4. End if

2. Else

1. create a node and store its starting address to TEMP
2. if TEMP=NULL then

1. print “node is not created ,term cannot be inserted”

3. else

1. TEMP->coeff=c
2. TEMP->exp=e
3. TEMP->link=TEMP1
4. PREV->link=TEMP

4. End if

3. end if

3. end if

6. end if

4.end if

5.stop

Addition of two Polynomial

Polynomial_addition(START_P,START_Q,START_R)

i/p: START_P,START_Q represent the starting address of polynomial P and Q respectively. START_R represents the starting address of the resultant polynomial.

O/p: resultant polynomial R

Data Structure: singly linked list

Algorithm

1. start
2. START_R=NULL
3. P_PTR=START_P
4. Q_PTR=START_Q
5. While P_PTR !=NULL and Q_PTR!=NULL do
    1. If P_PTR->exp=Q_PTR->exp then
        1. C=P_PTR->coeff+Q_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
        5. Q_PTR=Q_PTR->link
    2. end if
    3. if P_PTR->exp > Q_PTR->exp then
        1. C=P_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
    4. end if
    5. if P_PTR->exp <Q_PTR->exp then
        1. C=Q_PTR->coeff
        2. e=Q_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. Q_PTR=Q_PTR->link
    6. end if
6. end while
7. while P_PTR != NULL do
        1. C=P_PTR->coeff
        2. e=P_PTR->exp
        3. insert_term_poly(START_R,c,e)
        4. P_PTR=P_PTR->link
8. end while

9. while Q_PTR != NULL do
    1. C=Q_PTR->coeff
    2. e=Q_PTR->exp
    3. insert_term_poly(START_R,c,e)
    4. Q_PTR=Q_PTR->link
10. end while
11. stop

Multiplication of two Polynomial

Polynomial_Product(START_P,START_Q,START_R)

i/p: START_P,START_Q represent the starting address of polynomial P and Q respectively. START_R represents the starting address of the resultant polynomial.

O/p: resultant polynomial R

Data Structure: singly linked list

Algorithm

1. start
2. START_R=NULL
3. P_PTR=START_P
4. Q_PTR=START_Q
5. If START_P=NULL OR START_Q=NULL then
    1. Print "multiplication is not possible"
6. Else
    1. While P_PTR!=NULL do
        1. Q_PTR=START_Q
        2. While Q_PTR!=NULL do
            1. C=P_PTR->coeff * Q_PTR->coeff
            2. e=P_PTR->exp+Q_PTR->exp
            3. insert_term_poly(START_R,c,e)
            4. Q_PTR=Q_PTR->link
        3. End while
        4. P_PTR=P_PTR->link
    2. End while
7. End if
8. stop

MODULE 5

SORTING TECHNIQUES

Sorting method can be classified into two.

1)Internal Sorting

2)External Sorting

In internal sorting the data to be sorted is placed in main memory. In external sorting the data is placed in external memory such as hard disk, floppy disk etc. The internal sorting can be classified into

1)n^2 sorting

2)n log n sorting

n^2 sorting - it can be classified into

1)Bubble sort

2) Selection sort

3) Insertion sort

nlogn sorting - It can be classified into

1) Merge sort

2)Quick sort

3) Heap sort

Bubble sort

Bubblesort(a[],n)

Input- an array a[size], n is the no. of element currently present in array

Output- Sorted array

DS- Array

Algorithm

1. Start
2. i=0
3. While i<n-1
    1. j=0
    2. While j<n-1-i
        1. If a[j]>a[j+1]
            1. temp=a[j]
            2. a[j]=a[j+1]
            3. a[j+1]=temp
        2. end if
        3. j=j+1

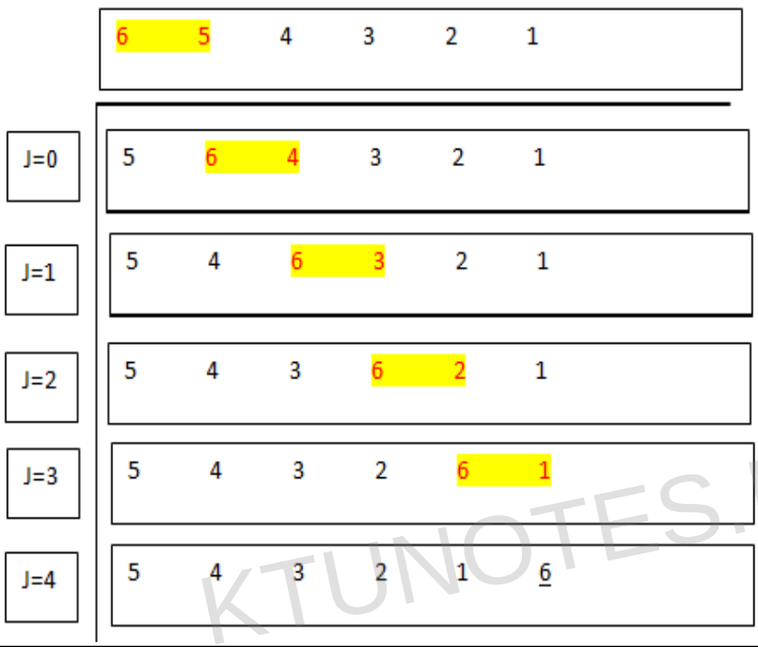
3.end while

4.i=i+1

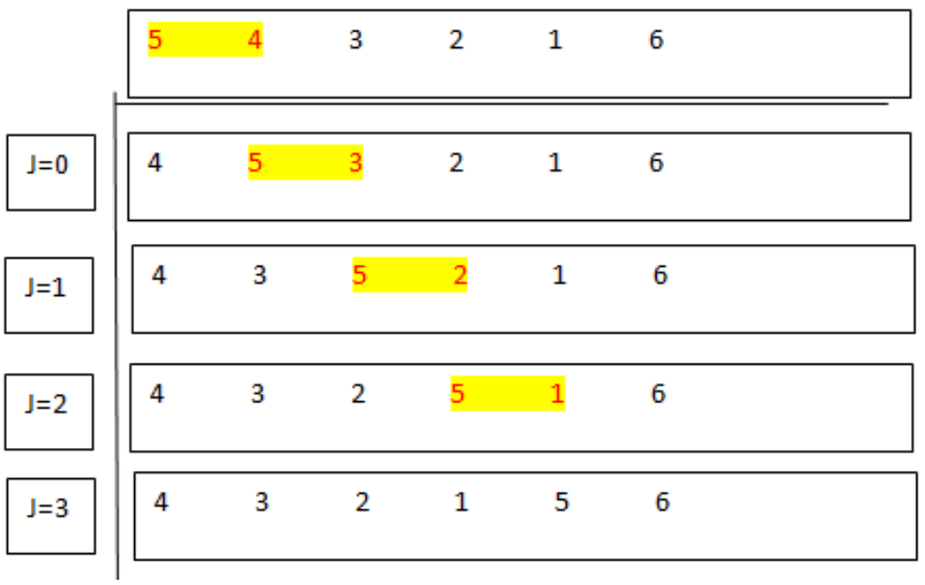
4. end while

Here first element is compared with second element. If first one is greater than second element then swap each other. Then second element is compared with third element . If second element is greater than third element then perform swapping. This process is continued until the comparison of (n-1)th element with nth element. These process continues (n-1) times. Consider the example-

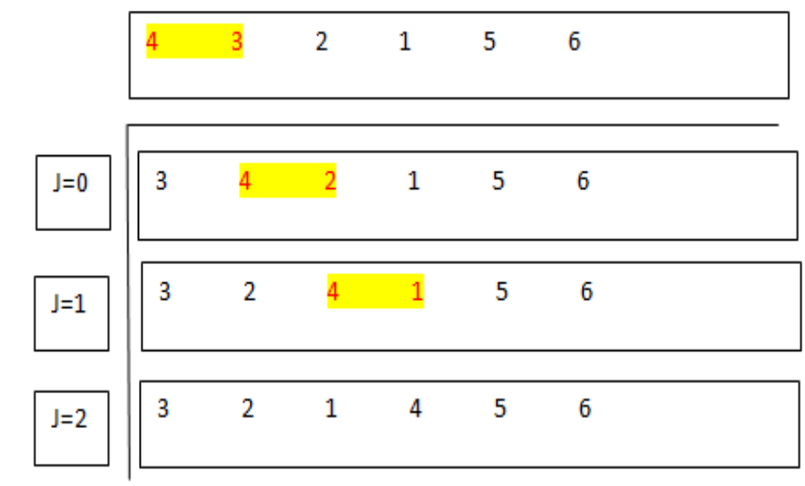
i=0



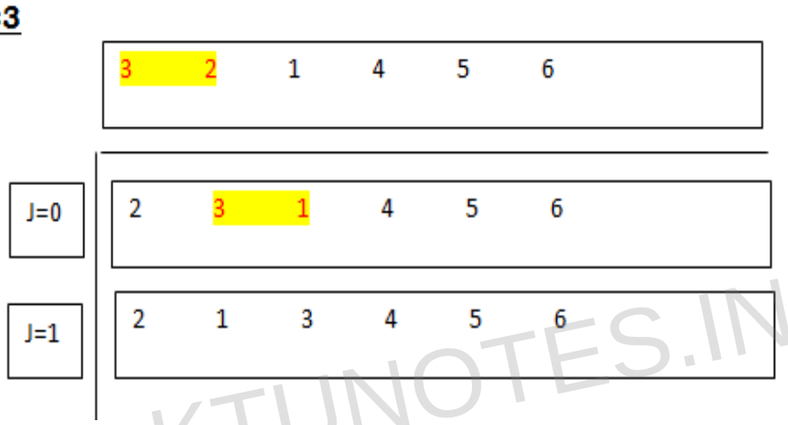
i=1



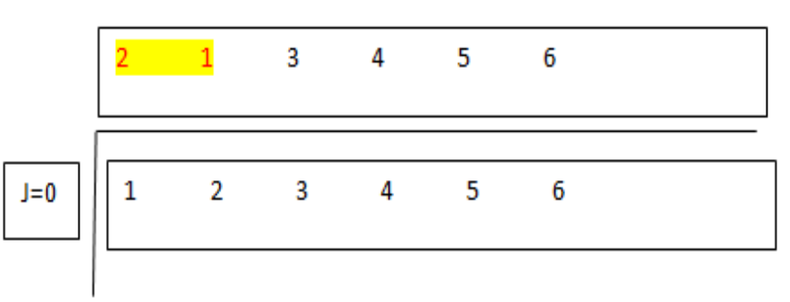
i=2



i=3



i=4



Thus i have values from 0 to n-1 and j have values from 0 to n-1-i.

Analysis

Here during the first iteration n-1 comparisons are required. During the second iteration n-2 comparisons are required etc..during last iteration, 1 comparison is required. Therefore total comparisons

= (n-1)+(n-2)+(n-3)+.....+1

$= \frac{n(n-1)}{2}$

=O(n^2)

SELECTION SORT

Selectionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

1: Start
2. i=0
3. while i<n-1 do
    1. j=i+1
    2. small=i
    3. while j<n do
        1.if a[small]>a[j]
            1. small=j
        2. end if
        3. j=j+1
        4. end while
        5. if i!=small
            1. temp=a[i]
            2. a[i]=a[small]
            3. a[small]=temp
        6. end if
        7. i=i+1
4. end while
5. stop

Here the first element in the array is selected as the smallest element. Then check for any element smaller than this from the second position to the last. If found any, then they are interchanged. Similarly next we check for the smallest element from third position to last. And the process continues upto (n-1)th position.

Analysis

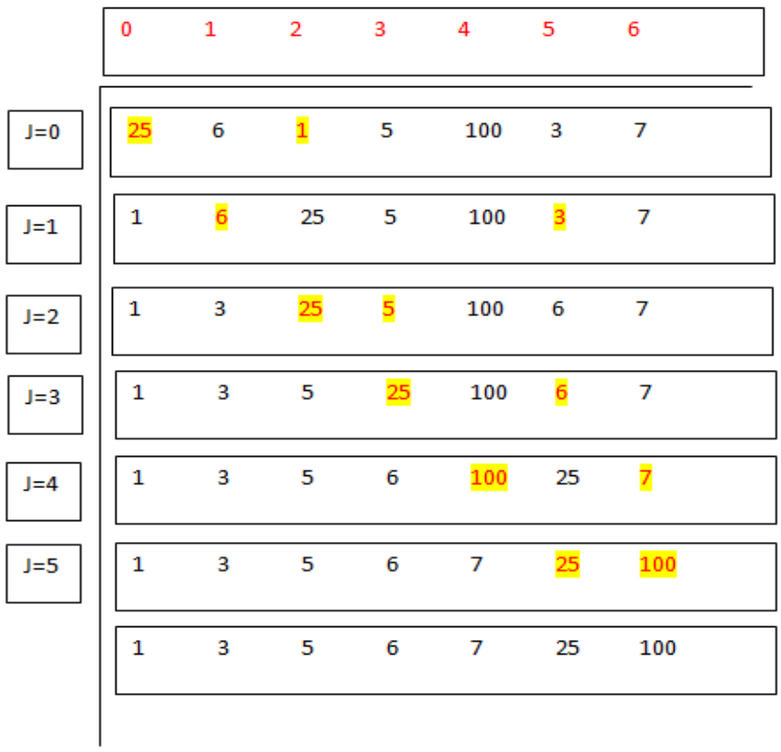
Here first iteration when i=0 takes n-1 comparisons, during second iteration takes n-2 comparison and so on.

Total no. of comparisons=(n-1)+(n-2)+(n-3)+.....+2+1

$= \frac{n(n-1)}{2}$

= O(n^2)

Example



INSERTION SORT

insertionsort(a[],n)

Input: An unsorted array a[], n is the no.of elements

Output: a sorted array

DS: Array

Algorithm

1. Start
2. i=1
3. While i<n
    1. j=i
    2. While a[j]<a[j-1] and j>0
        1. t=a[j]
        2. a[j]=a[j-1]
        3. a[j-1]=t
        4. j=j-1
    3. end while
    4. i=i+1

4.end while

5. stop

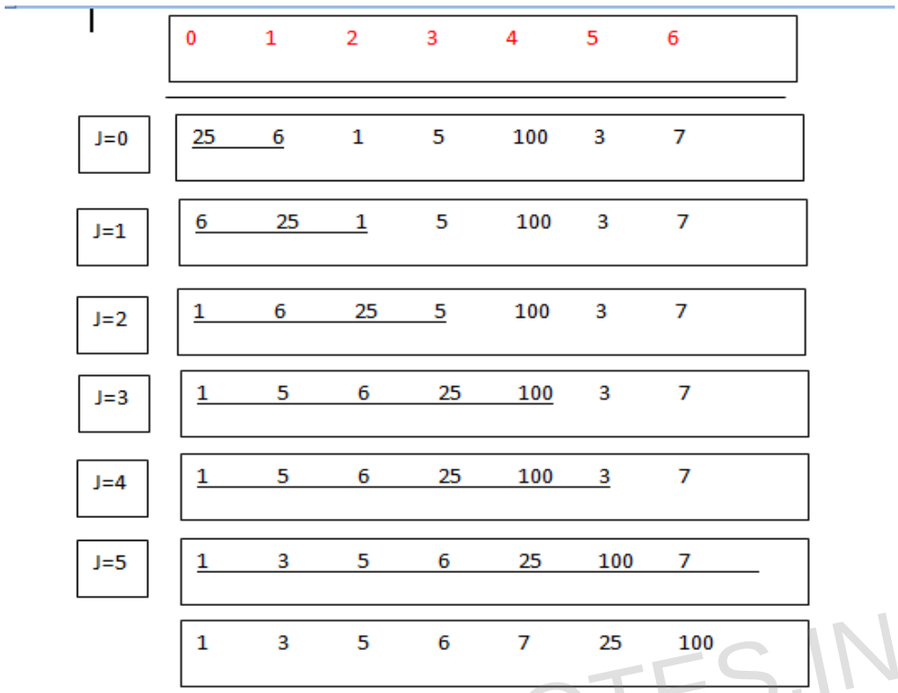
Here first iteration takes 1 comparison, 2nd iteration takes 2 comparison, and last iteration takes (n-1) comparison

Total comparison=1+2+…+n-1

$$= \frac{n(n-1)}{2}$$

$$= O(n^2)$$

Example



MERGE SORT

mergesort(start,end)

Input: An unsorted array a[], n is the no.of elements, start indicates lower bound of the array, end indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

1.start

2. if(start!= end)

    1. mid=(start+end)/2
    2. mergesort(start,mid)
    3.mergesort(mid+1,end)
    4. merge(start,mid,end)

3. end if

4.stop

Algorithm for merge

Merge(start,mid,end)

1. i=start
2. j=mid+1
3. k=start
4. while i<=mid and j<= end do
    1. if a[i]<=a[j]
        1. temp[k]=a[i]
        2. i=i+1
        3. k=k+1

    2. Else
        1. Temp[k] =a[j]
        2. J=j+1
        3. k=k+1
    3. end if

5.end while

6. while i<=mid do
    1. temp[k]=a[i]
    2. i=i+1
    3. k=k+1

7. end while

8. While j<=end
    1. Temp[k]=a[j]
    2. j=j+1
    3. k=k+1

9. end while

10. k=start

11. while k<=end
    1. a[k]=temp[k]
    2. k=k+1

12. end while

13.stop

Merge sort follows the strategy divide and conquer method. Here the given base array is divided into two sub lists. These 2 sub lists is again divided into 4 sub lists. The process is continued until subsists contain single element. Then repeatedly merge these two sub lists to a single sub list. So that a sorted array is created from sorted sub lists. The process continues until a sub list contains all the elements that are sorted**.**

Analysis of merge sort

Suppose the merge sort follows the recurrence relation.

$T(n)=2T(n/2)+n$

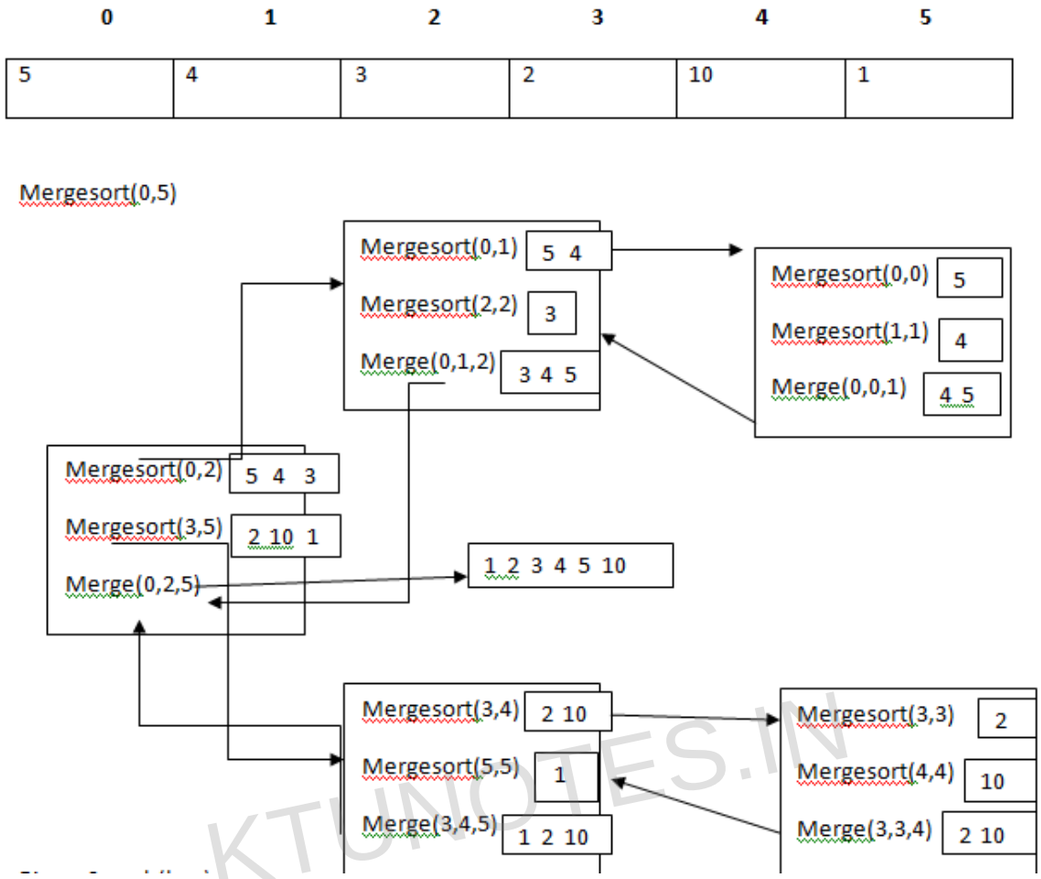
Comparing with standard recurrence equation $T(n)=aT(n/b)+f(n)$

a=2, b=2, f(n)=n

then $n^{\log_b a}$ =n

therefore time complexity is $\Theta(n \log n)$

Consider the following example



QUICK SORT

Quicksort(lb,ub)

Input: An unsorted array a[], n is the no.of elements, lb indicates lower bound of the array, ub indicates the upper bound of the array

Output: a sorted array

DS: Array

Algorithm

1.start

2.if lb<ub

1. loc=partition(lb,ub)
2. quicksort(lb,loc-1)
3. quicksort(loc+1,ub)

3. end if

4. stop

Algorithm for partition

Partition(lb,ub)

1. pivot=a[lb]
2. up=ub
3. down=lb
4. while down< up
    1. while pivot>=a[down] and down<=up
        1.down=down+1
    2. end while
    3. while a[up]>pivot
        1.up=up-1
    4. end while
    5. if down< = up
        1. swap(a[down], a[up])
    6. end if
5. end while
6. swap(a[lb],a[up])
7. return up
8. stop

Quick sort algorithm basically takes the following steps

1. Choose a pivot element(the pivot element may be in any position). Normally first element is chosen as pivot

2. Perform partition function in such a way that all the elements which are lesser than pivot goes to the left part of the array and all the elements greater than pivot go to the right part of the array. The partition function also places pivot in exact position.

3. Recursively perform quick sort algorithm in these two sub arrays

Analysis

The recurrence relation ofquick sort in worst case is

$T(n)=T(n/2)+T(n/2)+(n^2)$

$T(n)=2T(n/2)+(n^2)$

Comparing with the standard recurrence equation

$T(n)=aT(n/b)+f(n)$

Substitute the values of a and b in $n^{\log_b a}$ and compare it with f(n).

In this case a=2, b=2 then $n^{\log_b a}=n$

Whereas f(n)= n^2. Therefore time complexity of quick sort is $\Theta(n^2)$

MODULE 6

SEQUENTIAL SEARCH

Sequential_search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

DS: Array

1. Start
2. i=0
3. flag=0
4. While i<n and flag=0
    1. If a[i]=key
        1. Flag=1
        2. Index=i
    2.end if
    3. i=i+1
5. end while
6. if flag=1
    1. print “the key is found at location index”
7. else
    1.print “key is not found”

7.end if

8. stop

Analysis

In this algorithm the key is searched from first to last position in linear manner. In the case of a successful search, it search elements up to the position in the array where it finds the key. Suppose it finds the key at first position, the algorithm behaves like best case, If the key is at the last position, then algorithm behaves like worst case. Thus the worst case time complexity is equal to the no. of comparison at worst case ie., equal to O(n). The time complexity in best case is O(1).

The average case time complexity =(no. of comparisons required when the key is in the first position + no. of comparisons required when the key is in second position+…+ no. of comparison when key is in nth position)/n

$$\frac{1+2+\ldots.n}{n} = \frac{n(n+1)}{2n} = O(n)$$

Binary Search

Binary Search(key)

Input: An unsorted array a[], n is the no.of elements, key indicates the element to be searched

Output: Target element found status

DS: Array

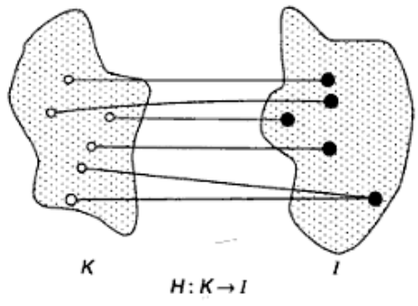
1. Start
2. Start=0,end=n-1
3. Middle=(start+end)/2
4. While key!=a[middle] and start<end
    1. If key>a[middle]
        1. Start=middle+1
    2. else
        1. end= middle-1
    3. end if
    4. middle=(start+end)/2
5. end while
6. if key=a[middle]
    1. print “the key is found at the position”
7. else
    1. print “the key is not found”
8. end if
9. stop

HASHING

We have seen about different search techniques (linear search, binary search) where search time is basically dependent on the no of elements and no. of comparisons performed.

Hashing is a technique which gives constant search time. In hashing the key value is stored in the hash table depending on the hash function. The hash function maps the key into corresponding index of the array(hash table).

The main idea behind any hashing technique is to find one-to-one correspondence between a key value and an index in the hash table where the key value can be placed. Mathematically, this can be expressed as shown in figure below where K denotes a set of key values, I denotes a range of indices, and H denotes the mapping function from K to I.



All key values are mapped into some indices and more than one key value may be mapped into an index value. The function that governs this mapping is called the hash function. There are two principal criteria in deciding hash function H:K->I as follows.

1) The function H should be very easy and quick to compute
2) It should be easy to implement

As an example let us consider a hash table of size 10 whose indices are 0,1,2,...9.Suppose a set of key values are 10,19,35,43,62,59,31,49,77,33.Let us assume the hash function as stated below

1) Add the two digits in the key
2) Take the digit at the unit place of the result as index , ignore the digits at tenth place if any

Using this hash function, the mapping from key values to indices and to hash tables are shown below.

K	I
10	1
19	0
35	8
43	7
62	8
59	4
31	4
49	3
77	4
33	6

H: K → I

0	19
1	10
2	
3	49
4	59, 31, 77
5	
6	33
7	43
8	35, 62
9	

Hash table

HASH FUNCTIONS

There are various methods to define hash function

Division method

One of the fast hashing functions, and perhaps the most widely accepted, is the division method, which is defined as follows:

Choose a number h larger than the number n of keys in K. The hash function H is then defined by

H(k)=k(MOD h) if indices start from 0

Or

H(k)=k(MOD h)+1 if indices start from 1

Where k€K, a key value. The operator MOD defines the modulo arithmetic operator operation, which is equal to dividing k by h. For example if k=31 and **h=13 then,**

H(31)=31 MOD 13=5 (OR)

H(31)=31(MOD 13)+1=6

h is generally chosen to be a prime number and equal to the sizeof hash table

MID SQUARE METHOD

Another hash function which has been widely used in many applications is the mid square method. The hash function H is defined by H(k)=x, where x is obtained by selecting an appropriate number of bits or digits from the middle of the square of the key value k. example-

k	**:**	**1234**	**2345**	**3456**
k^2	**:**	**1522756**	**5499025**	**11943936**
H(k)	**:**	**525**	**492**	**933**

For a three digit index requirement, after finding the square of key values,the digits at 2nd, 4th and 6th position are chosen as their hash values.

FOLDING METHOD

Another fair method for a hash function is folding method. In this method, the key k is partitioned into a number of parts k_1, k2..kn where each part has equal no.of digits as the required address(index) width. Then these parts are added together in the hash function.

H(k)=k1+k2+...+kn. Where the last carry, if any is ignored. There are mainly two variations of this method.

1)fold shifting method

2) fold boundary method

Fold Shifting Method

In this method, after the partition even parts like k2, k4 are reversed before addition.

Fold boundary method

In this method, after the partition the boundary parts are reversed before addition

Example

-Assume size of each part is 2 then, the hash function is computed as follows

Key values k :	**1522756**	**5499025**	**11943936**
Chopping :	**01 52 27 56**	**05 49 90 25**	**11 94 39 36**
Pure folding :	**01+52+27+56=136**	**05+49+90+25=169**	**11+94+39+36=180**
Fold Shifting:	**10+52+72+56=190**	**50+49+09+25=133**	**11+94+93+36=234**
Fold Boundary :	**10+52+27+65=154**	**50+49+90+52=241**	**11+94+39+63=207**

DIGIT ANALYSIS METHOD

This method is particularly useful in the case of static files where the key values of all the records are known in advance. The basic idea of this hash function is to form hash address by extracting and/or shifting the extracted digits of the key. For any given set of keys, the position in the keys and the same rearrangement pattern must be used consistently. The decision for extraction and rearrangement is finalized after analysis of hash functions under different criteria.

Example: given a key value 6732541, it can be transformed to the hash address 427 by extracting the digits from even position. And then reversing this combination.ie 724 is the hash address.

Collision resolution and overflow handling techniques

There are several methods to resolve collision. Two important methods are listed below:

1) **Closed hashing(linear probing)**
2) **Open hashing (chaining)**

CLOSED HASHING

Suppose there is a hash table of size h and the key value is mapped to location i, with a hash function. The closed hashing can then be stated as follows.

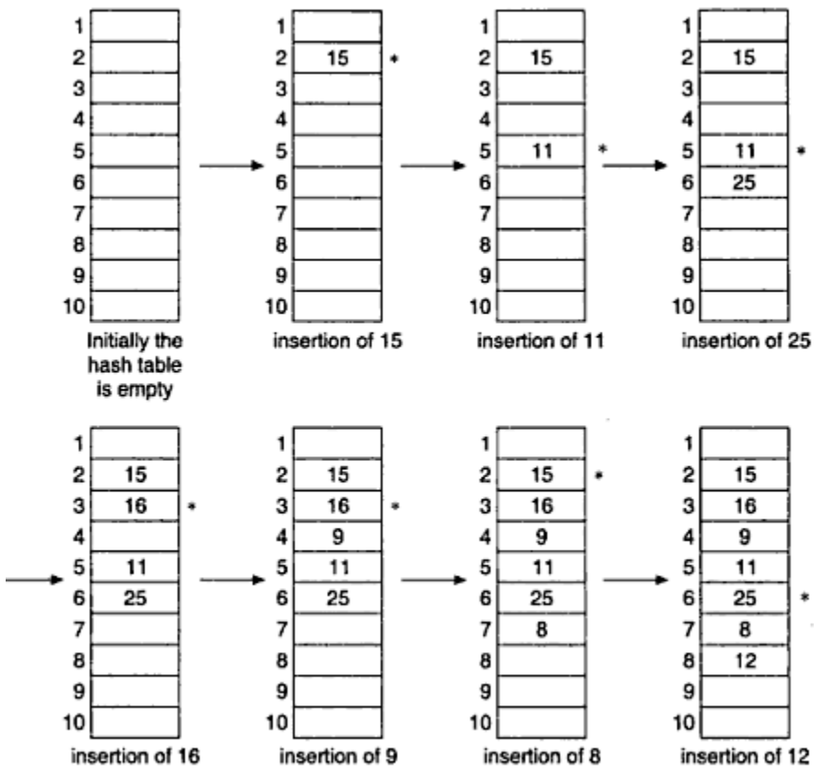
Start with the hash address where the collision has occurred,let it be i. Then carry out a sequential search in the order:- i, i+1,i+2..h,1,2...,i-1

The search will continue until any one of the following occurs

- The key value is found
- An unoccupied location is found
- The searches reaches the location where search had started

The first case corresponds to successful search , and the other two case corresponds to unsuccessful search.Here the hash table is considered circular, so that when the last location is reached, the search proceeds to the first location of the table. This is why the technique is termed closed hashing. Since the technique searches in a straight line, it is alternatively termed as linear probing.

Example- Assume there is a hash table of size 10 and hash function uses the division method of remainder modulo 7, namely H(k)=k(MOD 7)+1.The construction of hash table for the key values 15,11,25,16,9,8,12,8 is illustrated below.



Drawback of closed hashing and its remedies

The major drawback of closed hashing is that as half of the hash table is filled, there is a tendency towards clustering.That is key values are clustered in large groups.and as a result sequential search becomes slower and slower. This kind of clustering is known as primary clustering.

The following are some solutions to avoid this situation

1)Random probing

2)Double hashing

3)Quadratic probing

Random Probing

Instead of using linear probing that generates sequential locations in order, a random location is generated using random probing.

An example of pseudo random number generator that generates such a random sequence is given below:

I=(i+m)MOD h+1

Where m and h are prime numbers. For example if m=5, and h=11 and initially=2 then random probing generates the sequence

8,3,9,4,10,5,11,6,1,7,2

Here all numbers are generated between 1 and 11 in a random order. Primary clustering problem is solved. Where as there is an issue of clustering when two keys are hashed into the same location and then they make use of the same sequence locations generated by the random probing, which is called as secondary clustering

Double Hashing

An alternative approach to solve the problem of secondary clustering is to make use of second hash function in addition to the first one. Suppose H1(k) is initially used hash function and H2(k) is the second one. These two functions are defined as

H1(k)=(k MOD h)+1

H2(k)=(k MOD (h-4))+1

Let h=11, and k=50 for an instance, then

H1(50)=7 and H2(50)=2.

Now let k=28, then H1(28)=7 and H2(28)=5

Thus for the two key values hashing to the same location, rehashing generates two different locations alleviating the problem of secondary clustering.

Quadratic Probing

It is a collision resolution method that eliminates the primary clustering problem of linear probing. For linear probing, if there is a collision at location i, then the next locations i+1, i+2..etc are probed.But in quadratic probing next locations to be probed are $i+1^2$,$i+2^2$,$i+3^2$ ..etc . This method substantially reduces primary clustering, but it doesn't probe all the locations in the table.

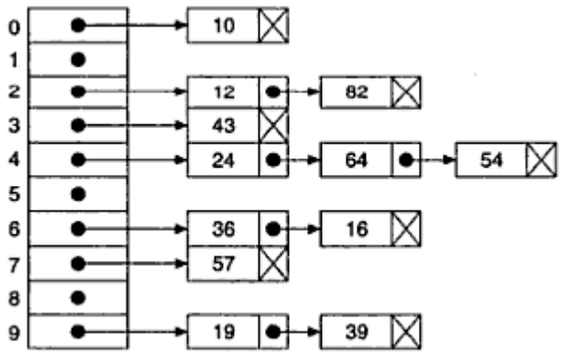
Open Hashing

Closed hashing method for collision resolution deals with arrays as hash tables and thus random positions can be quickly refer red. Two main disadvantages of closed hashing are

1) It is very difficult to handle the problem of overflow in a satisfactory manner
2) The key values are haphazardly intermixed and, on the average majority of the key values are from their hash locations increasing the number of probes which degrades the overall performance

To resolve these problems another hashing method called open hashing or separate chaining is used.

The chaining method uses hash table as an array of pointers. Each pointer points a linked list. That is here the hash table is an array of list of headers. Illustrated below is an example with hash table of size10.



For searching a key in hash table requires the following steps

1)Key is applied to hash function

2) Hash function returns the starting address of a particular linked list(where key may be present)

3)Then key is searched in that linked list

Performance Comparison Expected

Algorithm Name	Best Case	Average Case	Worst Case
Quick Sort	O(n log n)	O(n log n)	$O(n^2)$
Merge Sort	O(n log n)	O(n log n)	O(n log n)
Heap Sort	O(n log n)	O(n log n)	O(n log n)
Bubble Sort	O(n)	$O(n^2)$	$O(n^2)$
Selection Sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion Sort	O(n)	$O(n^2)$	$O(n^2)$
Binary Search	O(1)	O(log n)	O(log n)
Linear Search	O(1)	O(n)	O(n)

MODULE 3

Stack ,Queue and Strings

Stack

A stack is ordered collection of homogeneous data elements where insertion and deletion takes place at one end only. Like array and linked list stack is also a linear data structure, but the only difference is insertion and deletion takes place only at one position. The insertion & deletion in case of stack is specialy termed as PUSH and POP respectively. The position of stack where these operations are performed is known as TOP of stack. An element in stack is termed as ITEM. The maximum number of elements that a stack can be accommodate is termed as SIZE.

Stack can be represent in two ways

1. array representation
2. Linked list representation

Array representation

In array first we have to allocate a memory block of sufficient size to accommodate full capacity of stack. TOP indicate the top of the stack. Initial

value of TOP is -1. SIZE indicate the size of the stack. For PUSH operation first we have to check whether the stack is full or not. The condition TOP>=SIZE-1 indicate the stack is full. If the stack is not full, then we will increment TOP by 1 and an ITEM is inserted at this place. For performing POP operation first we have to check whether stack is empty or not. The condition TOP<0 indicate the stack is empty. If the stack is not empty then remove ITEM that are present in TOP and decrement TOP by 1.

PUSH Operation

PUSH(item)

i/p: an array STACK[SIZE],ITEM to be inserted, TOP indicate top of stack and its initial value is -1

O/p: ITEM is inserted to stack

Data Structure: an array STACK[SIZE]

1. Start
2. If TOP>=SIZE-1 then
    1. Print “stack is full”
3. Else
    1. TOP=TOP+1
    2. STACK[TOP]=ITEM
4. End if
5. Stop

POP Operation

POP( )

i/p: an array STACK[SIZE], TOP indicate top of stack and its initial value is -1

O/p: ITEM is deleted from the TOP of stack

Data Structure: an array STACK[SIZE]

1. Start
2. If TOP<0 then
    1. Print “stack is empty”
3. Else
    1. ITEM=STACK[TOP]
    2. TOP=TOP-1
4. End if
5. Stop

Linked List representation

the array representation is easy to implement, but it allows only to represent a fixed size. In several application the size of stack may vary during the program execution. This can be achieved by representation of stack using linked list. In linked list representation TOP indicates top of the stack, the TOP is a pointer which stores address of starting node of linked list. During the PUSH operation a new node is inserted at the beginning of the linked list and TOP is adjusted to new address. During POP operation , a node is deleted from beginning of the linked list and TOP is adjusted .

PUSH Operation

PUSH(item)

i/p: Linked list representation of stack, TOP indicate starting address of linked list and its initial value is NULL,ITEM is the data element to be inserted.

O/p: ITEM is inserted to stack

Data Structure: singly linked list

1. Start
2. Create a node and store its starting address to TEMP
3. If TEMP=NULL then
    1. Print “node is not created, PUSH operation cannot be performed”
4. else
    1. TEMP->data=ITEM
    2. TEMP->link=TOP
    3. TOP=TEMP
5. End if
6. Stop

POP Operation

POP( )

i/p: Linked list representation of stack, TOP indicate starting address of linked list and its initial value is NULL,ITEM is the data element to be inserted.

O/p: ITEM is deleted from the top of stack

Data Structure: singly linked list

1. Start
2. If TOP=NULL then
    1. Print “Stack is empty”
3. else
    1. TEMP=TOP
    2. ITEM=TEMP->data
    3. TOP=TOP->link

4. Delete node pointed by TEMP

4. End if
5. stop

Application of Stack

1. expression evaluation
2. Recursion
3. Tower of Hanoi problem

Infix expression:- is one where the operator is in between operands(eg A+B)
Prefix expression:- where the operator comes before operands (eg +AB). The prefix expression is also called Polish notation.
Postfix expression:- where the operator comes after operands (eg AB+). The postfix expression is also called reverse polish notation.

Q) convert the following expression into prefix and postfix notations
A+ B/C –D* E ^F

Postfix expression:-
→A+ B/C –D* E ^F
→A+ B/C –D* E F^
→A+ BC/ –D* E F^
→A+ BC/ – D E F^*
→ABC/+ – D E F^*
→ABC/+ D E F^*-

ABC/+ D E F^*-

Prefix expression
→A+ B/C –D* E ^F
→A+ B/C –D* ^E F
→A+ /BC –D* ^E F
→A+ /BC – *D ^E F
→+A /BC – *D ^E F
→+A /BC – *D ^E F
→-+A /BC*D ^E F

-+A /BC*D ^E F

Algorithm to convert infix expression to postfix expression

Suppose P is the expression in infix form, we have to convert these infix expression to postfix expression Q

The following algorithm uses the operators ^, *, /,+,-

1. Start
2. Push "(" (opening bracket) into stack and ")" (closing bracket) to the end of infix expression P
3. Scan infix expression P from left to right and repeat step 4 to step 7 until stack is empty
4. If an operand is encountered add it into postfix expression Q
5. If "(" is encountered push it to stack
6. If an operator Θ is encountered
    1. Repeatedly pop from stack and add it to postfix expression Q (the pop operation performs only when the operator at the top of the stack has same precedence or higher precedence than Θ
    2. Add operator Θ tp stack
7. If closing bracket ")" is encountered
    1. repeatedly pop from stack and add it to postfix expression(until an opening bracket is encountered)
    2. remove opening bracket
8. stop