Data structure

Unit-1 : Def^n, Heap & stack memory allocation, ~~Type of DS~~ Time complexity,

Unit-2 : Arrays, array ADT, memory representation, possible operation, ← stack memory

Unit-3 : Linked list ← Heap memory

Unit 4 : Stack, Application : Ifix to postfix, evaluate aisthmetic exp, Recurtion. ← LIFO

Unit 5 : Queue ← FIFO

Unit 6 : Tree (Hierarchy maintain)

Unit 7 : Graph ← Inter connected data

Linear & physical data s, define memory block by own

Logical DS (implementation of particular logic)

* Data type :- (needed ~~data to~~ property)

int = 3

float = 3.0

(a) ~~It~~ ~~to~~ How it is stored in memory.

(b) operation which can ~~to~~ be performed

* Abstract data type :- (ADT)

We know its operation but not their implementation.

eg stack :- push, pop

* Concrete implementation of ADT. } Data structur
* Management of data in main memory.
* (way in which data is stored in main memory).

09/08/23

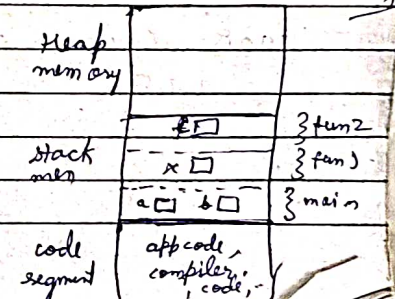
```
#include <stdio.h>

struct rectangle
{
    int l;
    int b;
};

~~void fun (struct rectangle *r1) (&r1)~~
{}
void fun (struct rectangle *r1)
{  r1 -> l = 20;      or  (*r1).l = 20;
   r1 -> b = 10;          (*r1).b = 10;
}

int main()
{
   struct rectangle r1;

   r1.l = 10;
   r1.b = 20;

   fun (&r1);
}
```

~~abc~~

* Data structure :-
  A way to organise the data in main memory.

* Static & Dynamic memory allocation :-

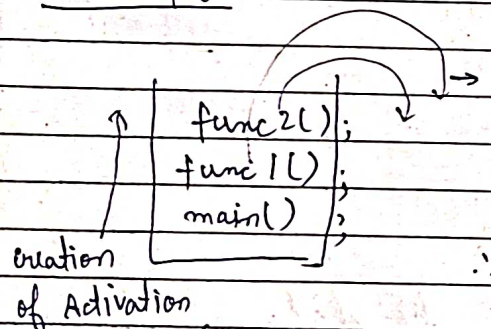
→ Activation record is assigned differentely to manage even same name variable.



*Static memory allocation:-

```
int main()
{
    int a;
    int b;
    fun1();
    fun2();
}

fun1()
{
    float x;
    func2();
}

fun2()
{
    int c;
    c=10
    print(c);
}
```

* Control flow:-



→ once func2(); ends control transfer to func1(); → then in main();

∴ LIFO is used as at last func2() 's ~~is active~~ (ARC) is made & at first is deleted

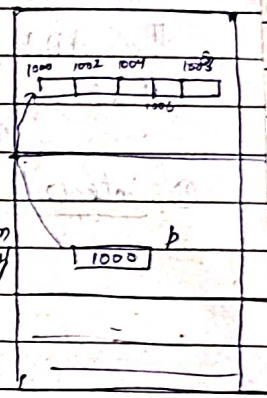
⇒ so called stack memory.

→ stack memory — memory is allocated at compiled time

malloc → return void pointer which is typecasted (int*)

* Dynamic memory allocation:-

```
int main()
{
    int *p;
    p = (int*) malloc (sizeof(int) *5);
    free(p);
    p=NULL;
}
```



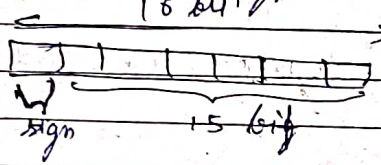
⇒ memory is allocated during runtime.

data type of → representation, operation.

Abstract data types (ADT's)

The ADT refers to the basic mathematical concept that define the data type.

① integer



+, −, *, /, %, ++, −−

② list

[8, 3, 9, 1, 6, 52 − − −]
0 1 2 3 4 5 6 − − −

operation
1) add
2) removal (index)
3) search (key)
4)

1) space
2) size of list
3) max^{n} capacity

③ Rational number :

$\frac{p}{q} = \frac{1}{2}$

operation
i) add
ii) sub
iii) * /

1) space
ii)

/* value def^{n} */

abstract typedef <integer, integer> RATIONAL; → Def^{n} domain

Condition Rational[1] != 0;

/* operator def^{n} */

① abstract rational make rational(a, b);
int a, b;
pre cod^{n} b != 0;
post cond^{n} make rational[0] = a;
make rational[1] = b;

② abstract rational add (a, b)

Rational a, b;

post cod^{n} add rational[0] = a[0] * b[1]
add [1] = a[1] * b[1]
add [0] = a[0] * b[1]
+ a[1] * b[0]

③ abstract rational mul (a, b)
Rational a, b;
post cond^{n} add [0] = a[0] * b[0];
add [1] = a[1] * b[1];

(iv) abstract equal (a, b);
rational a, b;
postcond^{n} ~~equal~~ => (a[0] * b[1] == a[1] * b[0])

16/08/23

Array :-

* declaration

int A[5]; ← static

dynamic

```
int *p;
p = (int*) malloc (sizeof(int)*5);
free(p);
p = NULL;
```

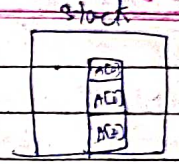
* 2D array :-
int A[3][4]

0	1	2	3
4	5	6	7
8	9	10	11

(3 rows, 4 columns)

→ row major formula is used to store 2d array in memory (i * r + c);

dynamically :-



① int *A[3]

```
for (i=0 → 3)
{ A[i] = (int*) malloc (sizeof(int) * 4)
```

② int **A = (int **) malloc (sizeof(int) * 3);

```
for (i=0 → 3)
{
    A[i] = (int*) malloc (sizeof(int)*4);
}
```

* ⇒
A[2] = *(A+2) = 2[A]

A[2][3] = *(*(A+2)+3)

* formula for access:

① 1D array

8	4	6	9	13
2000	2002	2004	2006	2008

A[2] = 6;

Addr (A[2]) = 2000 + 2 × 2

Addr [A[4]] = 2000 + 2 × 4

Addr [A[i]] = lo + w (i − 1)

- lo → A[0]
- w → sizeof (int)

② 2D array

ⓐ Row major :

2000	0	1	2	3
0	a_{00}	a_{01}	a_{02}	a_{03}
1	a_{10}	a_{11}	a_{12}	a_{13}
2				
3				

add [A[1][2]) = 2000 + (4*1 + 2)*2 = 2012

add [A[2][3]) = 2000 + (4*2 + 3)*2

a_{00}	a_{01}	a_{02}	a_{03}	a_{10}	a_{11}	a_{12}	a_{13}

1st row — 2nd row

2000 + 7 × 2 = 1014

m × n

add (A[i][j]) = lo + (n * i + j) * sizeof (int);

ⓑ Col major :

	0	1	2	3
0	00	01	02	03
1	10	11	12	13
2	20	21	22	23
3	30	31	32	33

m ↓ , n →

00	10	20	30	01	11	31	33

Add [A[3][1]

= lo + (m * j + i) * (sizeof(int));

~~= 200 + (n * 1 + 3) * (2)~~ 2000

3 × 4

Add [A[2][1]] = 2000 + (3 * 1 + 2) * (2)

= 2000 + 10 = 2010

Add (A[i][j]) = lo + (m * j + i) * sizeof (int);

* formula for 2d array :-

Addr [A[i, j]] = $l_0 + (i * n + j) * w$ — Row major

Addr [A[i, j]] = $l_0 + (j * m + i) * w$ — Col major

A [d_1] [d_2] [d_3] [d_4]

Row major ⇒ $l_0 + [i_1 \times d_2 \times d_3 \times d_4 + i_2 \times d_3 \times d_4 + i_3 \times d_4 + i_4] * w$

Col major ⇒ $l_0 + [i_4 \times d_1 d_2 d_3 + i_3 d_1 \times d_2 + i_2 \times d_1 + i_1] * w$

eg int A[l][m][n] l x m x n, l = 10, m = 5, n = 20, i = 2, j = 3, k = 4

Add of A[i][j][k]

⇒ $l_0 + [i \times m \times n + j \times n + k] * 2$

$l_0 = 1000$

$= 100 + [2 \times 5 \times 20 + 3 \times 20 + 4] * 2$

$= 1000 + [200 + 64] * 2$

$= 1528$

eg A[1 --- 10] [1 --- 10]

$l_0 = 100$

A[i] [j] = $100 + [(i-1) \times 10 + (j-1)]$

$= 100 + 10(i-1) + j - 1$

$= 100 + 10i - 10 + j - 1$

$= 100 + 10i + j - 10$

$= 90 + 10i + j$

$= 100 + [15(i-1) + j - 1] = 90 + 9i + j$

$= 84 + 15j + j$

	0	1	2	
0	1	2	3	
1	4	5	6	
2	7	8	9	
3	10	11	12	

200 ← x → 6

212 ← 1

*(x+1) ← 2

236 ← 3

x = 200

w = 4

(X + y) = X + y x (space of row)

*(x+y) + y = Starting index of y^{th} row + y x size

X + 3 = 200 + 3 x 4 = 212 X + 3 = 236

*(x + 3) = 4 *(x+3) = 236

*(x + 2) + 3 → 6 *(x+2) + 3

∴ 0 is defined doble pointer.

∴ X.

= 224 + 4 x 3

= 236

((X+2)+4) ⇒ *(X+2)+

Q $t_0 = i * 1024 = i \times 2^{10}$

$t_1 = j \times 32 = j \times 2^5$ X[][][]

$t_2 = k * 4 = k \times 2^2$ data-type ?

$t_3 = t_0 + t_1$ dimensions ?

$t_4 = t_3 + t_2$

$t_5 = X[t_4]$

int → 2

float → 4

Q

$t_3 = i \times 2^{10} + j \times 2^5$

$t_4 = i \times 2^{10} + j \times 2^5 + k * 2^2$

$t_5 = X[i \times 2^{10} + j \times 2^5 + k \times 2^2]$

$= X[i \times 2^8 + j \times 2^3 + k] \times 2^2$

∴ datatype = 2^2 → mxn, n, mxn

⇒ n = 8 m = 32

(w=4) I can't be determined

5 32
6.64
7 128
8 256
9 512

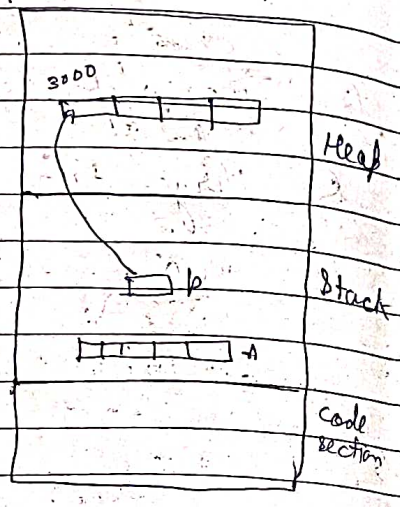
Q Insert an element in sorted array. (+ve & -ve element)

Q Two approach to find array is sorted or not.

Q Merge two array (already sorted):

Linked list :-

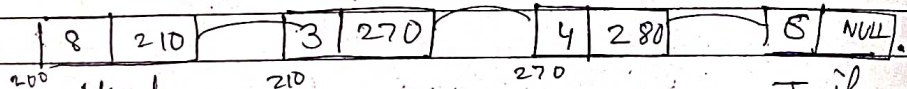
```
int A[5];
int *p
```



→ It is a variable size data structure which can grow or shrink according to the behaviour.

① What is linked list ?

A LL is collection of node where each node contains the data and pointer to next node



② What is node ?

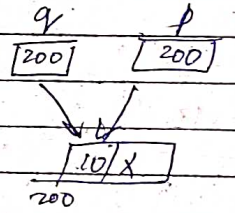
```
struct node                          → 2 bytes
{
    int data;                        → self prefencial structure
    struct node *next;               → 2 bytes
};                                   ∴ total 4 bytes
```

③ Create a node ?

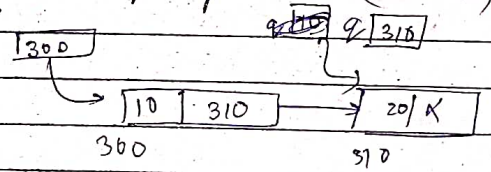
```
struct node *p;
p = (struct node *) malloc (size of (struct node));
```

(a) `(*p).data = 0`
`(*p).next = NULL`

Eg: `struct node *q, *p;`



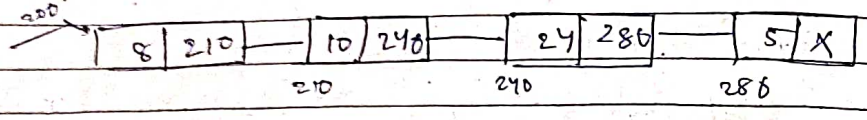
1) `q = p`
2) `q = p → next (= NULL)`
3) `p = p → next (= NULL)`



`p = p → next → next (= NULL)`

23/08/22

* display all elements :

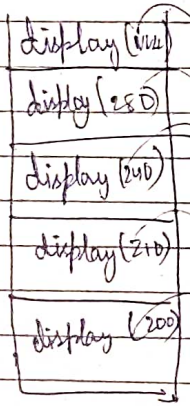



```
void display (struct node * p)
{
    while ( p != NULL)
    {
        printf ("%d\n", p -> data);
        p = p -> next;
    }
}
```

Output
8
10
12
5

Recur

```
void display (struct node * p)
{
    if (p == NULL) return;
    printf ("%d \n", p -> data);
    p = p -> next; display (p -> next);
}
```



* → reverse order

```
void display (struct node * p)
{
    if (p != NULL)
    {
        display (p -> next);
        printf ("%d", p -> data);
    }
}
```

5
12
10
8

* count node

```
int display (struct node *p)
{
    if (p == NULL) return 0;
    else
    { printf (p -> data);
      return (1+(display (p -> next)));
    }
}
```

* maxm element :

Recur

```
int max (struct node *p)
{
    if (p == NULL) return (-1e9);
    else return max(p->data, max(p->next));
}
```

iter

```
int max(struct node *p)
{   int count = 0; maxm = -1e9;
    while (p != NULL)
    {   count++;
        maxm = max(p->data, maxm);
        p = p->next;
    }
    return maxm;
}
```

* inserting a node :

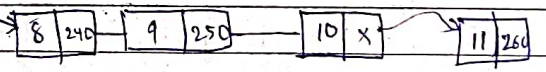
① at first pos

```
struct node *t;
t->data = data;
t->next = head;
head = t;
```

② at any pos

```
struct node *t, *p;
int count = 0;
t->data = data;
while (count != pos)
{   p = p->next;
    count++;
}
t->next = p->next;
p->next = t;
```

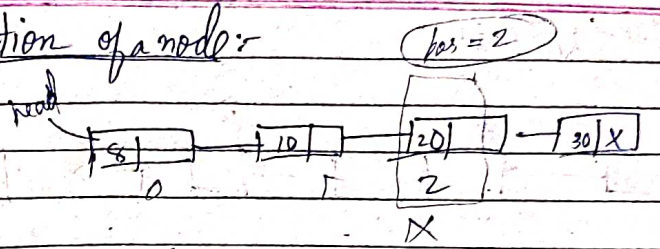
③ at last pos



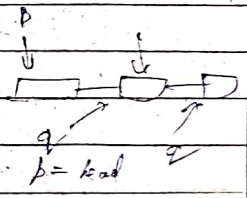

```
void insert (int data)
{   struct node *p = head, *t;
    t->data = data;
    while (p->next != NULL)
    {   p = p->next;
    }
    p->next = t;
}
```

```
void insert (int pos, int x)
{
struct node *t, *p;
  if (pos==0)
  {
     t=(struct node*) malloc (sizeof ~~node~~ struct node);
     t -> data = x;
     t -> next = first;
     first = t;
  }
else if (pos>0)
{
     p = first;
   for (i=0; i<pos-1; i++)
   {
     p = p -> next;
   }
     if (p)
   {
     t = (struct node*) malloc (size of struct node);
     t -> data = x; t -> next = p -> next;
      p -> next = t;
}
```

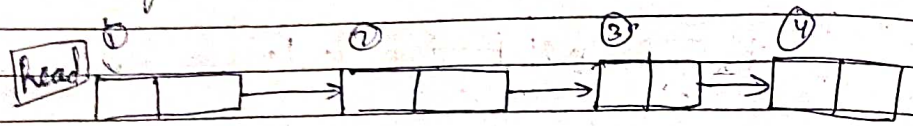
* Deletion of a node:




```
void delete (int pos)
{  struct node *p, *q;
   if (pos==0)
   {
        p = head;
      head = head -> next
        free (p);
        p = NULL;
}
else
{   p = head;
    int count = 0;
   while (count != pos-1)
   {
      p = q;
      q = p -> next;
   }
   p -> next = q -> next;
    free (q)
    q = NULL;
}
}
```



Reversing a linked list

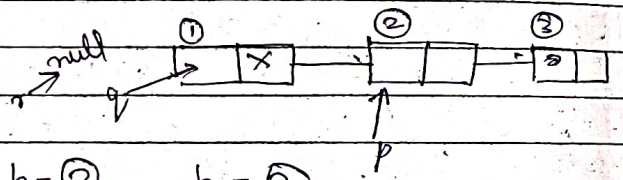


Three pointers : r q p q = null, r = null, p = first

```
while (p != null)
{
    r = q; q = p; p = p -> next;
    q -> next = r;
}
first = q;
```

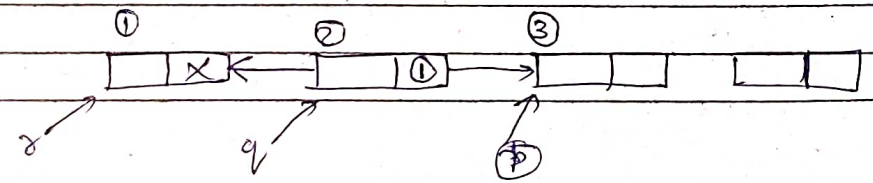
① r = q = null , q = p = first , p = p -> next = ②

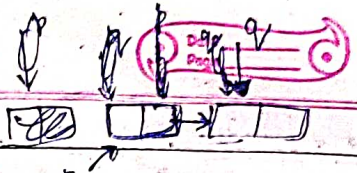
q -> next = null ⇒ first -> next = null



② r = q = ① , q = p = ② , p = ③

q -> next = r = ① ; ② -> next = ①

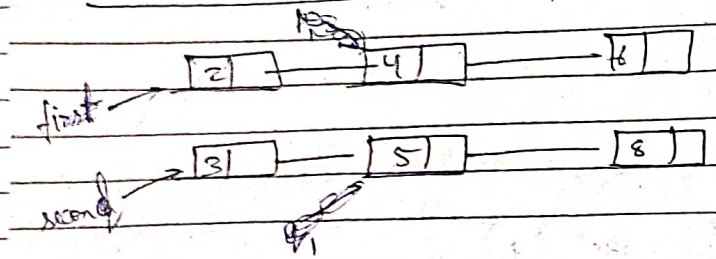




* Recursive :

```
void rev ( int *p, int *q)
{
    p -> next = q;
    p = q
    q = q -> next
    rev ( p, q);
}

void rev ( node *p, node *q)
{
    if (p != NULL)
    {
        rev ( p, p -> next)
        p -> next = q;
    }
    else
    {
        first = q ;
    }
}
```

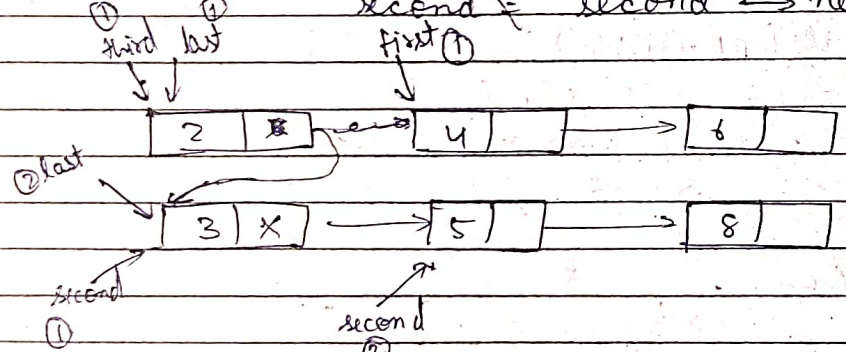
* Concatate two ~~to~~ linked list



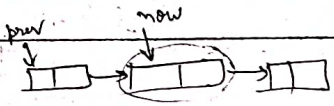

```
struct node * third, last;
if ( first -> data < second->data)
{   third -> data = first;
    first = first -> next;
}
else
{   third -> data =
    second = second -> next;
```

last -> next = first

last = first



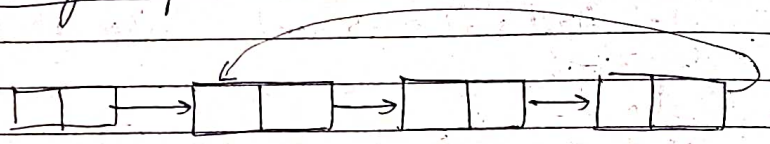

```
struct node * third, last;
if ( first -> data < second->data)
{   third = first; last = first; first = first->next;
}
else
{   third = 2nd; last = 2nd; 2nd = 2nd->next;
}
while ( first != null && 2nd != null)
{   if ( first -> data < second -> data)
    {
        last -> next = first;   last = first;
        first = first -> next;  last -> next = NULL;
    }
    else
    {
        third -> next = second;   last = second;
        second = second -> next;  last -> next = NULL;
    }
}
while( first != NULL)
{
    last -> next = first;   last = first;
    first = first -> next;  last -> next = NULL;
}
```



~~while (second != NULL)~~

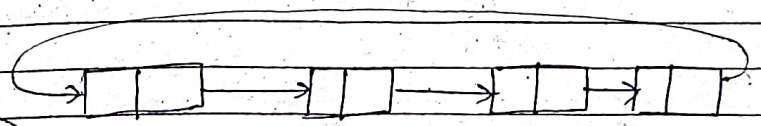
```
if (first == NULL)
{
    last -> next = second;
}
else
{
    last -> next = first;
}
```

Checking loop in linked list :-



```
while (p != q && p != NULL && q != NULL)
{
    p = p -> next;
    q = q -> next;
    if (q != NULL)
    {
        q = q -> next;
    }
    else
    {
        q = NULL;
    }
}
```

Circular Linked list



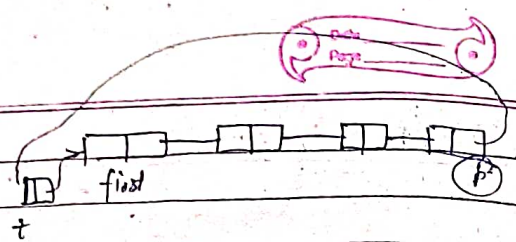
Print iterative

```
do {
    printf ("%d", p->data);
    p = p -> next;
}
while (p != head)
```

recursive

```
flag = 0;
void display ( p , head)
{
    static int flag = 0;
    if ( p != head && flag == 0)
    {
        flag = 1;
        printf ( p -> data) ;
        display ( p -> next) ;
    }
}
main ()
{
    p = head;
    printf ( p -> data);
    display (p -> next, head );
}
```

Insertion :-



① Insert at head

```
p -> next = t;
t -> next = first
first = t
```

② Insert at any:

Deletion :-

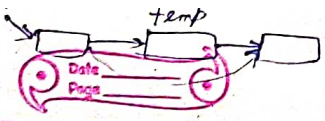
① Delete at head:

```
t = first;
p -> next = first->next;
first = first -> next;
free (t);
t = NULL;
```

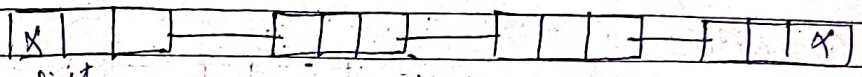
② Delete at any

```
int Delete_any ( struct node * head, int pos)
{  int pos=1; struct node * prev, *temp = head;
   =NULL
   while ( pos--)
   {  if ( temp != NULL)
      {  prev = temp;
         temp = temp -> next;
      }
   }
```

```
   int data = temp -> data;
   free (temp);
   return data;
}
```



Doubly linked list




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

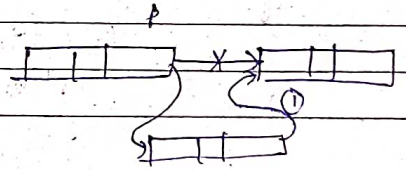
* In Display

* Insertion

① at head:

```
t -> prev = NULL;
t -> next = prev;
first -> prev = t;
first = t;
```

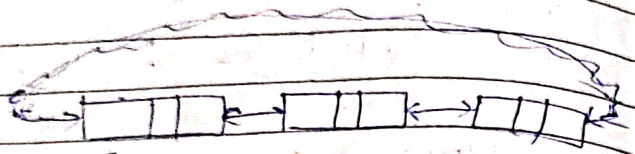
② at any pos:




```
if (p != NULL)
{  t -> next = p -> next;
   t -> prev = p;
{ if (p -> next != NULL) p -> next -> prev = t; }
p -> next = p -> t;
```

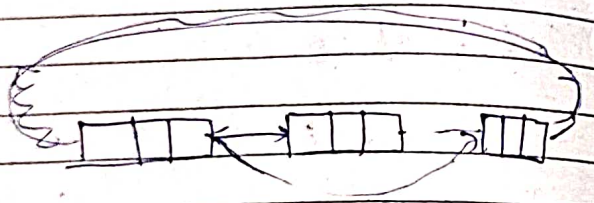
* deletion :-

① Delete head




```
if (first != NULL)
{
    t = first;
    first = first -> next;
(if first != NULL) first -> prev = NULL;   free(t);
}
```

② Deleteany



```
~~void~~ int deleteany (struct node * head, int pos)
{   struct node * temp = head, * prev1, * next1;
    while (pos--)
    {   if (temp != NULL)
        temp = temp -> next;
    }
    if (temp != NULL) { prev1 = temp -> prev, next1 = temp -> next;
    ~~temp -> prev = temp -> next~~; prev1 -> next = next1;
    } int data = temp -> data;   next1 -> prev = prev1;
    free (temp);
    return data;
}
```

final

```
int deleteany (struct node * head, int pos)
{
    struct node * temp = head;
    while (pos--)
    {
        if (temp != NULL)
            temp = temp -> next;
    }
    if (temp != NULL)
    {   if (temp == head) { ~~deletehead;~~ return deletehead;
        temp -> prev (-> next) = temp -> next;
        if (temp -> next != NULL)
        temp -> next -> prev = temp -> prev;
    }
    int data = temp -> data;
    free (temp);
    return data;
}
```

#

	Array	Linked list
Memory	Stack memory	heap memory
Size	fixed	variable
Memory utilization	under/over utilisation	efficiently utilized
Space	less	more
access mechanism	randomly O(1)	sequentially O(n)
Insertion at last	O(1)	O(n)
Insert at first	O(n)	O(1)
delete at last	O(n)	O(n)
delete at first	O(n)	O(1)
linear search	O(n)	O(n)
Binary search	O(log n)	~~not~~ no advantage

Stack :-

→ Basic behaviour :-

- push()
- pop()
- Isempty()
- Isfull()
- peak()
- size()

→ Basic requirement

- → top
- → size
- → space

512

69

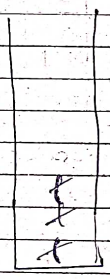
```
struct Inode
{   int data;
    struct node * peak; * prev;
    int size;
};

struct stack
{
    int top;
    int size;
    int *s;
};
```

```
int main()
{
    struct stack st;
    st.top = -1;
    scanf("%d", &st.size);

    st.s = (int*) malloc (sizeof(int) * size)

}

void push (int val; stack *st)
{
    if (st->top == (st->size) - 1) { stack is full return; }

    else { st[st->top + 1]
           st->top ++;
      st->s[st->top] = val;
           size ++;
    }

}
```

```
int pop (stack *st)
{
    if (st->top == -1) { empty return }
    else
    {
        int data = st->s[st->top];
        st->top--;

        return data;
    }
}
```

* Parenthesis matching:-

((a+b) * (c-d))



empty

```
bool check (char *brac[])
{
    struct stact st;
    st.top = -1; st.size = strlen(brac);
    st.size = 100; int i=0;
    while (brac[i] != '\0')
    {
        st.push(brac[i]);
        if(brac[i] == '(')
        { push (brac[i], &st);
        }
        else { if brac[i] == ')')
        {
```

```
if (top != -1)
{
    st.pop();
}
else
{ return 0; false;
}
}
return true;
if (top == -1) return true;
  else return false;
}
```

1	+/-	L-R
2	*, /	L-R
3	^	R-L

(-d)

↳ unary (-)
highest precedence

Infix to postfix:

Q. a+b*c-d/e

Q. abc*+de/-

Q. a+b*c-/(-d)^e^f/g+h-g

Q. abc*d-e f^^/g/+h+g-

~~ab*c*+(-d)e f^^/g/h+g-~~

~~abc*-def^^/g/f+h+g-~~

ab+c*

Algorithm:-

① Scan the given infix expression from left and initialise a empty string ans.

② If scanned term is operand (a-z, A-Z, 1-0) just add it to the ans.

③ If it is operator.

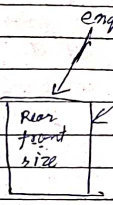
(a) If stack is empty or stack has less precedence operator (or same precedence & R-L associativity) (A)
→ push to stack.

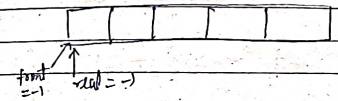
(b) If stack has higher precedence operator (or same precedence & L-R ass

```
char * ~~postfix~~convert(char *infix)
{
    struct stack st;
    st.size = strlen(infix) & top = -1;
    ~~int i=~~
    char *postfix = ~~=~~ (char *) malloc (sizeof(char) * strlen(infix));
    int i = 0, j=0;
    while (infix[i] != '\0')
    {
        if ( isoperand(infix[i])
        {
            postfix[j++] = infix[i++];
        }
        else
        {
            push(&st, infix[i++]);
```

q → (q)

* Queue :- Logical data-structure
→ FIFO






```
struct queue{
    int size;
    int front;
    int Rear;
    int * Q;
};
int main()
{
⇒   struct queue q;
    scanf("%d", &q.size);
    q.Q = (int *) malloc (sizeof(struct queue)*(q.size));
    enque (10, &q);
```

```
void enque (int x, struct queue *q)
{   if (q->size - 1 == q->rear)
    else if
    {(q->Rear)++;
     q->Q[q->Rear] = x;
    }
}
```

Start → front = -1 (prev to start)
rear = -1 (last elem)

int dequeue (

{

if (front == rear) empty return ;

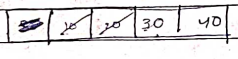
int x = ~~struct queue *q~~ q → Q [front] ;

front ++ ;

return x ;

}

* Circular queue :



Empty → F = R

full → ~~Rear = F~~

F = (R+1) % size

F = Ø X 2

R = Ø X 2 3

void enqueue (struct queue *q , int val)

{

~~if (F = (R+1) % (~~

if (q → Front == (q → rear+1) % (q → size)) {

printf("queue is full\n"); return; }

q → Rear = (~~q →~~ Rear+1) % (q → size) ;

q → Q [~~q →~~ Rear] = val ;

}

int dequeue (struct queue *q)

{

if (q → front == q → rear)

{

printf (" queue is empty \n");

return -1e9;

} int x = q → Q [front] ;

q → front = (q → front +1) % q.size ;

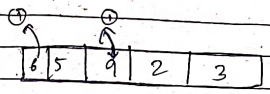
return x ;

}

Story Tarun

Doubly ended queue :

Priority queue



Insertion → O(n)

Deletion → O(n) + O(n) = O(n)

search shift

Arrd)

full);

rm ;

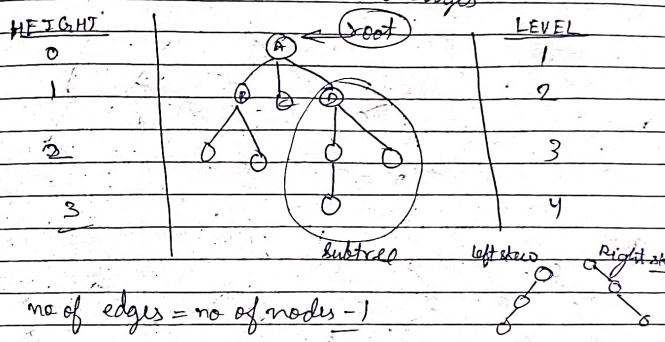
* Queue using linked list :-

```
struct node
{
    int data;
    int head;
    int tail;
    int * struct;
    struct node *next;
}
void enqueue ( struct node **head, struct node **tail, int data)
{
    struct node * temp;  temp = (struct node *)malloc(sizeof(st...);
    temp -> data = data;
    if (*head == NULL)
    {
        *head = temp;
    }
}
```

* Queue using linked list

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

void enqueue (int x)
{
    node * temp = (structnode *) malloc (sizeof(struct node);
    temp -> data = x;  if(temp == NULL) printf (full);
    if ( *front == NULL)
    {
        *front = *rear = t;
        return;
    }
    *rear -> next = t;
    *rear = t;
}

void dequeue ()
{
    if (*front == NULL) printf ( empty ) return;
    int x = (*front) -> data;  t = *front;
    (*front) = (*front) -> next;
```

Trees → Tree is a collection of nodes and edges



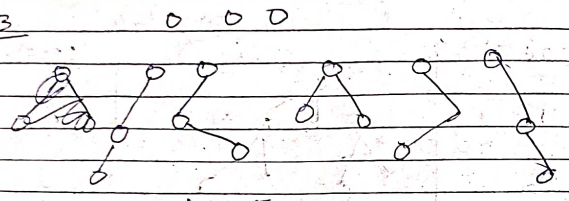
* no of edges = no of nodes - 1
* degree of node : no of children it has (0 → 2)
* Internal nodes / non leaf / Non-terminal nodes :
  nodes with degree > 0
* external nodes / leaf / Terminal nodes :-
  Nodes with degree = 0

Binary Tree : degree - 2 children - 0, 1, 2

* Number of binary tree using n nodes:-

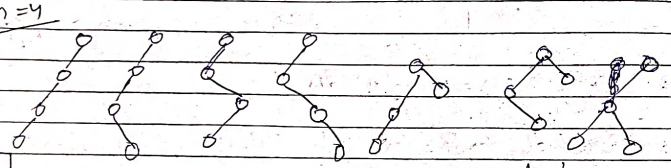
1) Unlabelled nodes:

$n=3$



$T(3) = 5$

$n=4$



and 7 mirror images

$T(4) = 14$

n	T(n)
0	1
1	1
2	2
3	5
4	14
5	42
6	132

$T(5) = 42$

$$T(n) = \frac{^{2n}C_n}{n+1} = \sum_{i=1}^{n} T(i-1) * T(n-i)$$

2) Labelled nodes:

$n=3$

(A) (B) (C) → order matter

$$T(n) = \frac{^{2n}C_n}{n+1} \times n!$$

* Height v/s node

$h = 1$

min: $n=2$

max: $n=3$

$h = 2$

min: $n=3$

max: $n=7$

$h = 3$

min: $n=4$

max: $n=15$

$1 + 2 + 2^2 + 2^3$

$\therefore$ min nodes $= h+1$

max node $= 1 + 2 + 2^2 + \cdots + 2^h$

$= 2^{h+1} - 1$

$$\boxed{h+1 \le \text{nodes} \le 2^{h+1} - 1}$$

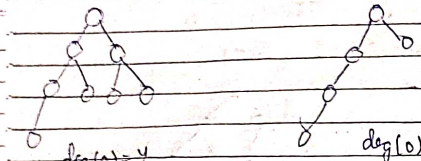
→ If nodes are given

i) Max height $h = n-1$

ii) Min height $h = \log_2(n+1) - 1$

$$\boxed{\log_2(n+1) - 1 \le \text{height} \le n-1}$$

internal nodes v/s external nodes in binary tree :



deg(0) = 4
deg(1) = 1
deg(2) = 3

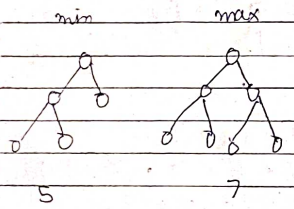
deg(0) = 2
deg(1) = 2
deg(2) = 1

* $\boxed{deg(0) = deg(2) + 1}$

~~# Strict binary tree~~ :- (degree = 0 or 2)

Height v/s nodes :-

h = 2



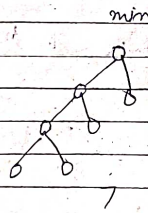
min

5

max

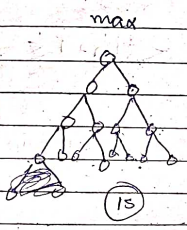
7

h = 3



min

7



max

15

max n = $2^{h+1} - 1$

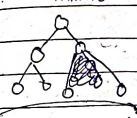
min n = $2h + 1$

$\boxed{2h+1 \leq n \leq 2^{h+1} - 1}$

~~* Height v/s node (SBT) :~~

N is given :- (SBT)

min h



max h

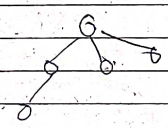
$\boxed{\log_2(N+1) - 1 \leq height \leq \frac{N-1}{2}}$

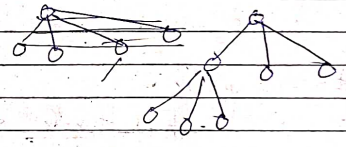
min $\boxed{\log_2(N+1) - 1}$

max $\boxed{\frac{N-1}{2}}$

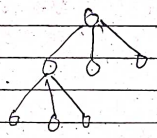
$\Rightarrow$ N (internal node) $_{d \neq 0}$ = No of (external node) $_{d=0}$ $- 1$. $\boxed{e = i + 1}$

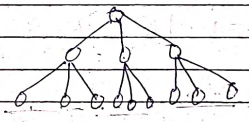
Thrd array tree ($0 \leq deg \leq 3$)





→ Strict 3array tree :





min n = $2h + 1$

max n = $\frac{3^{h+1} - 1}{3 - 1} = 3^0 + 3^1 + 3^2 + 3^3 + \cdots + 3^h$

for strict n-array tree :

* for n-array tree.

$min^m = nh+1$ $\quad max^m = \frac{n^{h+1}-1}{n-1}$

* for fixed n.

$max\ h = \frac{n-1}{m}$ $\quad min\ h = \log_m(nN+1)-1$

* 3-array tree : (ST) : $N(ext) = N(int) * 2 + 1$

* n-array tree : (ST) $N(ext) = N(int)*(n-1)+1$

n-array tree :- $\rightarrow 1+n+n^2+\cdots+n^h$

$$h+1 \le \text{nodes} \le \frac{n^{h+1}-1}{n-1}$$

$$\log_n(1+N(n-1))-1 \le \text{height} \le N-1 \qquad N \rightarrow \text{nodes}$$

strict n-array tree

$$nh+1 \le \text{nodes} \le \frac{n^{h+1}-1}{n-1}$$

$$\log_n(1+N(n-1))-1 \le \text{height} \le \frac{N-1}{n}$$

→ for n array tree $\boxed{e = (n-1)\,i + 1}$

Representation :-

Array

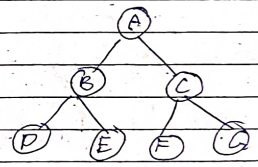
(i) root → ith index
left child → 2i
Right child → 2i+1

(ii) node → ith position
parent → $\left[\frac{i}{2}\right]$

Linked list :-

| lc | info | rc |

```
struct tree {
    char info;
    struct tree * lchild;
    struct tree * rchild;
}
```



→ 7 node
→ 8 null pointer

→ complete binary tree :-
(all level filled, and last level filled from left)
↳ full upto h-1 ; ↳ no blank spaces between elements - representation in array

* Full binary tree : A BT with of height h having max n number of nodes.

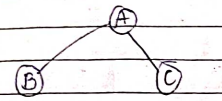
* strict & complete tree has no relation :

* Traversal :-

Preorder → Root then left child → Right child.

Inorder → left child → Root → Right child

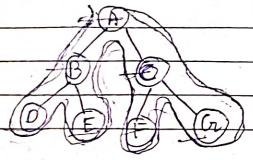
Postorder → left child → Right child → Root



Preorder → A B C

Inorder → B A C

Postorder → B C A



Preorder → A B D E C F G (left)

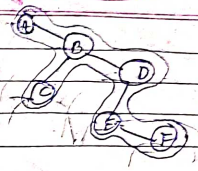
Inorder → D B E A F C G (down)

Postorder → D E B F G C A (right)

```
void preorder (Node *root)
{
    print (root->data);
    preorder (root -> lchild);
    preorder (root -> rchild);
}
```

```
Inorder (root)
{
    Inorder (root -> lchild);
    print (root)
    Inorder (root -> rchild);
}
```

```
Postorder (root)
{
    postorder (root -> lc)
    Inorder (root -> rc);
    print (root);
}
```

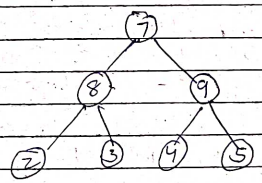


Pre: A B E D E F

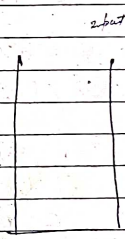
In: A C B E F D

Post: C F E D B A

⇒



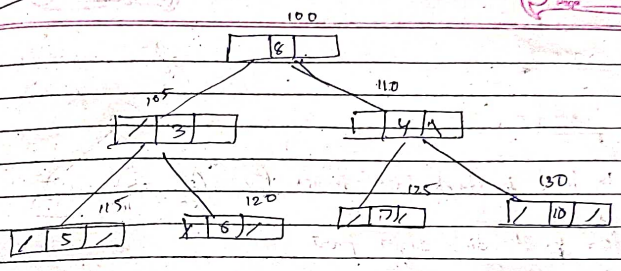
queue



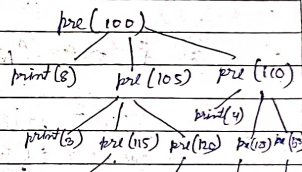
- struct tree, root;
  scanf ("%d", &data);
  root -> info = data;
  qq qq q.

```
void create()
{
  node *p, *t;
  int x;
  Queue q;
  Queue q; scanf("%d", &x);
  root = malloc( );
  root -> data = x;
  root -> lchild = null;
  root -> rchild = null;
  enque(root, &q);

  while (!isempty(q))
  {
      p = dequeue(&q);
    scanf(" leftchild ", &x);

    t1 = malloc( ) ...
    t1 -> data = x;  t1 -> lc = null;  t1 -> rc = null;
     p -> leftchild = t1;  enqueue(&q, t1);

    scanf(" right %d", &x);  // Right child
    t2 -> data = x;  t2 -> lc = t2 -> rc = null;

     p -> right child = t2;  enqueue(&q, t2);
  }
```

04/10/23




```
void preorder(Node *t)
{
  if (t != NULL)
  {
    printf("t -> data");
    preorder(t -> lchild);
    preorder(t -> rchild);
  }
}
```



↳ No of f^n call = 2x3+1 = 7
↳ no of activation recode = n+2

→ Iterative:

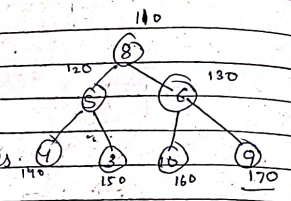
```
void preorder(Node* st)
{
  struct stad st;
  while (t != NULL || !emptyfy(st))
  {
    if (t != NULL)
    {
      print (t->data);
      push (&st, t);
      t = t -> lchild;
    }
    else {
      t = pop(&st);
      print (t -> data);
      t -> t -> rchild;
    }
  }
}
```

→ activation recode = n-1

05/10/23

Postorder Traversal

* Iterative program



→ Need to push in stack two times.

i) for iterating right part

ii) for printing value.

```
void postorder (Node *t)
{
    struct stack st;
    while (t!=NULL || !isempty(st))
    {
        if (t!=NULL)
        {
            if(t->data >=0)
            {
                push(&st,t);
                t=t->child;
            }
            else
            {
            }
        }
        else
        {
            t=pop(&st);
            if (t->data >=0)
            {
                push(&st,-temp);
                t=(-t)->rchild;
            }
            else
            {
                print(t->data);
                t=NULL;
            }
        }
    }
```

typecasting : t = (Node *) t->rchild

t = -(Node *) t

```
if(t!=NULL)
{
    push(&st,t);
    t=t->lchild;
}
```

140	-140 150 -150 160 170
120	-120 130 -130 -150 170
100	-100

4 3 5 10 9 6 8