Topological Sorting
call by.

Activation Record (Do More&More)

VAL:

Variable declaration in Pascal.

Aliasing

Anagram , ADT

go to basics like ADT, Aliasing etc.

Page No.
Date : / /

Ques- WAP in C to return the address of a next which contain data as X.

```
(struct node*) address (struct node *s)
{
    if (s->data == while (s->next != N
    if (s==NULL)
        return NULL
    if (s->data == x)
        return s;
    if (s == NULL) return NULL;
    while (s->next != NULL)
    if (s->next == NULL)
        if (s->data == x)
            return s;
        else return -1;
    While (s->next != NULL)
    {
        if (s->data == x)
            return s;
        else
            s = s->next;
    }
}
```

WC
1
⇒ Time complexity - O(n)
O(1) - BC

OR

```
(struct node*) address (struct node *s)
{
    if (s==NULL) return s;
    else
    while (s != NULL)
    { if (s->data == x)
```

{Merge Sort on Linked list ⇒ nlogn}

Page No.
Date : / /

{Min. time Complexity with linked list = O(n)}

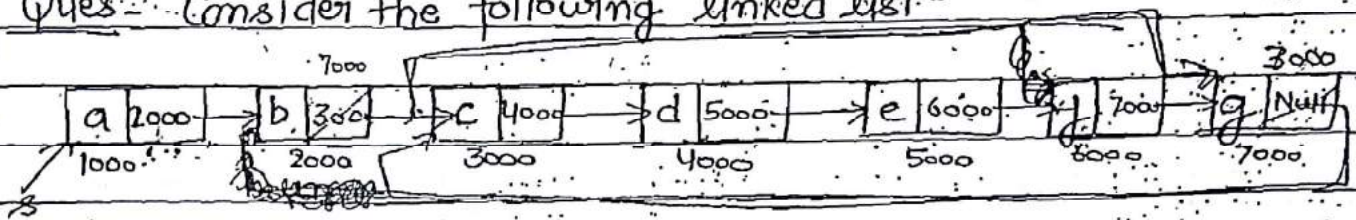
```
↑ find_x (struct node *s)
(struct node*)
    while (s != NULL && s->data != x)
        s = s->next;
    if (s == NULL) return NULL
        else return s.
}
```

(It will return the pos^n of I x present in list)

⇒ {Time Complexity - O(n)} WC

* **→ Triple pointer

Ques:- Consider the following linked list:-




(i) struct node *p;

(ii) p = s->next->next->next;

(iii) p->next->next->next->next = s->next->next;

(iv) s->next->next = p->next->next->next->next

(v) pf((p->next->next->next->next->next->next->next->data)

5000 6000 7000 3000 4000 500 6000

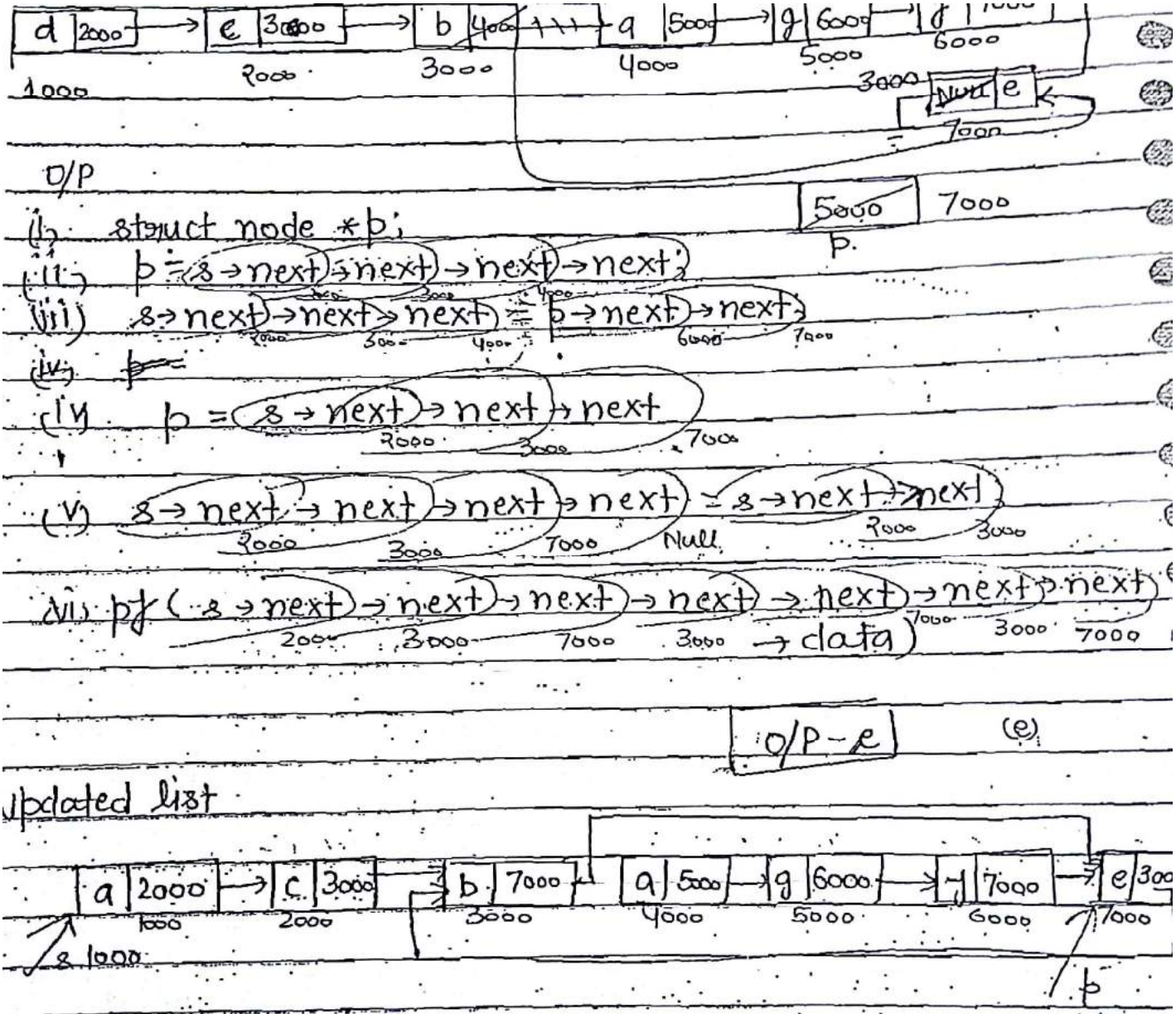
s = 1000 ~~s->next->((*s).next).next.next~~ *(*((*s).next).next).next

(ii) p = 4000

(iii) *(*(*(*p).next).next).next).next

(b)(f)

O/P - f


d 2000
e 3000
b 4000
a 5000
g 6000
f
1000
2000
3000
4000
5000
6000
3000
Null e
7000
O/P
(i) struct node *p;
(ii) p = s→next→next→next→next;
(iii) s→next→next→next = b→next→next;
(iv)
(v) p = s→next→next→next
2000
3000
7000
(vi) s→next→next→next→next = s→next→next
Null
2000
3000
(vii) pf(s→next→next→next→next→next→next→data)
2000
3000
7000
3000
7000
5000 7000
p.
O/P - e
(e)
Updated list
a 2000
c 3000
b 7000
a 5000
g 6000
f 7000
e 3000
1000
2000
3000
4000
5000
6000
7000
s 1000
b

Page No.
Date : / /

Ques - WAP in C to add a node with data x at the end of given linked list.

```
(struct node *) insert (struct node *s, int x)
{  struct node *p, *q;  q=s;
   if(s==NULL)
   {  p->data = x;
      p->next=NULL;
      s=&p;
      return s;
   }
   While(s->next !=NULL)
      s=s->next;
   p->data=x
   p->next=NULL;
   s->next=&p;
   return q;
}
```

(WC, AC, BC)
⇒ Time Complexity - O(n)
But linking take - O(1)

malloc(10B) - M/M of 10B get created in Heap area

malloc (sizeof(struct node))

⇓⇓ Here allocation of memory is done in Heap Area of size struct node of void type. i.e any type of data can be stored.

So it is a void m/m so typecasting is done here.

it will return address of that Memory.

= (struct node *) (malloc (sizeof (struct node))

↓ struct node type, since malloc will return add.

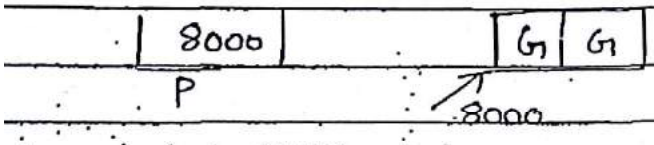
so * is used at type casting.

Let malloc return 8000

8000 + 1 = 8010

Page No.
Date : / /

struct node *P;
P = (struct node *) malloc (sizeof (struct node))
10B
Let 8000
8000 - (only typecasting of void to to struct node type)
ie typecasting of pointer.




printf (P) = 8000
*P = (*P).data ⇒ =x;
(*P).next ⇒ NULL;

Dynamic M/M allocation - M/M creates at run time & remain forever until u free it

Dynamic Time M/M - stack (when f^n over M/M to local variable also deallocated)
heap (m/M will remain after execution complete).

malloc - to allocate M/M dynamically initial value are Garbage

free - to free the created Memory, ie deallocation of M/M.
Here, deallocation of M/M is in user hand.

calloc → allocation of M/M by Clearing i.e M/M get created (alloc) (c).
by initialization of default value to zero.

EXCELLEN

Page No.
Date : / /

Ques - WAP in C to insert a node with data x before a node with data y in given linked list.

```
(struct node *) insert (struct node *s, int x)
{
    Struct node *P, *q, *r ; r = NULL;
    P = (Struct node *) malloc (size of (struct node))      // Creation of node
    P -> data = x;
    P -> next = NULL;
① if (s == NULL)
    return ;
    q = s
    while ( s -> data != y || s != NULL )
    {
    while ( s -> next != NULL )
    { while ( s -> data != y && s -> next != NULL
③ while (s != NULL)
    {
        if (s -> data == y) break;
        else {
            r = s;
            s = s -> next;
        }
    }
    r -> next = p;
    p -> next = s;
    return q;
}
④ if ( s -> next == NULL
        && s -> data == y)
    { r = p;
      p -> next = s;
      return r;
    }
    else
⑤ if (s == NULL)
    return (NO Y found)
}
```



EXCELLENT

Page No.
Date: / /

① struct node *P; *q;
P=(struct node*) malloc (size of (struct node));
*P.data = x;
*q.next = Null

2. if (s==NULL) return NULL. // list empty

3. if (s→data == y) // for node Y at 1st place
{ P→next=s;
s=p;
return s;
}

4. q= NULL

5. while (s→data != y && S→next != NULL)
{ q=s;
s=s←next;
}

if (s→data == y)
{
q→next=p; {Time Complexity - O(n)}
p→next=s
}
else if (no y)
}

EXCELLENT

(Deletion means freeing of...)

Page No.
Date : / /

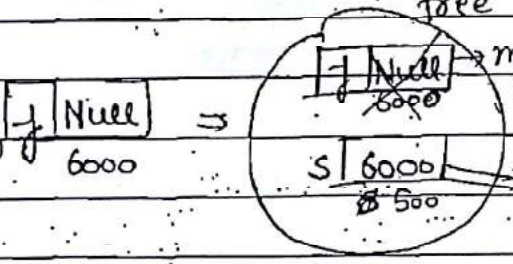
s->next free(s).

Ques WAP in C to delete a node at the end of given linked list.

```
(struct node*) delete_at_end (struct node *s)
{ struct node *s1, *q; q=NULL; s1=s;
  if(s==NULL)
    return;
  while(s1->next!=NULL)
  {
    q=s1;
    s1=s1->next;
  }
  free(s1); s1=NULL
  q->next=NULL;
  return s;
}
```

to free dangling pointer

Time Complexity - O(n)




free(s) - It will delete the m/m location 6000 but s still have data 6000 it is pointing to 6000 from where M/M is freed, but s is still pointing; s is known as dangling pointer.

```
if(s->next==NULL)
{ free(s);
} return NULL;
```

if you dont wanna use q, use (s->next->next)

then you can do like-

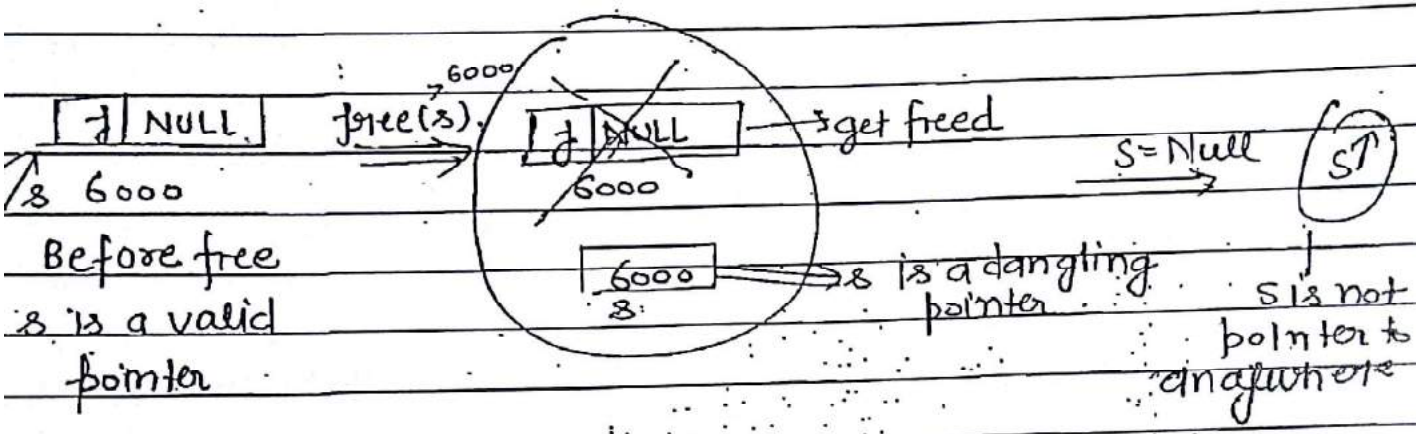
```
q=s
if(s==NULL) q
    return
if (s->next==NULL)
{ free s;
  return NULL;
}
while(s->next->next!=NULL)
{
  s=s->next;
}
p=s->next;
free(p); p=NULL
s->next=NULL;
return q;
```

Modification & Deletion

Page No.
Date : / /

Dangling pointer- A pointer pointing to M/M location which is not there ie if it is freed.

so use : s = NULL to free it.




When M/M is freed & still used it will give Garbage value.

i.e.

before free - print (s, s→data, s→next) ⇒ 6000, ₹, NULL

after free - print (s, s→data, s→next) ⇒ 6000, G, G

s = NULL

print (s, s→data, s→next) ⇒ NULL, segmentation error.

Ques - WAP in C to delete a node which contain data x in the given linked list.

```
(struct node *) delete x(struct node *s, int x)
{
    struct node *p, *q
    if (s == NULL)
        return;
    if (s→data == x)
    {
        free(s);
        s = NULL;
        return NULL;
    }
```

```
q=s
while( q->data != x && q->next != NULL)   (You cannot change the order here because AND operator works on short circuit property)
{
    p=q;
    q=q->next;                            { Time complexity - O(n) }
}
p->next=q->next;
free(q);
q=NULL // freeing dangling pointer.
return s; if(s==NULL)
{
        Now there so return;
    return s
}
```

Ques - WAP in C to move last node of linked list to front of linked list.

```
(struct node *) movelast to first (struct node *s)
{
    struct node *p, *q; q=s;
    if(s==NULL)
        return NULL
    if(s->next==NULL)
        return NULL;
    while(s->next!=NULL)
    { s=s->next; p=s;
                s=s->next;
    }
    ① p->next=NULL;
    ② s->next=q;                  { change of order of change here will change the code }
    ③ q=s;
    return q;
}
```

In array - base ... & no of ... is given
in linked, - only start add... is given

Page No.
Date : / /

Ques- WAP in C to find middle of given linked list.

```
(struct node *) middle (struct node *s)
{
    if (s == NULL) return 0;
    if (s->next == NULL) return -1;
    while (s ... != NULL)
    { count++;                          } O(n)
      s = s->next;
    }
    count = count/2;
    for (i=1; i<count; i++)             } O(n/2)
    {
      s = s->next;
    }
    return (s);
}
```

while (count != 1)

$\Rightarrow$ Total complexity $= O(n) + O(n/2) = O(3n/2) \simeq O(n)$

Last element - stack
first element - Queue

$\frac{3n}{2} + \frac{n}{2} + \frac{n}{4}$

$\frac{3n}{2} + \frac{3n}{4} + \frac{3n}{23} + \cdots$

$3n\left[\frac{1}{2} + \frac{1}{2^2} + \frac{1}{2^3} + \cdots\right]$

$3n\left[\frac{1}{2} \cdot \frac{1-(1/2)^n}{(3n/2)}\right]$

EXCELLENT

Page No.
Date : / /

Algorithm 2 -

| p | q |
|---|---|
| 1 | 1 |
| 2 | 3 |
| 3 | 5 |
| 4 | 7 |
| 5 | 9 |
| 6 | 11 |
| : | : |
| 50 | 100 |
| : | : |
| n/2 | n |

p is always incremented by 1.
q is always incremented by 2
When q reaches at last of the list, p reaches mid of the list.

to check condition for q, skip two element is required.
[q->next->next != NULL]

- If q has to be incremented, q must have two elements after it.
- if q has one element after it it will not incremented.
- If q has zero element after it, it will not incremented.
- either both will incremented or no one get incremented.

mid(s)

Algorithm-

```
p=q=s
while( q!=NULL && q->next!=NULL && q->next->next!=NULL)
{  p=p->next;
   q=q->next->next;
}
cond if(q==NULL)
    return NULL
else if(q->next==NULL)
    q=q->next (to be at end)
return p; // mid of list
return q; // last element
```

O(n/2)

Time complexity = O(n)

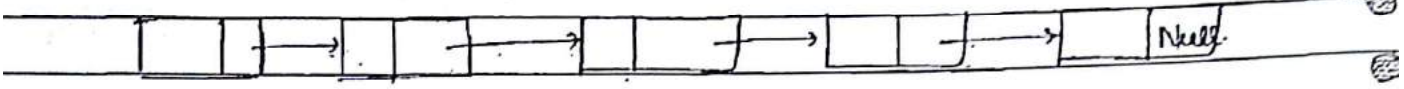
if return floor ceiling value of mid if n is even



EXCELLENT

Page No.
Date : / /

Ques - WAP in C to perform Binary Search in linked list




- List must be sorted.

```
Binary Search(s,x)
    if (s->next==NULL)                    // 1 element
        { if (s->data==x)
            return s
          else return -1 }
    else p=q=s
    While(q!=NULL && q->next!=NULL && q->next->next!=NULL)
    {
        r=p;
        p=p->next;                        } O(n)
        q=q->next->next;
    }
    if(q!=NULL)
        return;
    else if (q->next==NULL)
        q=q->next;
    if (p->data==x)
        return p;
    else (p->data>x)
        Binary Search r->next=NULL;
        Binary Search (s,x);              T(n/2)
    else
        Binary Search (p->next,x);        T(n/2)
```

Tc = O(n) (WC)
Bc = O(n)
TC = O(n)
BC = O(n)

EXCELLENT

~~To find mid~~

```
BS(S, x)
{
  if (s->next == NULL)          ⇒ O(1)
  { if (s->data == x)
      return s;
    else
      return Null;
  }
  else
  { m = mid(s)                  ⇒ O(n)
    if (m->data == x) return m  ⇒ O(1)
    else if (m->data > x)
    { m->next = NULL
      BS(S, x);                 // T(n/2)
    }
    else
    { BS(m->next, x);           T(n/2)
    }
  }
}
```

$$T(n) = \begin{cases} O(1) & ; \text{ if } n = 1 \\ n + O(1) + O(1) + T(n/2) & ; n > 1 \end{cases}$$

Time complexity

$T(n) = T(\frac{n}{2}) + n$

$= O(n)$

// Use Brain Never consigo on your consideration

space complexity $= O(\log n)$



EXCELLENT

(Binary search is applicable to ... structures)

Page No.
Date : / /

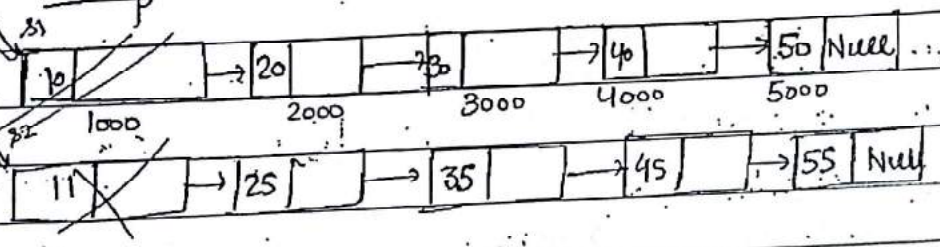
Note- On linked list, Binary Search is possible but it is not efficient because it will take time complexity of O(n) {Linear Search also O(n)}. (BC, WC, AC)

Ques - Write a C program to perform Merge Sort on given linked list.

struct node *) Merge_Sort_LL(struct node *s)

{

Merge -

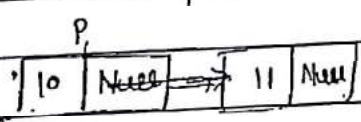



if (s1→data < s2→data)

p = s1. ; s1 = s1→next p→next = Null ; R = p

else p = s2 s2 = s2→next p→next = Null ; R = p.

& soon but R = p → R→next = p;




⇓ O(n) (since n/2 + n/2 moves are here)

Inplace Algorithm since same m/m is used again & again.
No Need of Creating M/M

Initially compare the elmt in list which one is smaller store in p, increment the list & store p in R i.e R is pointing P & again element getting added to R.

Time = O(n) as Total moves = O(n)
↳ Total comparison = O(n)

EXCELLENT

Note - [Merging two sorted ~~subarrays~~ linked list each of size n/2 will take O(m+n) time (BC, WC, AC).] · In place algorithm but in arrays, it is outplace.

Page No.
Date : / /

```
(struct node*) Merge_Sort (struct node *s)
{
    if (s->next == NULL)  ⇒ O(1)
    {
        return s;
    }
    else
    {
    m = mid(s)  ⇒ O(n)
    s1 = Merge Sort (m->next);  ⇒ T(n/2)
    m->next = null;
    s2 = Merge Sort (s);  ⇒ T(n/2)
    S = Merge (s1, s2)  ⇒ O(n)
```

$$T(n) = \begin{cases} O(1) & ; \text{if } n==1 \\ 2T\left(\frac{n}{2}\right) + O(n) + O(n) + O(1) & ; \text{if } n>1 \end{cases}$$

$$T(n) = 2T\left(\frac{n}{2}\right) + O(n) + O(n)$$

$$= 2T\left(\frac{n}{2}\right) + 2n$$

$$= 2n \log n$$

$$= O(n \log n) \quad (\text{Inplace})$$

[It increases the time to solve ~~due to~~ because random a not possible]

Page No.
Date : / /

Ques Consider the following C program-

```
f(struct node *p)
{
    return ((p==NULL) || (p->next==NULL) ||
            ((p->data <= p->next->data) && f(p->next)))
}
```

the above fn f on given linked list p always return 1 then linked list p should be-

1. p contain 0 node
2. p contain 1 element
3. the data in p is in increasing order/ascending element
4. the data in p is in decending order. ||

Ans - (C)

Option a & b are contained in c.

```
return ((p==NULL) || (p->next==NULL) || ((p->data <= p->next->data)
                                        || f(p->next)))
```

any sequence always return 1

Time Complexity = O(n) (WC)
= O(1) (BC)

Ques - WAP in C to perform Selection Sort on given linked list.

```
(struct node *) Selection_Sort(struct node *s)
{
    while if(s->next==NULL)
        return s;
    while(s != NULL)
```

EXCELLENT

```
        count++;
        s = s->next;
    }
    for(i=1; i<count; i++)
    while(count!=0)
    { p = s ~~->data~~
```

ModifiedSelection Sort(s)  (Modified Selection Sort)

```
{
    for(p=s; p!=NULL; p=p->next)
    {
        for(q=p->next; q!=NULL; q=q->next)
        {
            if(q->data < p->data)
                swap(p->data, q->data)
        }
    }
}
```

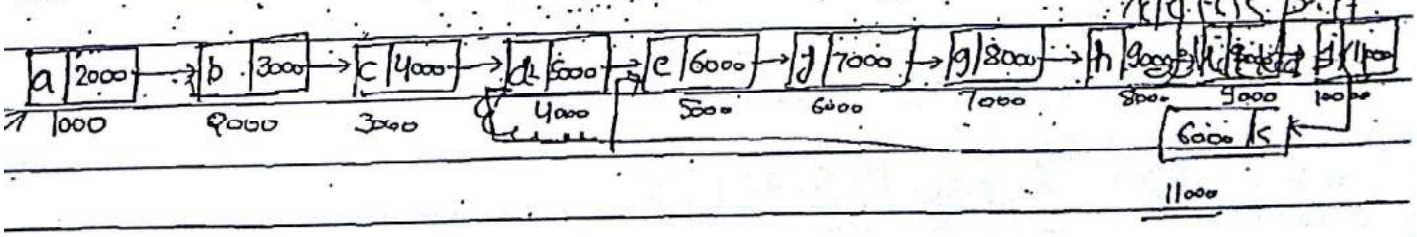
Time complexity = $n + n-1 + n-2 + \cdots 1$

$= O(n^2)$

Swaps = $O(n^2)$ (Worst case) &

But in case of array, swaps were $O(n)$ → in actual ($\leq n-1$)

Ques- WAP in C to find loop in given linked list.






EXCELLENT

Page No.
Date: / /

Algorithm-1:

Assume each node has three fields of linked list (data, flag next). & flag field is 0 for all model initially.

Visit each node & check flag field

```
if (flag == 0)
    flag = 1
else
    by (loop is found)
```

Here a extra field is added it seems to be added but actually when LL is created for project initially five fields are ~~required~~ there so that when required, project can be expanded.

if no extra M/M is there, we can use resize fn to resize the structure.

To initialize the flag with 0, we can use calloc memory.

{Time complexity = O(n)}

Note

TC of kruskal Algorithm when edges weight ~~con~~ are already sorted. O(E) + (E+V)

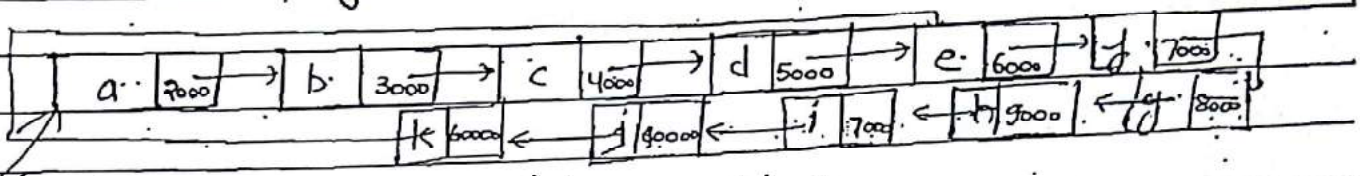
↳ by ~~cycle~~ DFS cycle also checked

O(E+V)

Algorithm 2- Apply BFS or DFS

Page No.
Date : / /

Algorithm3- p & q will start from same place p will go slower than q. if p & q never meet, then no cycle otherwise no cycle-




p q p increment by 1
q by 2

1000 — 1000
2000 ← 3000
3000 → 5000
4000 ← 7000
5000 → 9000
6000 ← 11000
7000 — 7000 (cycle is present)



- You can increment q by any value, it is possible that at end it is performing many cycles

p: ——
q: ——

1. p=q=s
2. p by 1
   q by 2
3. if(p==q)
   printf(cycle)
   else
   goto step2.

Time complexity = O(n)
Space Complexity = O(1)
Time complexity is due to p only.

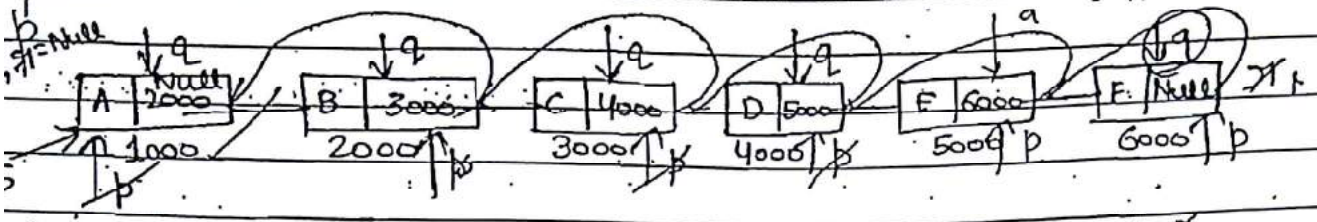


EXCELLENT

p=Null q=s
s→next=p
p=s

Page No.
Date : / /

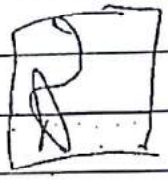
Ques:- WAP in C to reverse a linked list.




```
(struct node *) Reverse (struct node *s)
{
    p=null;
    r=s→next;
    while (r != NULL)
    { s→next=p
      p=s; s=r;
      r=r→next;
    }
```

R P
R P P
R P P
and soon

```
    p=s
    q=r=Null
    while (p != next)
    {
      r=q;
      q=p;
      p=p→next
      q→next=R;
    }
    S=q;
    return s
```



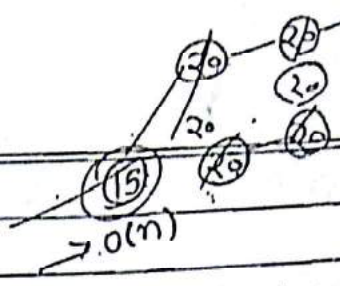
~~Time Compl~~

{Time Complexity - O(n)}
(WC, AC, BC)

To print data of linked list in reverse order (singly linked list)
TC = O(n) + O(n)
↑ to reverse ↑ to print

To print data of doubly linked list in Reverse order
= TC = O(n)

EXCELLENT




## HomeWork-

1. WAP in C to perform concatenation, Union & Intersection b/w two linked list → O(nlogn)
2. Cardinality algorithm → O(n)
3. Membership algorithm → O(n)
4. Write a P in C to implement Linked list using array
5. Write a P in C to implement stack using linked list.
6. Write a C program to implement Queue using LL.

$3x^4 + 5x^2 + 7$ : [3|4] → [5|2] → [7|0] → Null

$10x^5 + 20x^3 + 12x$ : [10|5] → [20|3] → [12|1] → Null

to add two polynomial-

```
if L1→pow == L2→pow
    l3→coff = L1→coff + L2→coff
    l3→pow = L1→pow
else
if L1→pow > L2→pow
    l3→coff = L1→coff
    l3→pow = L1→pow
else
    l3→coff = L2→coff
    l3→pow = L2→pow
    L = L→next;
```

Time complexity = O(m+n) (when no add)

↓ (WC)

O(m) best case (when all power getting added to other list)

due to move

WAP in C to perform polynomial addition & multiplication using linked list- → O(m,n)

(each value get multiplied to other)

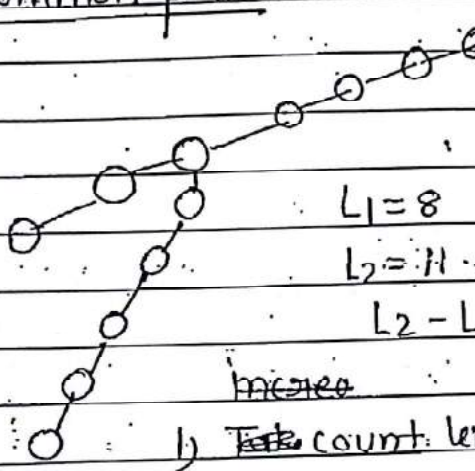


EXCELLENT

Page No.
Date : / /

~~Drawback of singly linked list~~-

finding common point:-




$L_1 = 8$

$L_2 = 11$

$L_2 - L_1 = 3$

$L_2 - L_1 = 3$
increase 3 to $L_2$
& increase both

1) ~~Take~~ count length of $L_1$ & $L_2$
2) find $L_1$ & $-L_2 = c$
3) find whether $L_1$ or $L_2$ greater
4) initially increase greater than c. time
5) then increase both at a point of time
they will meet
7) start count from point of meeting

Complexity:- $O(m) + O(n) + O(n-m)$

- $O(m)$ ↓ to count no of elmt
- $O(n)$ ↓ to count no of elmt
- $O(n-m)$ ↓ (to ~~go~~ calculate length)

Drawback of singly linked list-

1.> We cannot go back because every node contain only one pointer & that pointer also contain next node address.
2. In singly linked list, last node next part is always null i.e., we are not utilising it properly.

To eliminate above two drawbacks, we are moving toward circular singly linked list.

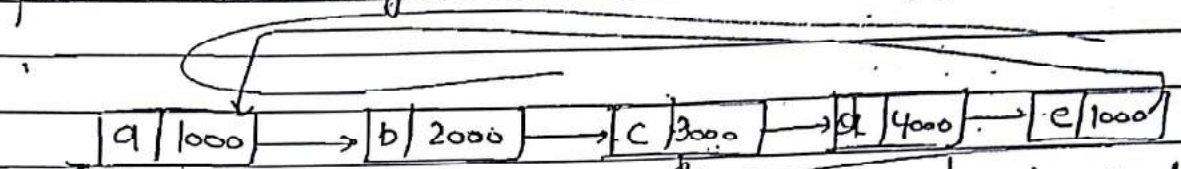
EXCELLENT

Do each operation of single Linked list to circular LL

Page No.
Date : / /

## Circular Singly linked list-

- In single linked list, last node next part, if we restore first address then it became circular singly linked list.




due to this, it is possible to traverse previous node as well.

- We can go back by visiting last node and then first node & so on. but it will take O(n) time.

in doubly linked list, visiting last node is O(1) but it may take some space.

So Here, to check end of ~~loop~~ list-

```
p=s
While (p->next != s)
    p=p->next;
```

p will finally point at last node.

to add a node at first of LL

q (node created) ⟹ O(1)

```
p=s;
While (p->next != s)      // goto last node T(n)
    p=p->next;
p->next=q;                // linking node
q->next=s;                // O(1)
```

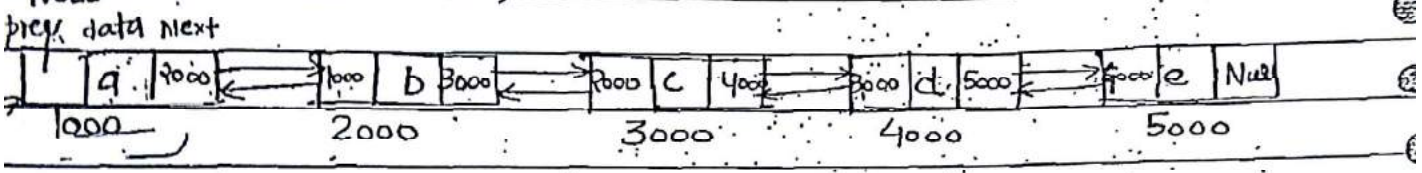
→ TC = O(n)

EXCELLENT

Date : / /

An operation in Singly linked list which is independent of length of list = Insertion at front.

Doubly linked list- struct node
{
~~int data;~~
struct node *prev; int data;
struct node *next;
};




o print data in reverse • go to last
o print by s=s→prev

first node = s→prev = NULL
Last Node = s→next = NULL

Ques) - WAP in C to insert a node with data x at end of DLL.

```
struct node *q = (struct node *) malloc (sizeof(struct node));
q => data = x
q→prev = q→next = NULL
if(s==NULL)
    return NULL
while(s→next != NULL)
    s = s→next;          } O(n)
s→next = q;
q→prev = s;
```

TC = O(n)

EXCEL

Page No.
Date: / /

Ques WAP in C to insert a Node with data x before a node with data y in DLL.

```
struct node *p;
p = (struct node *) malloc (size of (struct node));
p->data = x;
p->prev = p->next = NULL;
while (s->data != y && s->next != NULL)
    s = s->next;
if (s->data == y){
(s->prev)->next = p;
p->prev = s->prev;
p->next = s;
s->prev = p;
}
```

order is important

connect outside people first

```
p->prev = s->prev
p->next = s
s->pre = p
p->pre->next = p
```

order matters at 1st & 4th stmt (1,3)

Ques
WAC to delete a node at last in DLL.

```
① while (s->next != NULL)
      s = s->next
② (s->prev)->next = NULL
③ free(s);
④ s = NULL;
```

Ques - Delete a node in middle

```
while (s1 != NULL)
{ c++;
} s1 = s1->next;
c = ⌈c/2⌉;
while(c != 1)
{ s = s->next;
```



EXCELLENT

3-Nov

Date : / /

```
(s->prev)->next = s->next;
s->next s->prev
(s->next)->prev = s->prev;
free(s);
S = NULL;
}
```

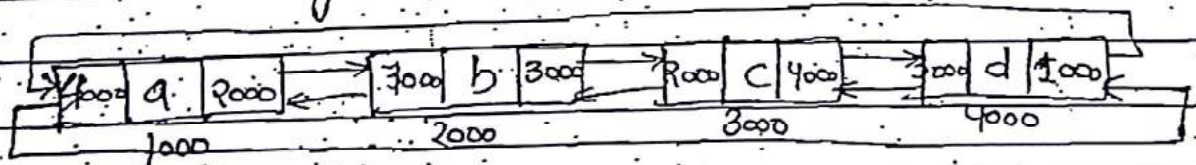
Ques- WAP in C to delete a node with data x in DLL.

```
(struct node*) delete(struct node*s)
{
    while (s->data != x && s->next != null)
        s = s->next;
    if (s->data == x);
    {(s->prev)->next = s->next;
     (s->next)->prev = s->prev
     free(s);
     S = NULL
    }
    else if (s->next == NULL)
        No Node with data x
}
```

order is important

only Modification

Circular Doubly linked list-




| | CLL | DLL | CDLL | |
|---|---|---|---|---|
| 3-4 | O(1) | O(1) | O(1) | No random access possible as to visit 10th node you have to traverse 10 node. |
| 1-3 | O(n) | O(1) | O(1) | |
| ton | O(n) | O(n) | O(n) | |
| 1 to 1 | O(1) | O(n) | O(1) | |

EXCE

Page No.
Date : / /

In Circular DLL, O(1) time to add node at last.

```
p = s → prev
p → next = q
q → prev = p        q - new node
q → next = s
s → prev = q
```

add a node at start = O(1)

: middle

Delete ~~all~~ a node at start

In Circular DLL - O(1) time to concatenate two list.

```
(s1, s2)
p = s1 → prev;
p → next = s2;
s2 → prev = p
q = s2 → prev;
s2 → prev = p;
s1 → prev = q;
```

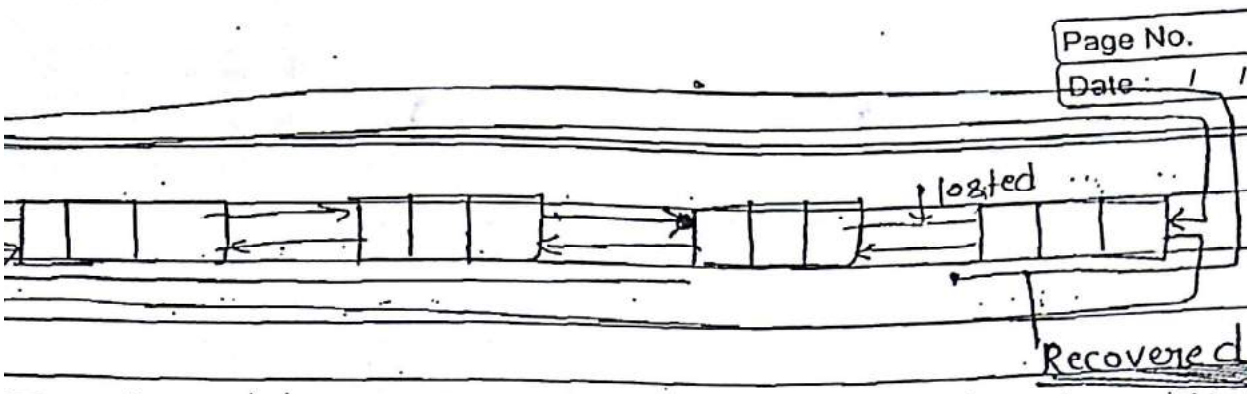
Concatenate DLL - O(m)

In circular linked list - O(m+n)

because two list has to be traversed

CDLL - if a link get lost you can use another link to recover.

The greatest advantage with CDLL or DLL is if you lost one pointer, with help of another pointer, we can get back. (there is a possibility

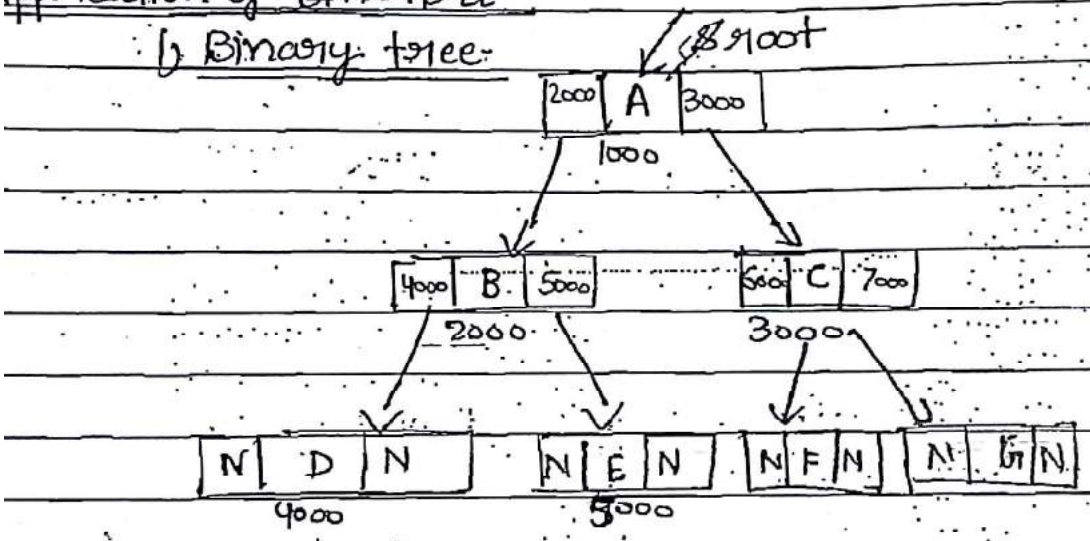



There is no possibility of recovery of link in single linked list.

but if you lasted both pointers of a node in list, you cannot recover even with CDLL.

Application of ~~Binary~~ DLL-

1) Binary tree-




In DLL, a small change is done in DLL, it does not point back to previous node but it point to another node

Each node is pointing to two node, acting as child.

prev → left pointer

next → right ptr

When a tree is given, address of root is given i.e starting node is given.

Page No.
Date : / /

```
Struct BTnode
{
    struct BTnode *lc;
    int data;
    struct BTnode *rc;
};
```

print(root) - 1000
root → lc = 2000
root → rc = 3000

if (root == NULL) tree is empty.
if (root → lc == NULL && root → lc == NULL) a single node.
if (node → lc != NULL || node → rc != NULL) ⇒ Internal node
if (node → lc == NULL && node → rc == NULL) ⇒ leaf node.

Height = 2 (path length from root node to leaf node)
||
No of level - 1

Ques. To find out left ~~node~~ most node of given tree:

```
while (root → lc != NULL)
    root → root → lc
```

∴ Here random access not possible, so traverse is ~~also~~ only done from root.
Use linked list for binary tree, when it is not almost complete binary tree or complete binary tree because it will save space.
No formula to access parent the node.
Use Array for BT where BT is ~~also~~ almost or complete binary tree.
if CBT or ACBT, we used array, else use linked list only.

EXCELLENT

When No Mention about BT- use linked list

Page No.
Date : / /

## Tree-Recursion

Q- WAP in C to count total no of leaf node in given binary tree

```
int leaf(structnode *root)   m=0 initial
{
  if(root==NULL)
    return(0);
  if(root->left==NULL && root->rc==NULL)
    return 1;
  else
    m = m + leaf(root->left) + leaf(root->right);
  return m;
}
```

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

(BC, WC, AC) = O(n) because total tree has to be visited once.

int internal (Unbalanced tree)

Worst case- $T(n) = T(n-1) + c$
$= O(n)$

Termination condn for leaf no tree = leaf nodes

Stack space = O(log n) BC
O(n) WC

Ques WAP to count non leaf node in given tree-

```
m=0;
int non-leaf(struct BTnode *root)
{
  if(root==NULL)
    return(0);
  else if(root->left==NULL && root->rc==NULL)
    return 0;
  else
  {
    m = 1 + non-leaf(root->lc) + nonleaf(root->rc);
  }
  return m;
}
```

Time Complexity = $2T(\frac{n}{2}) + c$ (Best case)

= O(n) (Balanced tree

Stack space = O(log n) (Balanced tree)

EXCELLE

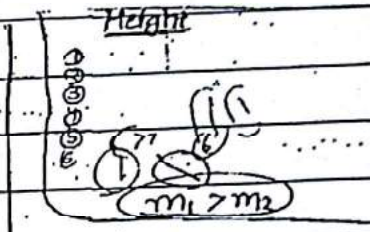
Page No.
Date : / /

Worst case- when tree is an unbalanced tree.

TC; $T(n) = T(n-1) + c$
$= c \cdot n$
$= O(n)$

Space of stack = O(ton) (due to n levels)

for any tree, Recursion is used & leaf node serves as termination (coz you cannot move further)




Ques WAP to find total no of nodes

```
total node(){
  if (root==NULL) return 0
  else
    m = 1 + total_node(root->lc) + totalnode(root->rc);
}
```

or

```
{ if (root==NULL) return 0
  if (root->lc==NULL && root->lc==NULL)
    return 1
  else
    m = 1 + totalnode(root->lc) totalnode(root->rc);
}
```

```
1. if (root == Null) return 0
   if (root->lc == NULL && root->rc == NULL)
        return 1;
   else
   { xL = CNLN(root->lc)
     xR = CNLN(root->rc)
     c = 1+xL                  ⇒return leftmost path visited
     return(c);                  count.
   }
```

{ if c = xL → leaf node in leftmost path.
  c = 1+xL ⇒ & return(0) ⇒ non-leaf node in leftmost path }

Ques- WAP in C to count height of a given BT

```
int height_of_BT(struct node *root)
{
    if (root == NULL)
        return 0;
    if (root->lc == NULL && root->rc == NULL)
        return 0;
    else
    {
        hl = 1 + height(root->lc);
        hr = 1 + height(root->rc);
    if (hl > hr)
        return(hl);
    else
        return(hr);
    }
```

(See the code for)

Page No.
Date: / /

Height of a Binary Tree = the length of the longest path from root to leaf node.

TC → $T(n) = 2T(\frac{n}{2}) + C$ (Balanced Tree)

$= O(n)$

$T(n) = T(n-1) + C + T(1)$ (UnBalanced Tree)

$= O(n)$

(BC, AC, WC)

Stack Space- O(logn) (Best case Balanced tree)

O(n) (Worst case unbalanced tree)

You can find height by finding no of level if you wanna find no of level just count leaf node as well then finally subtract by 1.—

```
xL = height(root->lc)
xR = height(root->lc)
c = 1 + max(xL, xR)
```

To count no of levels-

```
1 if (root == Null) return 0
2 if (root->left == Null && root->rc == Null)
      return 1;
3 else { xL = level(root->lc);
         xR = level(root->rc);
         c = 1 + max(xL, xR);
         return c;
       }
```

if leaf node return 1o → it will give (height + 1o)

if c = 1 + max(xL, xR)

return 1o in (2) then (6 times height + 1o)

Use Boom

EXCELLENT

Page No.
Date : / /

Ques Write a C Program to verify given BT is Strict BT or node.

```
bool Verify_Strict_BT(struct node *root)
{
1. if(root == Null)
       return 1;
2. if(root->lc == Null && root->rc == Null)
       return 1;
3. else
   {
       x1 = Verify_Strict_BT(root->lc)+1
       xR = 1+ Verify_Strict_BT(root->rc)+1
       if(x1 == xR
   }
3. else { if(root
       a = Verify_Strict_BT(root->lc);
       b = Verify_Strict_BT(root->rc);
       if(a == b)
         return 1
       else
         return 0;
SBT(root)
{ 1. if(root == NULL)
       return 1;
2. if( root->lc == NULL && root->rc == NULL)
       return 1;
3. else
   { if( root->lc != NULL && root->rc != NULL)
     return( SBT( root->lc) && SBT(root->rc) );
     else
       return 0;
}
```

EXCELLENT

You cannot write the boolean exp by dividing in two sub expression (boolean exp) donot divide) so time & space get wasted as if SBT(root->l) && (SBT(root->l). Page No. Date: If it → 0 → no need of doing right traversal.

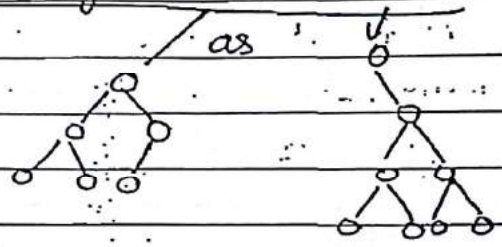
Time Complexity:- When it is strict binary tree

$$T(n) = 2T\left(\frac{n}{2}\right) + c$$

$$= O(n) \text{ (Worst case)}$$

When it is not a strict binary tree

it may be O(n) or O(1)




So: Time Complexity; { BC = O(1) ; WC = O(n) }

If code is written as

```
a = SBT(root->lc)
b = SBT(root->rc)
c = a && b
```

then TC; T(n) = O(n) (WC, AC, BC)

because no use of property of short circuit.

Modification:- if (root ——— )

return (SBT(root->lc)

check whether leftmost path is SBT.



EXCELLENT

Page No.
Date : 7 / .

WAP in C to verify given two Binary trees are equal or not

```
equal (root1, root2)
{
    if(root1==NULL && root2==NULL)
        return 1;
    if((root1->lc==NULL && root1->rc==NULL) && (root2->lc
    ==NULL && root2->rc==NULL))
    { return 1;
        if(root1->data == root2->data)
            return 1;
        else
            return 0;
    }
```

own

```
① if(root1==NULL && root2==NULL) return 1;
② if(root1==NULL && root2!=NULL) return 0;
③ if(root2==NULL && root1!=NULL) return 0;
④ if((root1->lc==NULL && root1->rc==NULL) &&
    (root2->lc==NULL && root2->rc==NULL))
  { if(root1->data==root2->data)
        return 1;
    else
        return 0;
  }
⑤ else
  { if(root1->data==root2->data)
    {
      return (equal(root1->lc) && equal(root2->rc));
    }
    else
      return 0;
  }
```

root1->lc  root2->rc

EXCEL

Page No.
Date : / /

Here it is not important to put fourth cond, it may terminate when root is empty.

equal($r_1$, $r_2$)

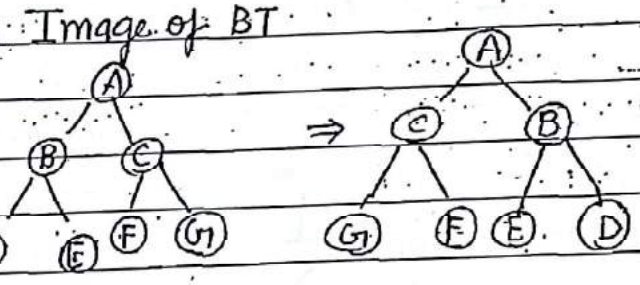
① if ($r_1$ == NULL && $r_2$ == NULL) return 0;

② if (($r_1$ == NULL && $r_2$ != NULL) || ($r_1$ != NULL && $r_2$ == NULL))
return 0;

else if ($r_1$->data == $r_2$->data)
return (equal($r_1$->lc, $r_2$->lc) && equal($r_2$->lc, $r_1$->lc))
else
return 0;
}

TC, $T(n) = O(n)$

Ques - Mirror Image of BT




~~Mirror Image~~

Write a Program in C to convert given binary tree into its mirror image. mirror(root)

1. if(root == NULL)
return NULL;

2. if (root->lc == NULL && root->rc == NULL)
return root;

3. else
{
~~swap(root->lc, root->rc)~~ // swaping of nodes
~~l1 = mirror(root->lc);~~
~~l2 = mirror(root->rc);~~ l = root->rc
root->rc = mirror(root->lc);
root->lc = mirror(root->rc);
}

EXCELLENT

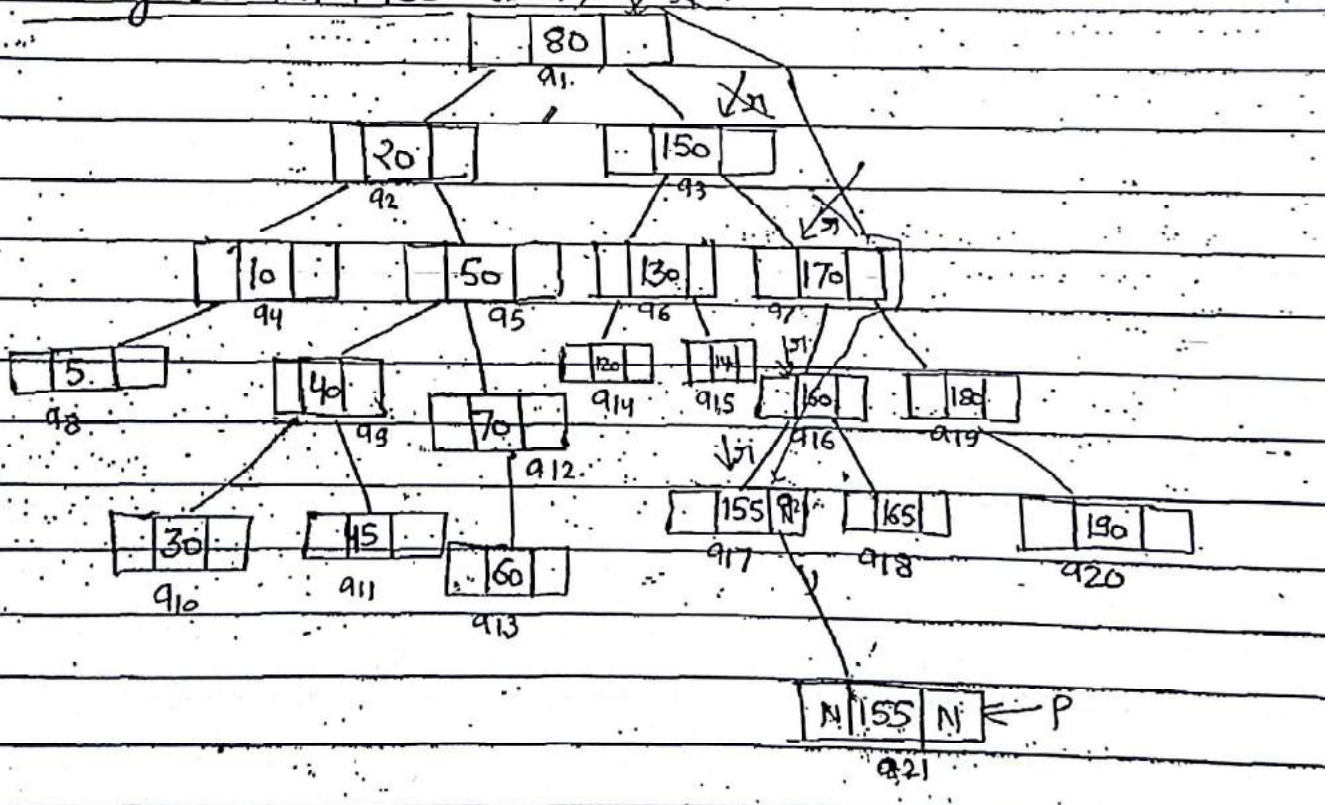
Page No.
Date : / /

```
Mirror I (root) :
{
  if (root=NULL) return root
  if( root→lc == NULL && root→rc = NULL)
      return @root;
  else
  { t = root→rc;
    root→rc = Mirror-I(root→lc);
    root→lc = Mirror-I (t);
  }
```

T.C, T(n) = O(n)

- To convert into Mirror image
  - firstly convert the left tree store to right of root.
  - Again convert right to mirror image & store to left of root.

Binary Search tree - (BST)



Page No.
Date : / /

* BST with n nodes, Minimum level = $\log_2(n+1) \approx \log n$
Maximum level = $n$

2. AVL tree with n nodes, no of level = $\log n$ (Max = Min) (because it is a balanced BST)

3. To find smallest person in BST, goto LeftMost node

```
while(x->lc != NULL)
{   x = x->lc
```

[ Best case - O(1)
Worst case = O(n) (all element in left side )]

4. To find largest element in BST, goto rightmost node

```
while(x->rc != Null)
x = x->rc
```

[ Best case = O(1) (root itself is greatest)
Worst case = O(n) (all elmt in right side )]

5. [To find smallest person in Binary tree,
O(n) (BC, WC, AC)
(no all elmt. has to be traversed)]

Maximum element - O(n)
(WC, BC, AC)

6. Smallest element in AVL tree = $O(\log_2 n)$ (~~Maximum~~ minimum)
(BC, WC, AC)

O(1) - not possible as no case ( \ ) ( / )

EXCELLENT

If tree is given, if not mentiod, it is provided with LL.

Searching an element x

- Binary Tree - Worst Case - O(n) ($2T(n/2)+1$)
  - Best Case - O(1)
- Binary Search Tree - Worst case - O(n) → ($T(n-1)+1$)
  - Best case - O(1)
- AVL tree - Worst Case - O(log n) ⇒ $T(\frac{n}{2})+1$
  - Best Case - O(1)

To say, element x is not there, (Best case)

- Binary Search Tree - BST - O(1) when (Let I was searching for 48 & root = 80 & root → lc = NULL)
  - & Worst case - O(n)
- Binary Tree - Worst case - O(n)
  - Best case = O(n) (all elmt to be checked)
- AVL tree - Worst case = O(log n)
  - Best Case = O(log n)

Insertion of a node - • Initially create a Memory
• traverse to Right pos$^n$ upto u find Null
link node

Best case = O(1)

if Root - 80 & wanna insert 1000

Root → rt = Null.

insert it.

Worst case = O(n)

check Root
is Root 80 158
no
greater than
root
go right
if right = NULL
insert there

EXCE

Algo:- 1. Create a node ⇒ O(1)
2. Traverse to Right posⁿ, Find place = O(n)
3. Links ⇒ O(1)
O(n) (WC)

In AVL tree, Insertion of a node,
Best case = O(log₂n) (because only at last level, null will be there & insertion is done when null is there)
Worst case = O(log₂n)

IN BST, always insertion will be done always at null place.

Searching an element in mm heap = O(n)

It is not the case that value close to root will be placed closed to root in tree

Ques- A element to be inserted in mm heap but checked first whether exist or not.
⇒ TC = O(n) + O(logn)

In BST TC = O(n). (to search to find it is there + insert as after finding null you can directly insert it.

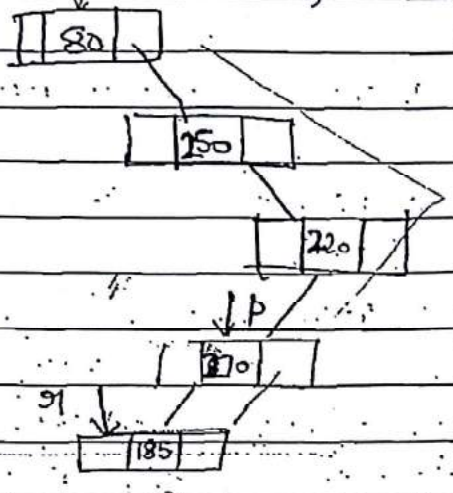
In AVL TC = O(logn) as (to search 'null' or to know it is there or not logn time & then insertion is directly done)

Data Structure- Insertion an element if data not there = O(logn)

Page No.
Date : / /

## Deletion of a Node-

- Deletion of node with 0 child
- Delete x
- Find x
- Store parent of x in p (going with find) (because it is not possible to come back)

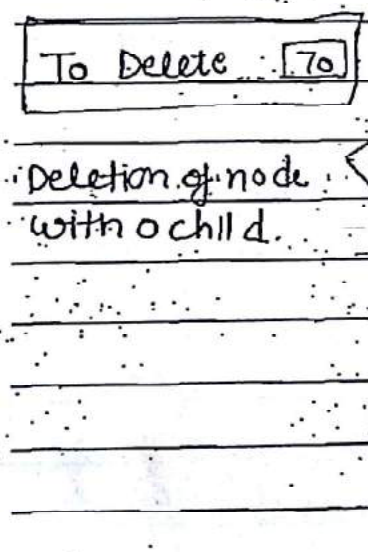



O(n) time to search
- Worst case

O(1) time to search
- Best case

```
p->lc=null;
free(p,r);
r=NULL;
```

⇒ TC = O(n) Worst case




- to find x
- check for posn of child (left, right)

```
p->rc = r->lc
free(r)
r=NULL
```

T(n) = O(n) (Worst case)

EXCELLENT

• Deletion of node with two child -

predecessor of a node - greatest Elmt in left subtree
Successor of a node - smallest elmt in right subtree.

5 10 20 predeccessor is found by going rightmost path in left subtree.
Successor by going leftmost node in right subtree.

- ~~Delete the node~~
- ~~Del Replace by~~

- Replace the node by its predeccessor or successor
- Delete the node get replaced.

A node is said to be predeccessor ~~left poin~~ if Right ptr is null; left ptr may or may not be null.

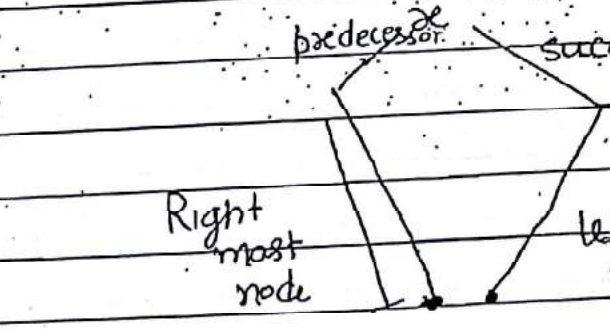
- predeccessor is always leaf (No)

Successor - left Most pointer must be null.
- Right Most pointer may or may not null.

predeccessor or successor of a node on maximum can have one child only.

Best case - find ~~minor~~ =O(1)
find predeccessor =O(1)

Ques



- If finding element takes O(n) then finding predecessor will take O(h) time only as no of levels are mostly.
- if finding element takes O(1) time then finding predecessor can take O(n) in worst case

Page No.

## Deletion - Algorithm -

① If

① Find element(x) ⟹ O(n), O(1) → WC, → BC

② (i) 0-child - delete simply by replacing Null at parent node. ⟹ O(1)

② (ii) 1-child - ~~Come~~ delete x by connecting graph father & grand child. ⟹ O(1)

(iii) 2-child - (a) Find successor or predecessor
(b) replace x data by predeccessor data or succesor data.
(c) delete the node (ie predecessor or successor)

Total time taken for finding an element & predeccessor = O(n)

So on total - Delete time complexity = O(n) (WC)
O(1) = Best case

it is never possible that finding x = O(n) & finding predeccessor = O(n) as No of level is only O(n) but not 2n.

To Create BST with n elements, $O(n^2)$ time required (n insertion & each insertion will take n time);

EXCELLENT

Page No.
Date : / /

AVL Trees - • Balanced binary search trees. (left height & right height must be equal)

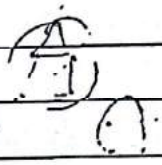
(Adelson, Velskey, Landis)

Balanced BST = Height Balanced BST (no of level = log n)

A BST Balanced is said to be height balanced if

H(LST) - H(RST) ⇒ 0, +1, -1

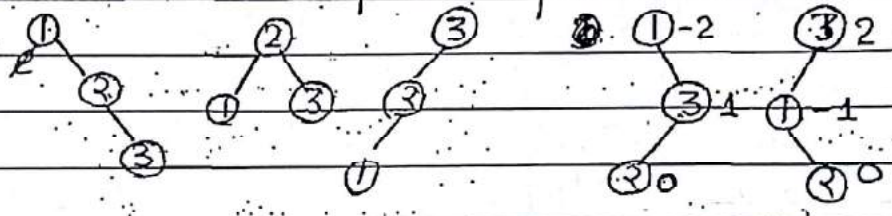
[Height of left subtree] — Balanced factor

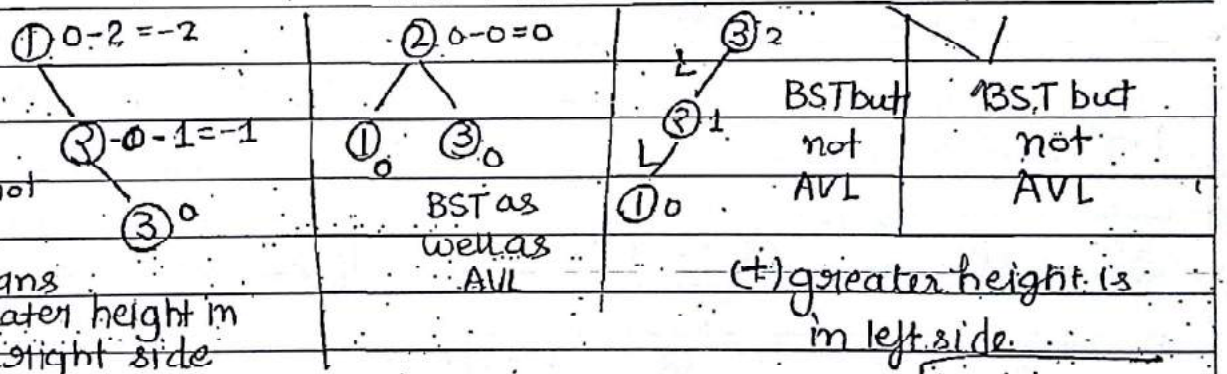


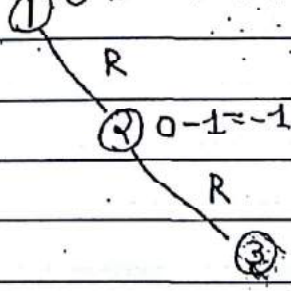
Balanced factor = H(LST) - H(RST)

= 0, -1, +1 only

Example - n = 3 {1, 2, 3}. find all possible BST.




(It is RR problem because right side is incremented by 2)

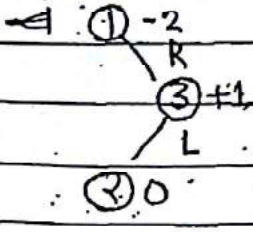


EXCELLENT

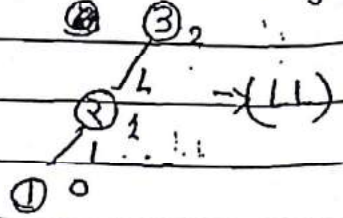
Each AVL is BST but each BST is not AVL

Page No.
Date:

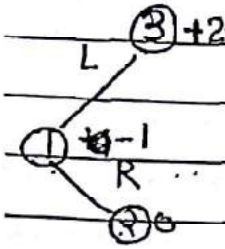
•start giving ~~name~~ from starting if go on to zero




(RL problem)



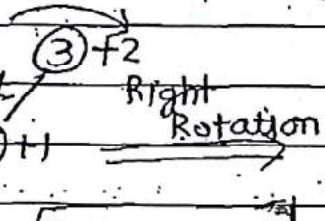

→ (LL)



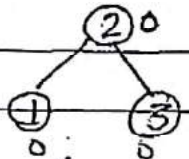

⇒ (LR problem)

If LL → Rotate Right
RR → Rotate Left

## LL problem-




2 → rc = 3
3 → lc = Null




AVL

the node which have problem ie which is not balanced. Rotation it.

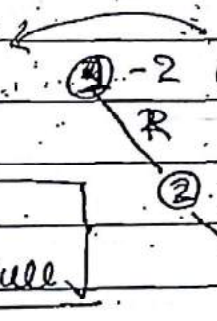
After rotation, parent A & child B get rotated as Parent B child A

When parent fall down in right side ⇒ Right Rotation

T(n) = O(1)

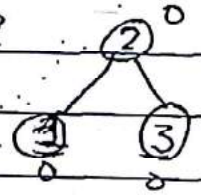
## Left Rotation

## RR problem



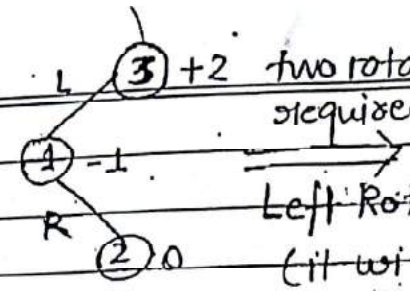

2 → lc = 1
1 → right = Null

Left Rotation



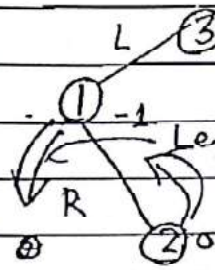

Time complexity = O(1)

## LR problem-

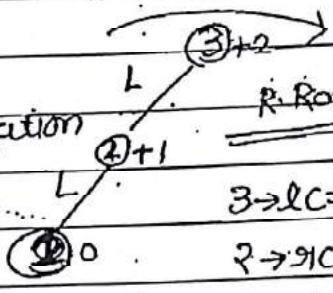



two rotation required.

Left Rotation + Right Rotation
(it will convert it to LL)
(then it'LL) get solved

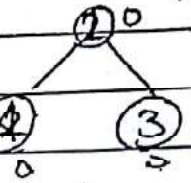



Left Rotation




L Rotation

R Rotation




1 → rc = Null
2 → lc = 1
3 → lc = 2

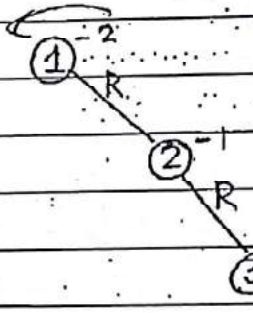
3 → lc = Null
2 → rc = 3

Time Complexity = O(1)

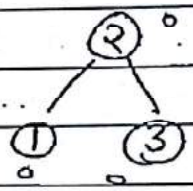
O(1)

## RL problem


Right Rotation




Left Rotation




1 → rc = 2
2 → rc = 3
3 → lc = Null

2 → lc = 1
1 → rc = Null

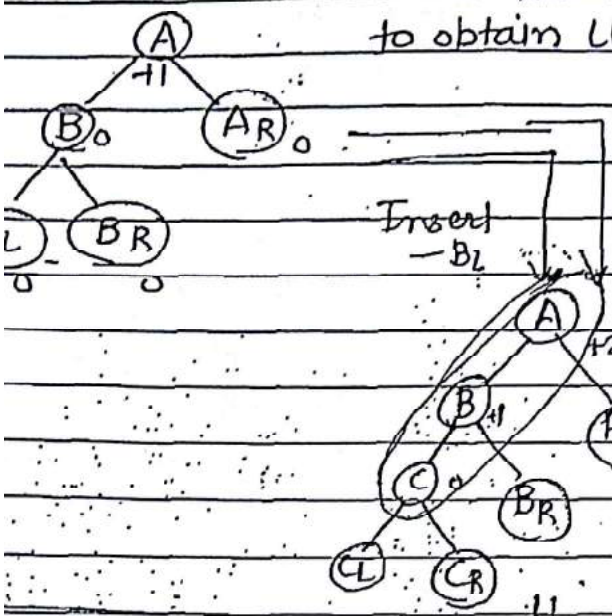
Time Complexity = O(1)

After Rotation, parent & child get exchanged

Page No.
Date : / /

## Insertion - LL Problem -

$A_R$, $B_L$ & $B_R$ are 3 AVL trees
to obtain LL problem add Node to $B_L$.




Insert $-B_L$




Add a node to AVL may or may not create a problem

$O(1)+O(\log n)+O(1)$

$O(1)$ ... creation of node

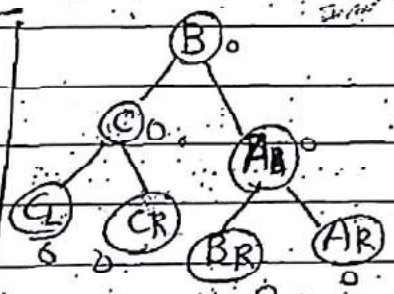
$O(\log n)$ ... finding link nuel

a Tree has Balance for +1, it is close problem ie not an tree)

~~Retro~~ check for AVL
$O(\log n)$

Rotate

Right Rotate

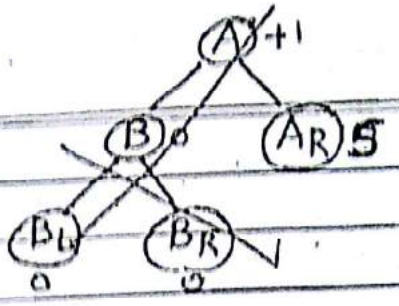



here $B_R$ must low the same path as it was wing in riginal LR (from $B_R$ to root)

∴ Total comp=
$1+\log n+1+\log n$
$=2\log n$
$=O(\log n)$

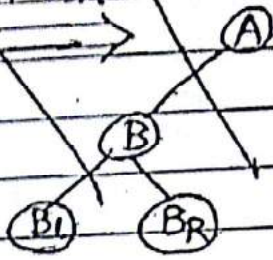
EXCELLENT

## RR problem

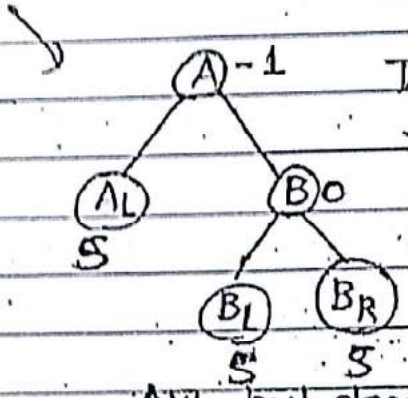



• No problem but close to problem

Insert-AR




• No problem but close to problem.

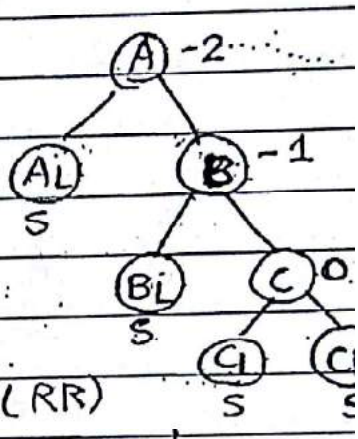



AVL but close to RR

(to get RR)

Insert in BR

O(1) + O(logn) + O(1)




(RR)

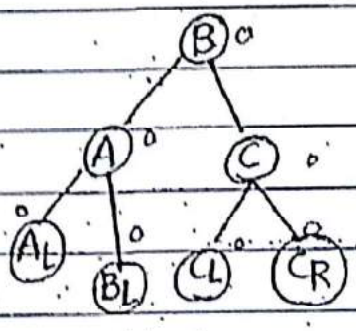
O(logn)

Check for problem

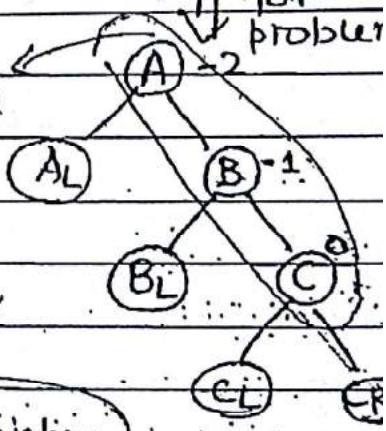
(We came to know about RR by - sign)

RR problem

Left Rotate




(AVL




BL become victim

Time complexity = O(1) + O(1) + O(logn) + O(logn)

= ? logn

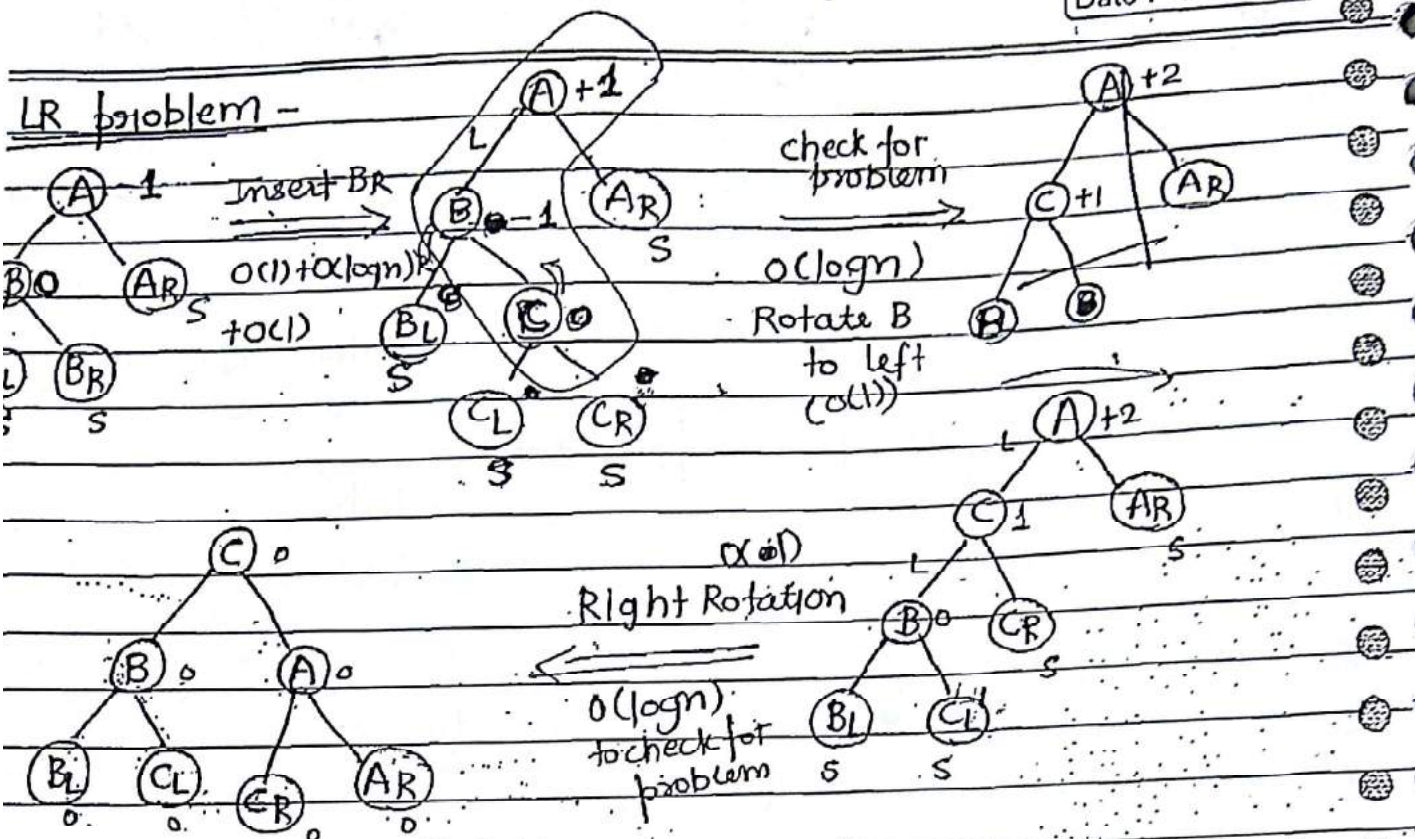
= O(log n)

To attach BL, check its behaviour 'Less than B more than A' do it in same way

to check for problem same path has to be taken to root from which path you reac to that node)

Page No.
Date : / / 

## LR problem -



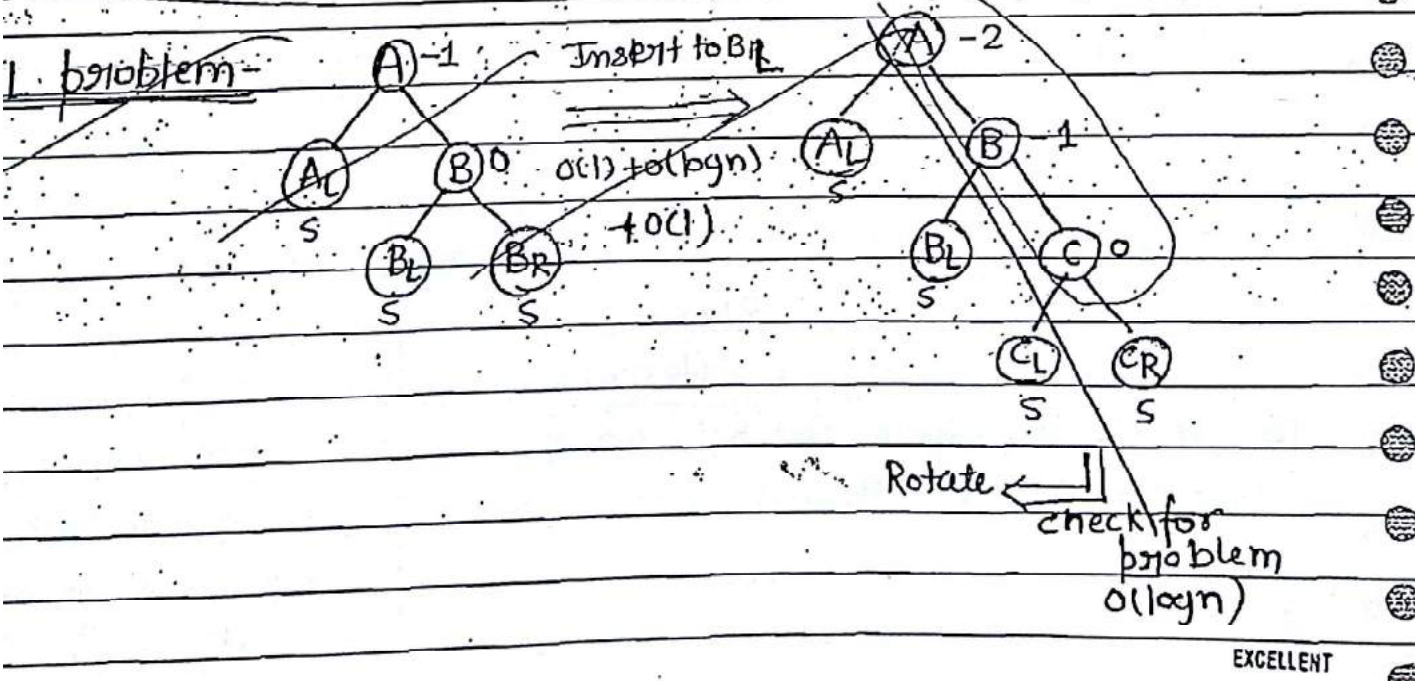

Total time = O(logn) + (log n) + log n

left Rotate — Right rotation

$= 3 \log n$

$= O(\log n)$

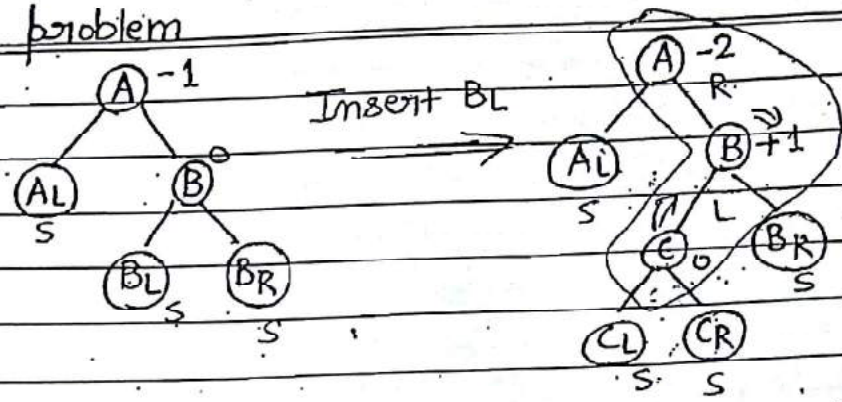
## I problem -

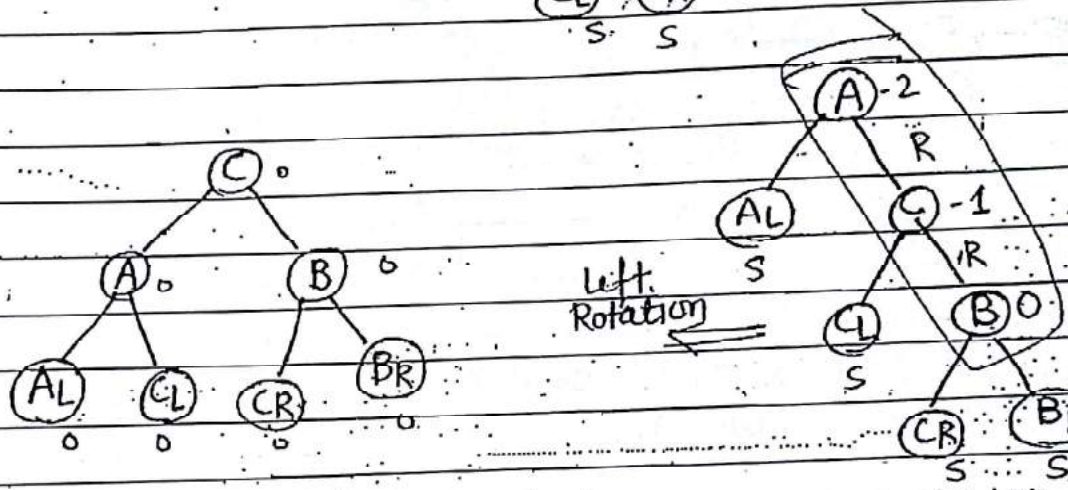



EXCELLENT

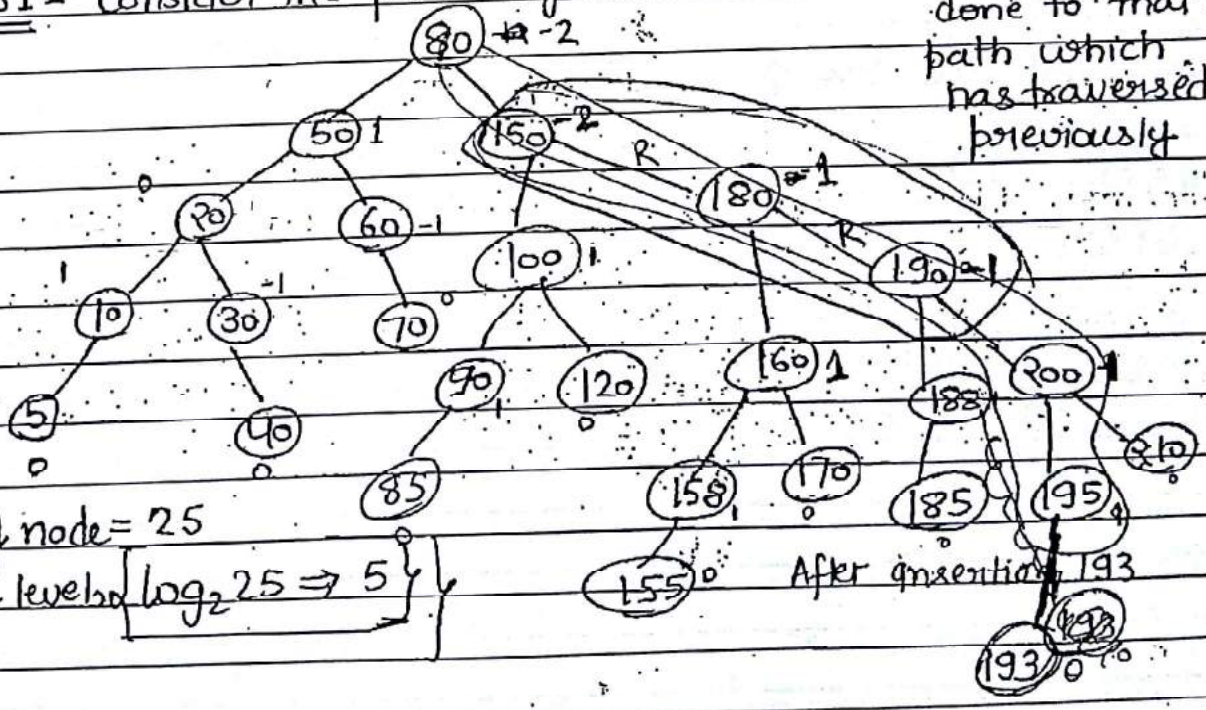
Page No.
Date : / /

RL problem




Ques 1 - Consider the following AVL tree-




check for AUL done to that path which has traversed previously

no of node = 25

{no of levels} $\log_2 25 \Rightarrow 5$

After insertion 193

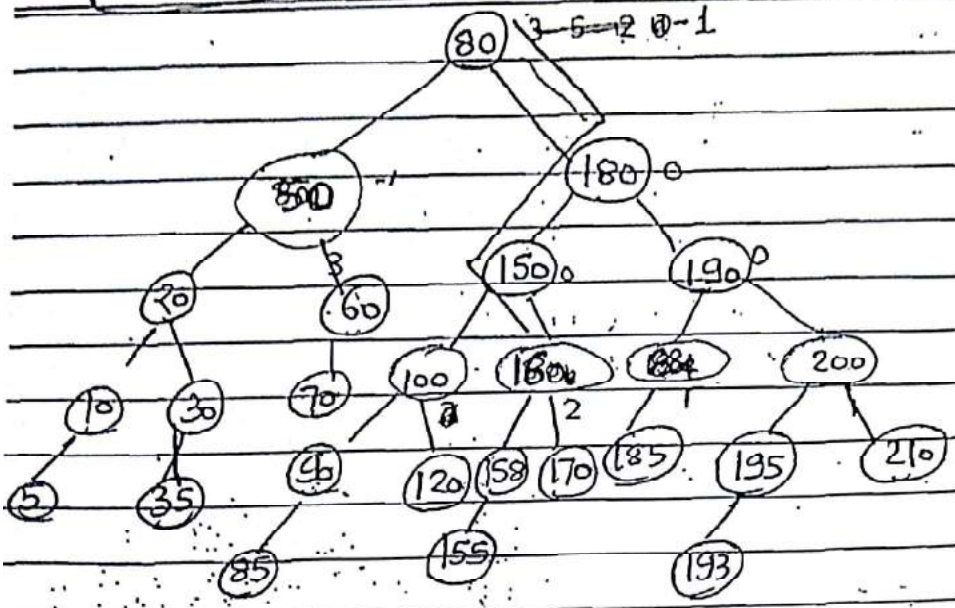


EXCELLENT

$O(1) + O(\log n) + O(1) + O(\log n) + O(1)$

creation ⇓ find ⇓ link ⇓ check ⇓ Rotate.

Page No.
Date : / /




Note – Insertion an element into AVL tree will take $O(\log n)$ time (BC, WC, AC).

n elements to create AVL tree = $O(n \log n)$

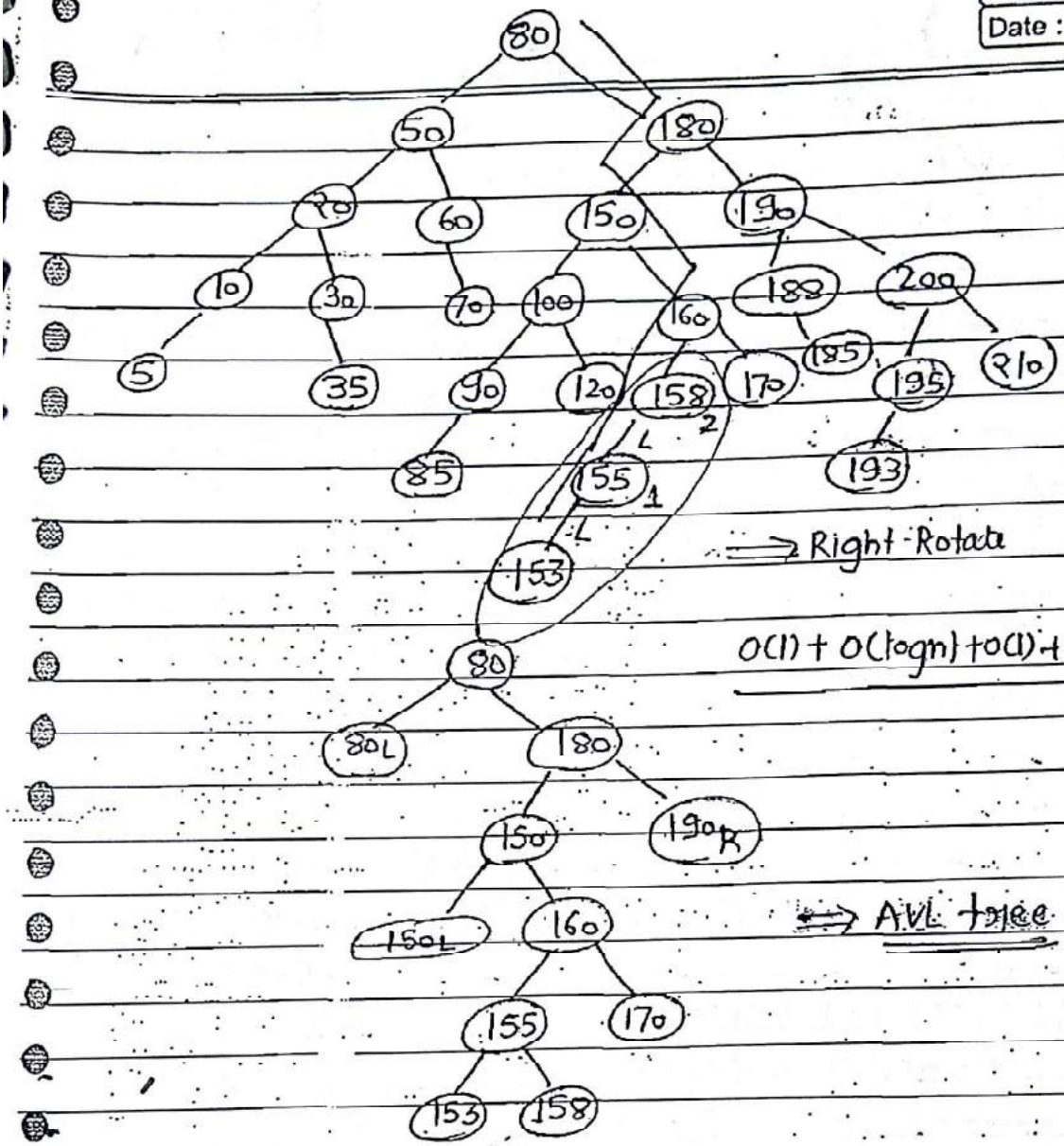
reating AVL tree for n elements = $O(n \log n)$

e only diff b/w AVL tree & BST is the balanced structure n AVL

~~ies – Consider the following~~

Insert element (153)

EXCELLENT




⟹ Right Rotate

O(1) + O(logn) + O(1) + O(logn) + O(1)

⟹ AVL tree

**Insertion-**

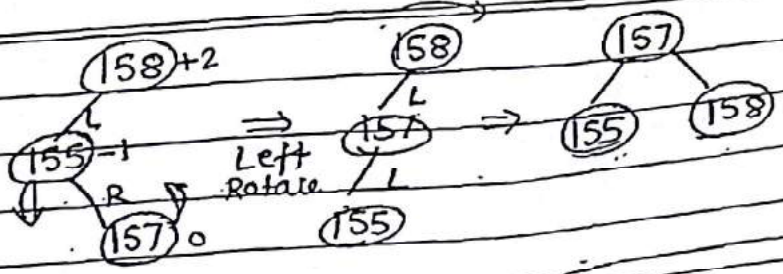
1. Create a node ⟹ O(1)
2. find its place ⟹ O(logn)
3. link ⟹ O(1)
4. check update tree. Is AVL or not, by again following the same path you traversed. ⟹ O(logn) (WC), O(1) (BC)
5. Rotate acc. to req. ⟹ O(1)



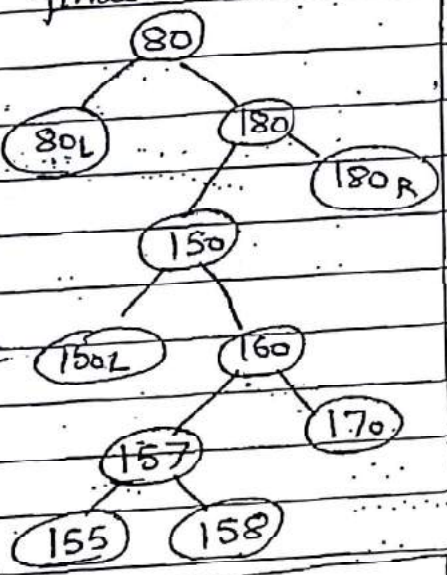
EXCELLENT

Page No.
Date : / / 

Insert 157




final tree




No of rotation = 2
No of problem = 1

- the person who are part of rotation will change first & then there children will change

// just rectify one problem ie 1st problem in your path. No need to check after it.

Ref. comp - O(logn) in Best case
O(2logn) in worst case

EXCELLENT

Dynamic Sets- the set change over time as when manipulated by algorithm.

Dictionary- a dynamic set that supports insertion, deletion, searching of elmt.

Each element in dictionary or a dynamic set is represented by an object whose attribute can be examined & manipulated if we have a pointer to object.

Satellite Data- any other attribute

Operation- • Search
Insert
Delete
Minimum
Maximum
Successor
Predecessor



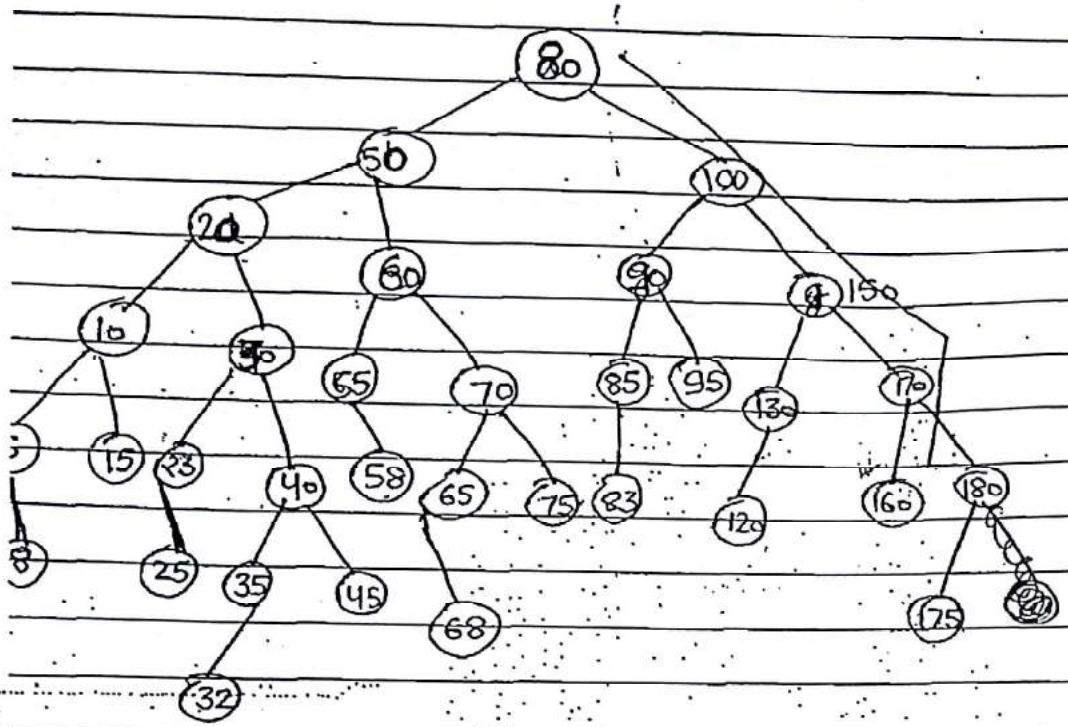
EXCELLENT

2-Apr

Parents of node performing rotation remain same

Page No.
Date : / /

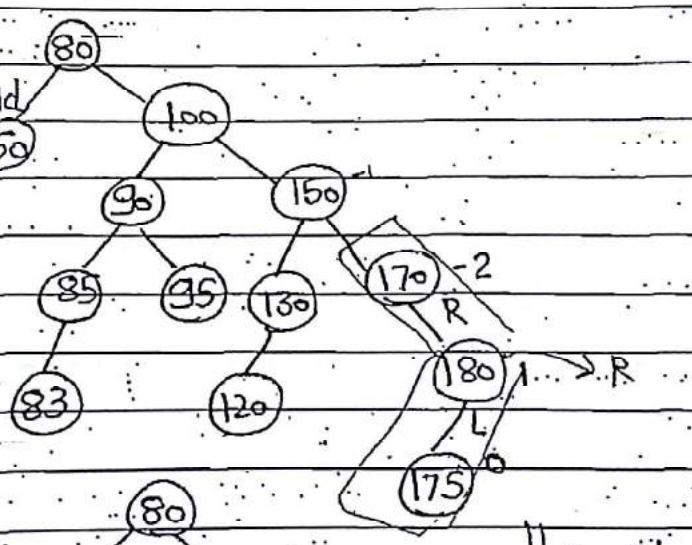
Ques.- Consider the following AVL tree-



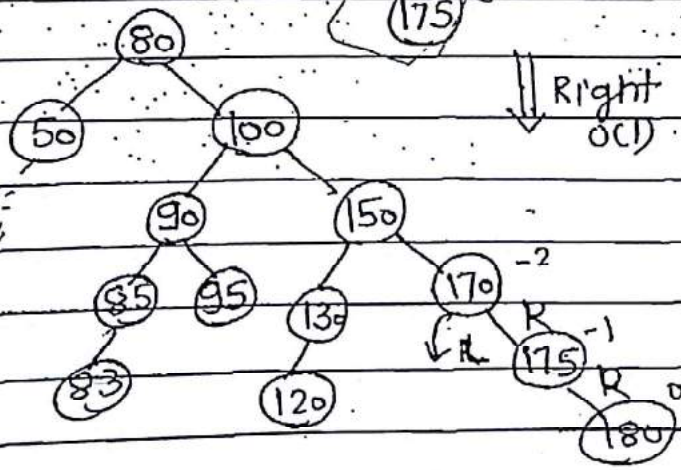

Delete 160

1. Find 160
2. Check for child
3. Delete

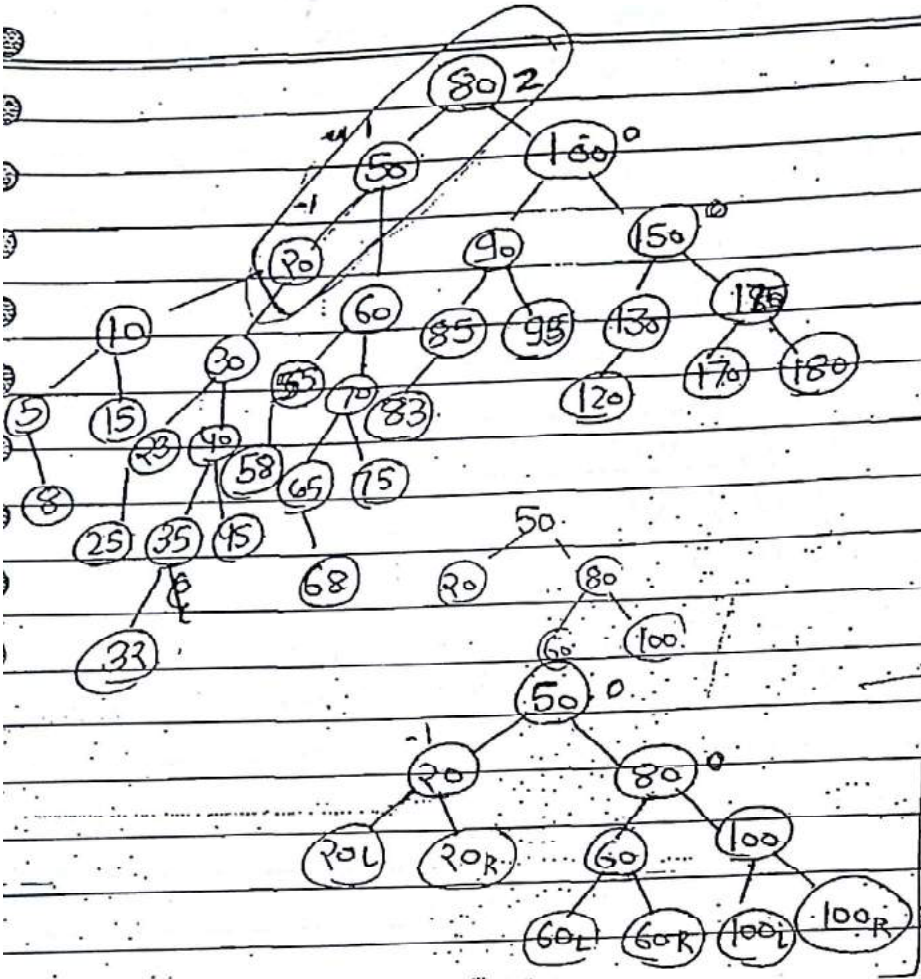
4. check for AVL from the parent of deleted node
5. check for problem type
6. Do Rotation.
7. Again check for AVL for the same path which you traversed while finding the node upto the root




Right O(1)




EXCEL




To find predecessor of a given node (O(~~log n~~) m) to find that node + O( O(log n - m) to find predecessor

↓ Total time O(log n) (WC) BC (O(1))

+2 - left

-2 - Right

- Hence, it is a AVL tree

Time Complexity - ~~O(log n) + O(1) + O(log p) + O(1) + O(log m)~~

(p → when every node is in problem, p+m = in entire path)

O(log n) + O(1) + O(log n) + O(log n)

↓ to find, ↑ Delete, ↓ check for AVL, ↑ Rotation

No of Rotation = 3 [LR (1) RR (2)]

↓ Left Rotation, ⇓ Right Rotation

(When you insert a node, height will increase or may not increase) when you height decrease or may not ~~increase~~ decrease

At time of insertion, you have to check for AVL one time only but in delete you have to check for the whole tree



EXCELLENT

find ↑ delete ↑ AVL ↑ Rotation ↑

$T(n) = O(\log n) + O(1) + O(\log n) +$

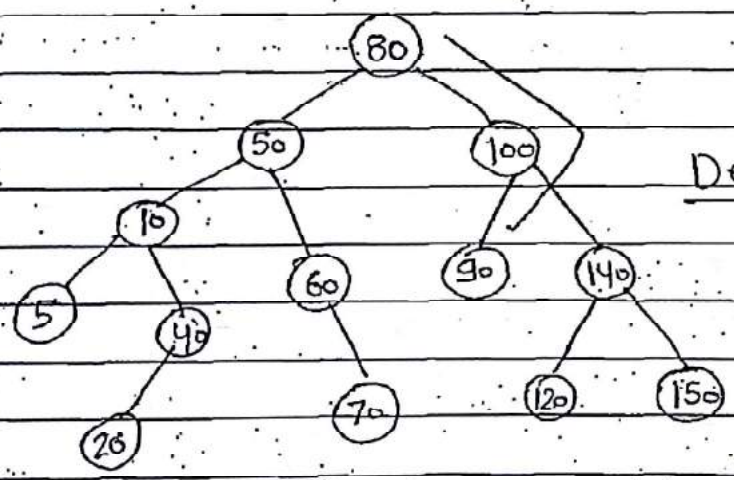
$= O(2\log n)$ (WC, BC, AC) → (to find predecessor)

$= O(1) + O(1) + O(1) + O(\log n) \Rightarrow O(\log n)$

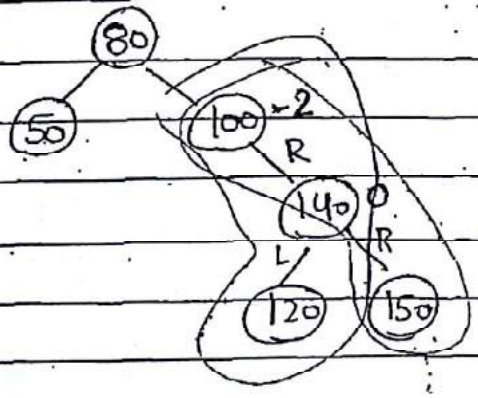
Note - { In AVL tree, after insertion, in that path, max. one problem can happen
Max Rotations are 2.
$O(2\log n)$ (WC)
$O(\log n)$ (BC) }

In AVL tree, after deletion, in that path while backtracking max ($\log n$) can happen.
Max. Rotation - ($2\log n$) Rotations (if each problem is LR)

Ques - Consider the following AVL tree -



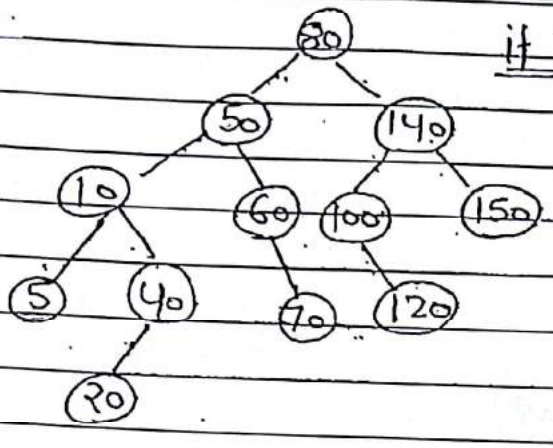

Delete 90




RL or RR
(You can choose anyone of them)
Acc. to your req.

EXCELLENT

AVL tree can not be balanced at once
You can not do it by using ~~Max heap~~



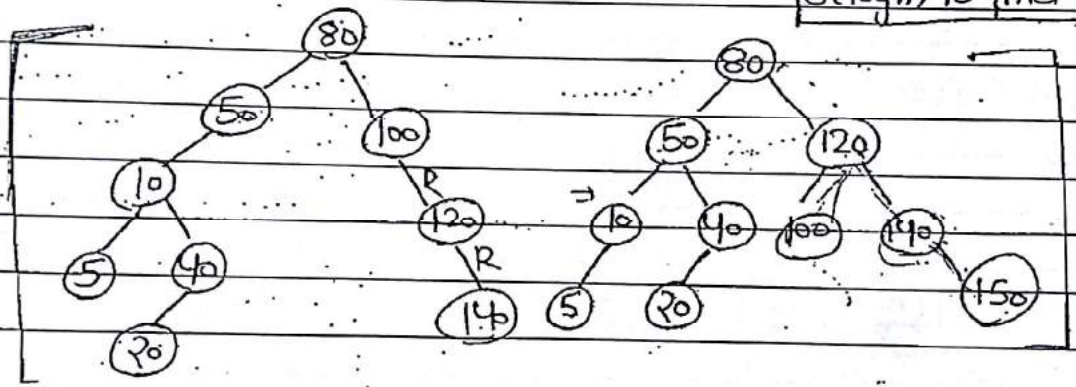

if I choose RR (because we want min. rotation

~~You cannot RL to be~~ choose
~~solved bcoz RL~~ convert to RR & Here itself is RR. ~~You cannot~~ make it RR.

Min. no of rotations required = 1 (Left Rotation)

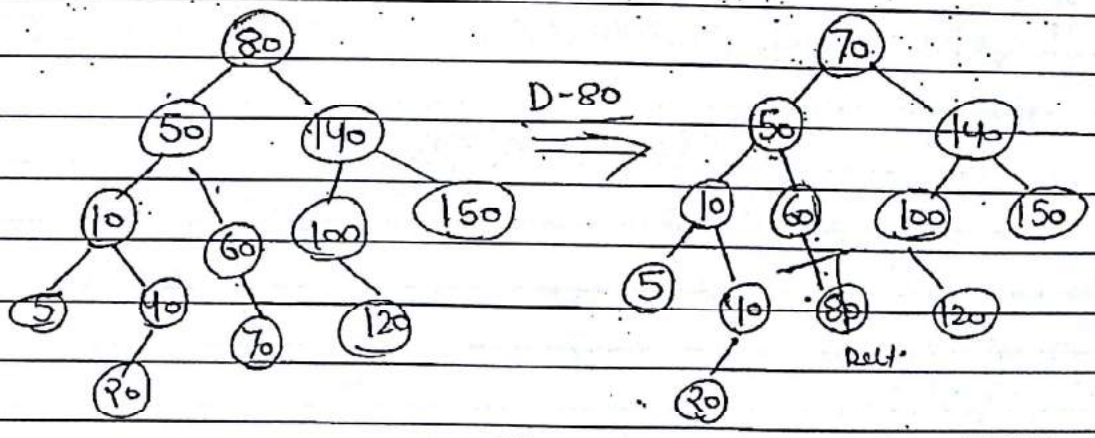
O(1) → to find at BC
O(1) → to delete
~~O(h) to find pt~~ O(logn) → to check AVL
O(logn) to find predeccessor

if I choose RL




- Due to Rotation, height may be decreased or may remain same but not ~~rem~~ will never increase

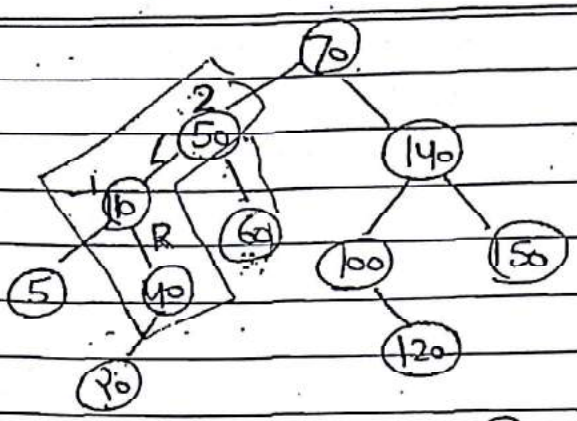
Ques -

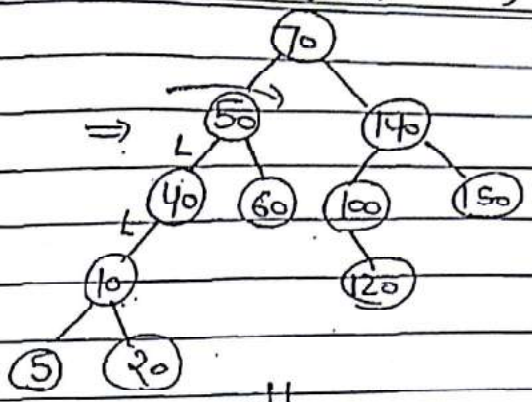




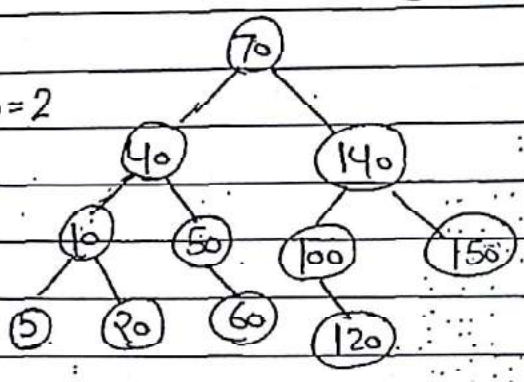

EXCELLENT

In Deletion, check for AVL = O(logn), (Wc, Bc, Ac)

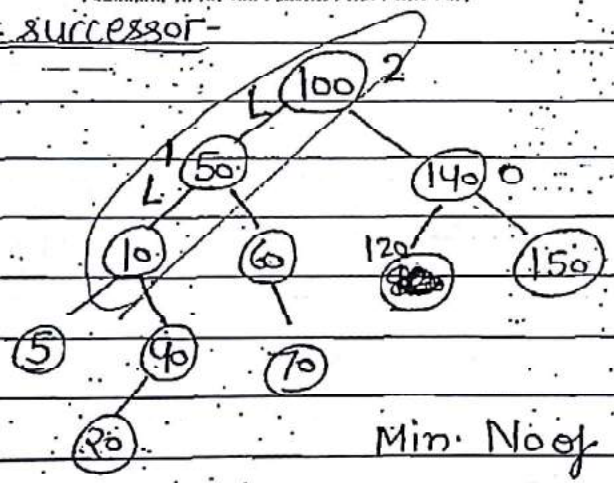


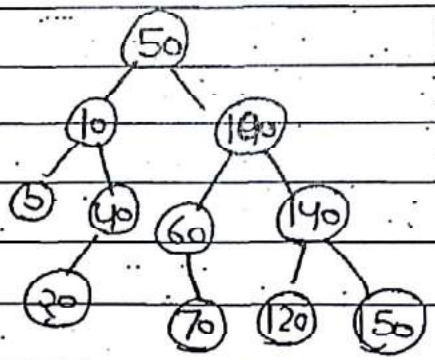




Min. No of Rotation = 2




Replace by successor-




Min. No of Rotation = 1

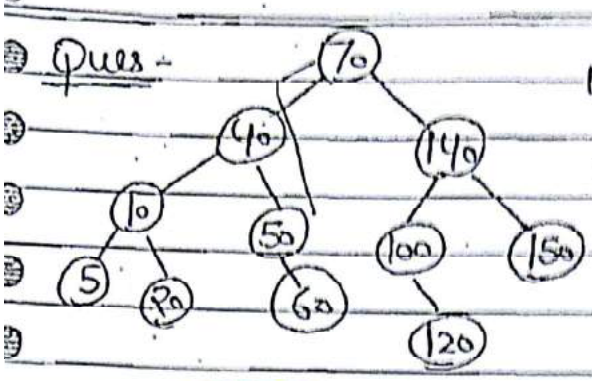
You can Replace by predeccessor or successor, if not mentioned it you wanna get minimum rotation, check for both

O(logn) + O(logn)

= O(logn)

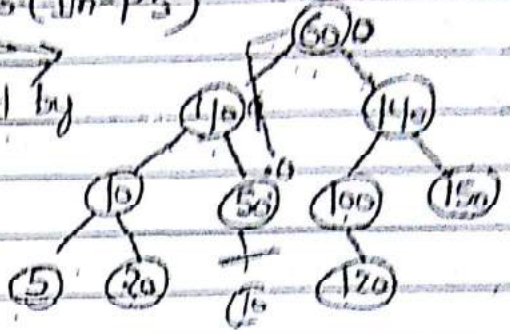
EXCELLENT

Ques -



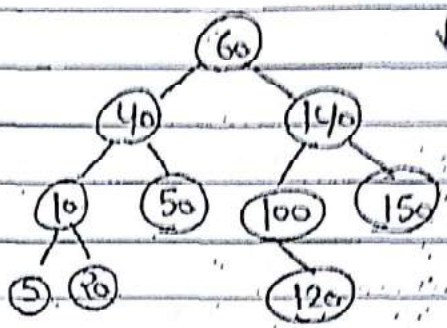

Delete 70 (In-PS)

replaced by pre






⇓ No problem




- When Deletion is done, problem may or may not occur.

Ques - What is the max. height of an AVL tree with 20 nodes?

Max. height of BST with 20 nodes? - 19

Let 20 levels: 0 - ⊙ → 0

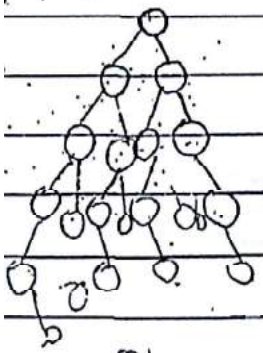
1 - O → 0

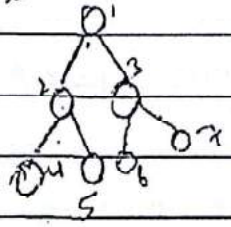
2 - O → 1

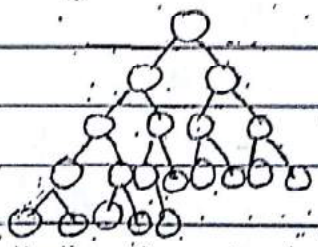
3. → 1

4. → 2

5. → 2



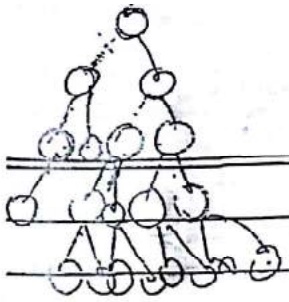




Max height → n-1

Min height = $\lfloor \log n \rfloor$

Min height = $\lceil \log n \rceil$



Page No.
Date : / /

min. no of node

h = 0 (1 node)

h = 1 (2 node)

h = 2 (4 node) (2+2)

h = 3 (7 node)
(4+1+1)

h(3) is obtained from attaching h(1) tree to h(2) tree

h = 4 (12 node)

h = 5

h = 4
12

h = 3
7 node

12+7+1 = 20

(20 node)

$h(n) = h(n-1) + h(n-2) + 1$

$h(6) = h(5) + h(4) + 1 = 33$

min no of nodes (H) = min no of node (H-1) + min no of node (H-2) + 1

↓
Height of tree

EXCELLENT

Page No.
Date :

min no of node in Height h AVL tree,

$$MNN(H) = \begin{cases} 1 ; & \text{if } H=0 \\ 2 ; & \text{if } H=1 \\ MNN(H-1)+MNN(H-2)+1 ; & \text{if } H \geq 2 \end{cases}$$

~~$T(n) = T(n-1) + T(n-2) + 1$~~

∴ Height of AVL tree with n nodes = $\log n$

Max. height of AVL tree = $\lceil \log n \rceil$ $n=100$ $\log n = 6.$ $\approx 7$

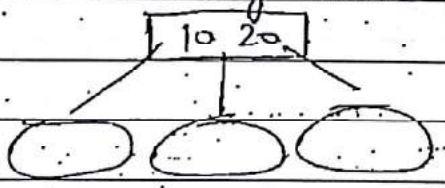
Min height of AVL tree, with n nodes = $\lfloor \log_2(n+1) - 1 \rfloor$

as $n = 2^k - 1$

no of level, $k = \log(n+1)$

height = $\lfloor \log_2(n+1) - 1 \rfloor$

B tree- if a particular node has more than one data field i.e it may have more than two children




multiway search tree.

Height → $\log_3 n$ as it is three pointer field.

- B tree has lesser height than AVL tree.

if it is two data field, then it will store 3 pointers.



EXCELLENT

Date:

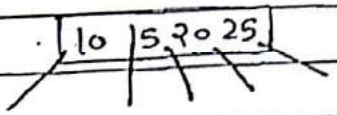
rder of a tree- no of children it can have.

rder of a Binary tree- no of children = 2

order = 5 No of children = 5

data field = order - 1

= 4.

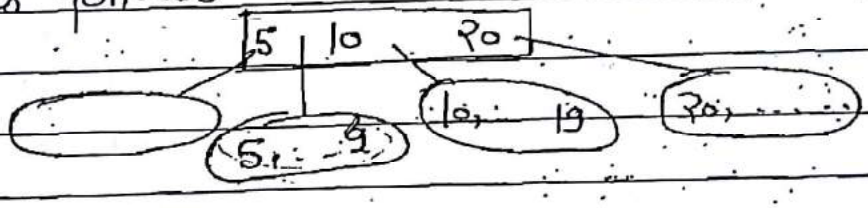



VL, B-Tree both are search tree as well as they both re balanced.

No of level in B-tree = $\lceil \log_{order} n \rceil$ (BC, WC, AC)

if order = 5 = $\log_5 n$ (No of level)

Tree- right pointers of a node will also point to No. itself as follows




dvantage- All the Nodes will be at leaf level itself, you can perform linear search.

It is also a type of B-tree.

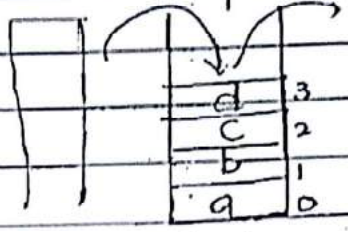
EXCELLENT

exit() - Exit out of program

Date : / /

# Stack

Defination- Oneside open, another side is closed




To Insertion- Top = Top+1

To delete Top = Top-1

: LIFO, FILO (Last In First Out, First In Last Out)

ADT of Stack - push()
pop()

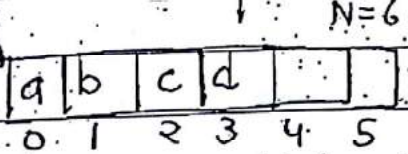
ADT of a particular Data Structure- the opn which are possible on a particular DS, but you donot bother for the implementation.

Implementing Stack- implementation is done using a particular data structure

Implementation of Stack using array-

queue -

Using Array-




initially, int top = -1

```
void push(char x)
{
    if(Top == N+1)
    {   printf("Stack overflow");
        exit(1);
    }
```

```
else
{
    Top++;                  (increment & then store)
    s[Top] = x;        |    s[++Top] = x      |    if s[Top++] = x X
}
}
```

```
char POP( )                 // returning the element which get deleted
{   char y;
    if (Top == -1)
    {
        printf("Stack Underflow");
        exit(1);
    }
    else
    {   y = s[Top];    (store & then decrement)  |  y = s[Top--]
        Top--;                                  |  y = s[--Top] X
    }
    return y;
}
```

Time - O(1) (Push, pop) (BC, WC, AC)

## Implementing Multiple stacks in Single Array-

- A single array will store multiple stacks for each stack in that array, there will be top variable

Let array size, N = 9

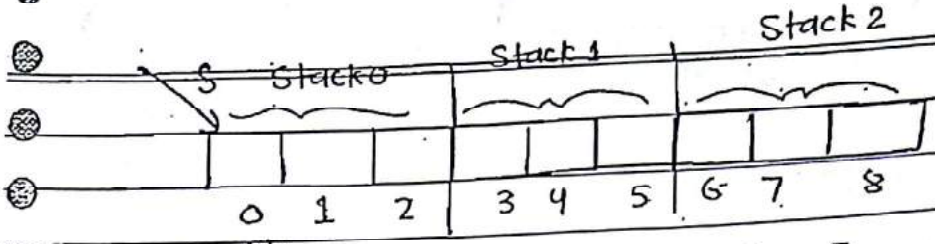
No of stack = m = 3

∴ Stack size = $\frac{N}{m} = 3$

If N = 8 stack size ?

∴ 3, 3, 2

EXCELLENT




top, $T_0 = -1$ (initially)  $T_1 = 2$ (initially)  $T_2 = 5$ (initially)

To calculate initially top of ~~array~~ of any stack.

Initially for the whole array, it is -1.

∴ for 0th stack, Initial Top of Stack, $0 \times 3 - 1 = -1$

(0 → no of stack before that stack; 3 → no of elmt in that stack; * Base address (i.e the initial top of stack for the whole array)

for 1th stack, Initial Top of Stack = $1 \times 3 - 1 = 2$

for 2nd stack, Initial Top of Stack = $2 \times 3 - 1 = 5$

for ith stack, Initial Top of Stack = $\left(i * \frac{N}{m} - 1\right)$

```
Void push(Ti, x)  (insertion to stack i)
{
   if (Ti == ((i+1)*N/m - 1))
   { by("Stack overflow");
     exit(1);
   }
   else{
        Ti++;
        s[Ti] = x;
```



EXCELLENT

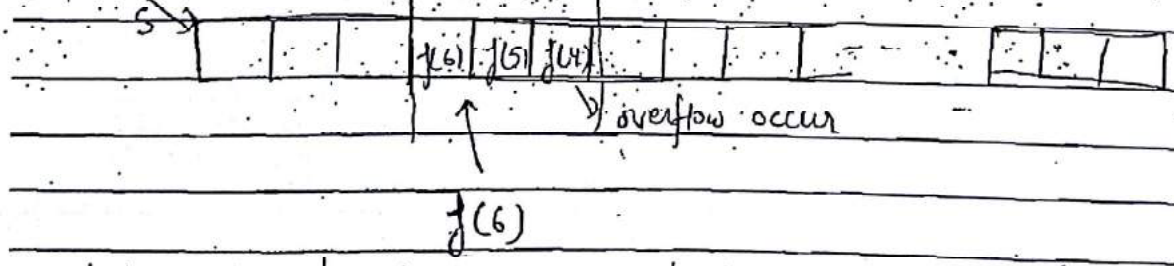
```
char pop(Ti)  (deletion from stack i)
{
    if (Ti == (i*N)/M - 1)
    {
        pf("Stack Underflow");
        exit(1);
    }
    else
    {
        y = s[Ti];
        Ti--;
    }
    return y;
}
```

TC (push pop) = O(1)

Advantage of Multiple Stack in a single array-

- If there is a single stack only, in a single array, you can not run two recursive program at a time.
  By using MSSA concept, you can run two recursive program at a time

Draw Back:-



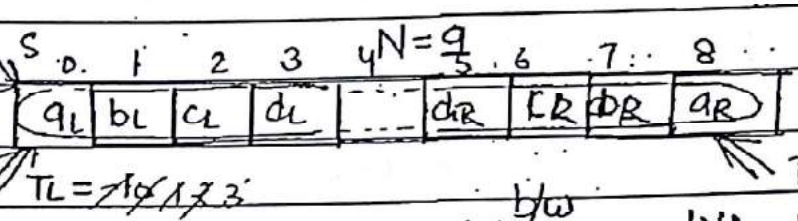

Using above program we can implement multiple stacks in a single array but it is not effident as lot of space is empty still error msg came if one stack is full & other remaining

EXCELLENT

are empty we are getting error msg, stack overflow even though lot of space available.

Implementing Multiple Stacks in single array efficiently :-

ADT- left push
left pop
Right push, Right pop




Here Memory is shared b/w two diff DS, which lead to the synchronization problem & that M/M Must be considered as critical section which must be handled by semaphore.

An array is full $T_L = T_R - 1$ or $T_R = T_L + 1$ (same for both stack)

Stack empty - if $T_L = -1$ if $T_R = N$

In this strategy, one stack will start from left & other will start from right. left will have top = -1 (initially) & Right will have top = N (initially)

For insertion, right will decrement Top
left will increment the Top

if somewhere; $T_L = T_R$ (when overwritting of data occured)

If given, $T_L = T_R = \frac{N}{2}$, it may or may not possible

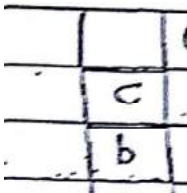
As it might be possible that no of insertion in $T_L$ is less than $T_R$

if given $T_L == N$, it may or may not be possible
$T_R == 0$ MAY or may not possible

If you want to implemented multiple stack efficiently, you can only implement two stacks at a time.

Stack life time of an element-

Ex 1:- n=3 (3 elmt pushed continously followed by 3 pops)




Let us assume a push or pop take 5 min. time.

Elapsed time- y=3 time wasted b/w two ~~push~~ operation

~~3 (time elapsed b/w push op^n of e & pop op^n of e)~~

| | | | |
|---|---|---|---|
| | | | 3 |
| pop(c) | 5 | c | 5 (push(c) |
| | 3 | | 3 |
| pop(b) | 5 | b | 5 (push b) |
| | 3 | | 3 |
| pop(a) | 5 | a | 5 (push a) |

Life time of elmt in stack is time after push op^n of that element before pop op^n of that element.

Life tym of c = 3
Life tym of b = 19 (3, 5, 3, 5, 3)
Life tym of a = 35 (3,5,3,5,3,5,3,5,3)

Life tym of an elmt is the smallest then that element is the last elmt of the stack.

Avg: life time of elmt in stack = $\frac{3+19+35}{3}$

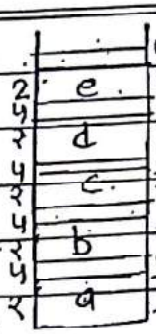
$= (19)$

life tym of an elmt is basically ~~time~~ time for which elmt above that is present in stack.

EXCELLENT

Page No.
Date : / /

e2- $n = 5$ (a, b, c, d, e)
$x = 2$ (push & pop time)
$y = 4$ (elapsed time)




L.T of e = 4
LT. of d = time taken to push e & pope
& then elapsed time to pop d
= ~~4+2+2~~ 4, 2, 4, 2, 4 = 16

LT. of c = (4,2) (4,2) (4,2) (4,2), 4 = ~~38~~ ~~28~~ 28
d e e d

LT. of b = (4,2) (4,2) (4,2) (4,2) (4,2) (4,2), 4 = ~~48~~ 40
c d e e d c

LT of a = (4,2) (4,2) (4,2) (4,2) (4,2) (4,2) (4,2) (4,2), 4
b c d e e d dc b = 52

Total time = 136+4

Avg. time = ~~26+12~~ ~~27+2~~ ⇒ ~~28~~ $\frac{n(x+y)-x}{30-2}$

$= \frac{140}{5} = 28$

for a = $4*2(x+y) + y$
↑ for push & pop
for 4 element

b = $3 \times 2(x+y) + y$
c = $2 \times 2(x+y) + y$
d = $1 \times 2(x+y) + y$
e = $y + 0 \times 2(x+y)$

If n no

$(n-1) \times 2(x+y) + y$

Sum,

$= 2*(x+y)[0+1+2+\cdots n-1] + ny$

$= 2*(x+y)\frac{n(n-1)}{2} + ny$

$= n(x+y)(n-1) + ny$

$= n[(x+y)(n-1)+y]$

Avg = $(x+y)(n-1)+y$

$= n(x+y) - x$

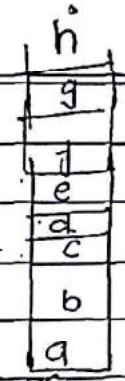
EXCELLENT

88

Page No.
Date : / /

## Implementing queue using stack-

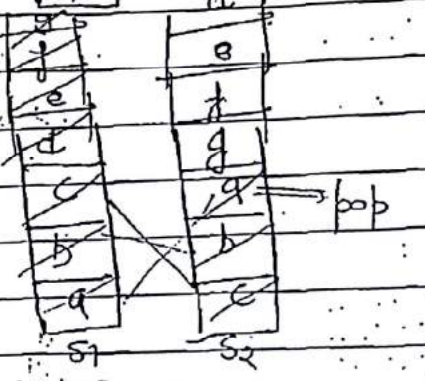
Enqueue- 


1 Enqueue = 1 push opn in $S_1$

2 Enqueue = 2 push opn in $S_1$

Dequeue- 1 Dequeue- 3 pop opn ($S_1$)
+3 push opn ($S_2$)
+1 pop opn ($S_1$)




pop(a)
push ($S_2$)
pop ($S_1$)
push ($S_2$)
pop ($S_1$)
push ($S_2$)
pop ($S_2$)

Time taken by Enqueue - O(1).

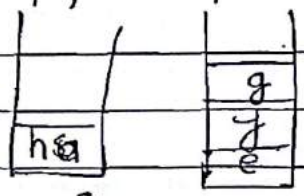
Taken taken by Dequeue (first d-Q) = O(n)
(2n+1)

but II element when Deleted = O(1) as $S_2$ has elmt.

3- EQ = 3 push $S_1$
7- DQ = 3 pop $S_1$ +
3 push $S_2$ +
1 pop $S_2$

1- DQ = 1 pop $S_2$
1- DQ = 1- pop $S_2$

then again
4- EN = 4 push
1- DQ = 4 pop $S_1$ + 4 push $S_2$ + 1 pop $S_2$.

3- EQ 


1- DQ = 1 pop - $S_2$
= (e)

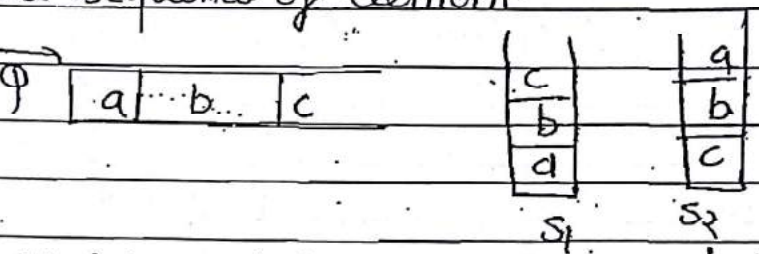
EXCELLENT

Page No.
Date : / /

Observation-

if you want to Enqueue the elmt. directly visit $S_1$ & insert the elmt.

if you want to dequeue, directly visit $S_2$ if $S_2$ is empty goto $S_1$ perform pop on $S_1$ & push it into $S_2$ $S_2$ will have original sequence of element




it will take 3 pop $S_1$ + 3 push $S_2$ + 1 pop $S_2$.

but if $S_2$ is not empty directly delete from $S_2$ lead to O(1) time-

∴ 1-ENQUEUE = 1 push

1-DEQUEUE → 1 pop (if $S_2$ is not empty)

→ n push() + n pop + 1 pop (if $S_2$ is empty)
($S_2$ $S_1$)

- Insertion is done in $S_1$.



EXCELLENT

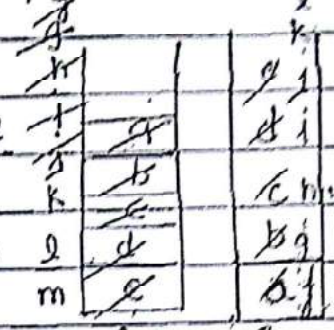
Page No.
Date: / /

Ex 2

5EQ
3DQ
8ENQ
7DQ
15ENQ
14DQ

how many push & pop using queue by stack.

| a | b | c | d | e |




5-EQ = 5 push-S1
3-DQ = 5 pop-S1
+5 push-S2
+1-pop-S2
1-DQ = 1-pop-S2
1-DQ = 1-pop-S2
8 ENQ = 8 push-S1
1-DQ = 1 pop-S2
1-DQ = 1-pop-S2
1-DQ = 8 pop-S1
+8 pop-push-S2
+1 pop-S2
64-DQ = 4 pop-S2
15-EQ = 15 push
1-DQ = 1-pop-S2
1-DQ = 1-pop-S2
1-DQ = 1-pop-S2
1-DQ = 15-pop-S1
+15 push-S2
+1-pop S2
10-DQ = 10 pop-S2

Total push - 56
pop - 52

Program ENQUEUE-

```
void ENQUEUE(S1, x)
{
    push(S1, x);
}
char DEQUEUE(S1, S2, x)
{
    if(S2 = empty)
    { x = pop
        while(S1 is not empty)
        {
            x = pop(S1);
            push(S2, x);
        }
    } pop(S2);
```

TC = O(1)

EXCELLENT

To Implement Queue, No of stack = 1

Page No.
Date: / /

```
else
{
    y = pop(S1);
}
return y;
}
```

TC = O(1) = {Avg. Case, Best case}
Worst case = O(n)

HomeWork- Implement Stack using Queue. {push = O(1), pop(O(n))}

push = 3 EQ - Q1
1 - Pop = 2 DQ - Q1
2 - EQ = Q2
1 - DQ (Q1)
3 push = 3 EQ - Q2
to check it push the elmt where queue is not ~~empty~~ empty -
pop = O(n) (BC, AC, WC)

Applications of Stack-

1. Recursion
2. (i) Tail Recursion
   (ii) Non Tail Recursion
   (iii) Indirect Recursion
   (iv) Nested Recursion
3. Infix to postfix
3. prefix to postfix
4. postfix evaluation
5. Tower of Hanoi
6. Fibonacci Series

1. Recursion- WAP in C using Recursion to print the array of n elements

```
print( ~~a[]~~ int a[], int n)
{
    ~~if~~ static int i=0;
    if(i<n)
        printf(a[i])
    i++;
    printf(a[], n);
}
else
    ~~exit(0)~~ or return. 90
```

EXCELLENT

Page No.
Date: / /

```
print(a, i, j)
{
  if (i == j)
  { printf(a[i])
    return;
  }
  else
  { printf(a[i])
    print(a, i+1, j);
  }
}
```

Tail Recursion- When a $f^n$ call poped out i.e. a $f^n$ come back after its completion and there is no work to do, known as Tail Recursion.
i.e when last stmt of a $f^n$ is itself a Recursion. at this stage is not required actually.

to 20 ~~Recursion~~

```
PA(a, i, j)
{
  if
  2: to
  P(a,i,j)
  3:
  {
  }
}
```

Here no work to do but still it wait in stack known as tail recursion.
Here, in actual, no stack is required but it uses so known as tail recursion.

Non-Tail Recursion-, When $f^n$ call is ~~the~~ Not the last stmt of $f^n$ i.e some ~~function in call~~ stmt in calling $f^n$ depends upon called $f^n$.

Recursion = { $T(n) = T(n-1) + c$ } $= O(n)$

10-Apr

If we have
PA( ) → Non-tail Recursion
PA( ) → Tail Recursion

Page No. combining
Date Non-tail Recursion

Ex for Tail Recursion - above program.

(2) The disadvantage with tail recursion is lot of stack space is wasting.

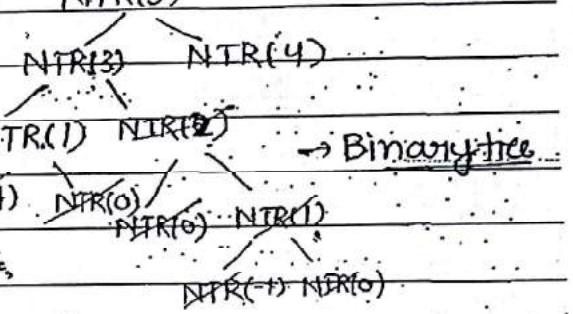
(3) The advantage with the tail recursive program is we can write equivalent non recursive program easily with the help of loops.

Non-Tail Recursion-

```
NTR(n)
{
  if(n<=0) return;
  else
  {
    NTR(n-2);
    pf(n);
    NTR(n-1);
  }
}
```

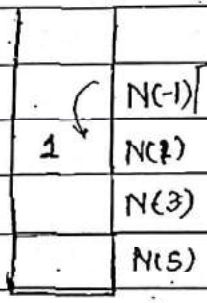
O/P - ~~1,3,2,1,2,1,1,3,2~~
1,3,2,1,5,2,1,~~3~~,1,3,~~2,1~~

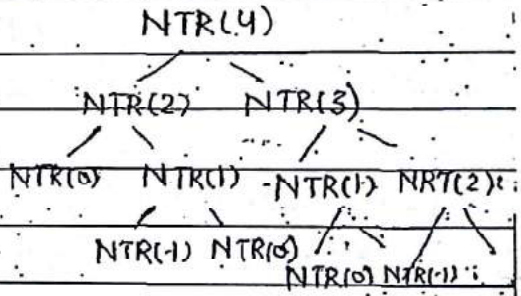
I/P = 5




→ Binary tree

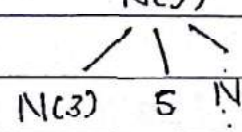
~~1,3,5~~




Calling sequence - preorder.

[O/P of sequence - postorder (as parent is getting completed only when child are completed)]




→ Binary tree-
as 5 is only a printf taking constant tym.

$TC = O(2^n)$

$TC = T(n-1) + T(n-2) + O(1)$

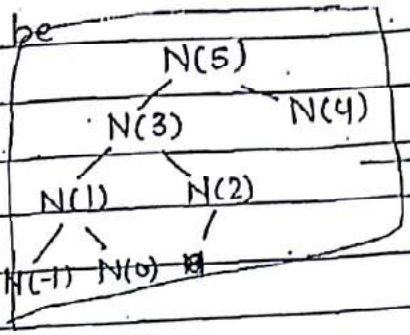
EXCELLENT

stack space = O(n)

- By using DP, TC = O(n+1) ≈ O(n) (as n+1 distinct Dpm calls)

Page No.

New tree will be




→ O(n) stack size = O(n)

stack table space = O(n+1)

If ask- simple, TC = $O(2^n)$

no repition of $f^n$ call = O(n) (using DP)

```
NR( )
{
push(N(5)
push(N(3)
,
}
```

| | TC | SC |
|---|---|---|
| | | |
| With DP | n | n+n = 2n |
| Without DP | $2^n$ | n |

- Pascal do not support recursion as it does not have stack.

- for a recursive program, if you want to write equivalent Non-Recursive program the stack is required as will be doing it by push & pop.

- for a recursive program, if a stack is not required for non-recursive program, then that must be a tail recursion.

1. IN the Given recursive program, After the function call, there is something to do then it is called Non-tail recursion.
2. It is very difficult to write equivalent Non recursive program for given non-tail recursive program.
3. We are not wasting stack space unneccessarily in Non-Tail Recursive program.
4. For the given non tail recursive program, if you write non recursive program then you have to take stack

EXCELLENT

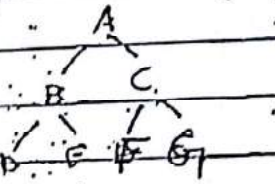
Page No.
Date: / /

5. For the given tail recursive program, if you write Non-Recursive program, then stack not required.

```
Preorder(x)
{ x
  po(x->lc)
  po(x->rc)
}
```

if you wanna write its equivalent preorder

```
(x) {  pf(x)
       push(A); (because after going to left, when you came back you required A->right)
       x = x->left
       pf(x)
       push(B)
       x = x->left
       push(D)
       x = x->left
       push(E)
       pop(D)
       push(E)
       pop(E)
       pop(B)
       x -> A->right
       pf(x)
}
```




Whenever I write NR of NTR then you have to take care everything taken care by system & stack is always required.

for QuickSort, when worst partitioning occur

@ ( ) ( )
1 n-1

QS(B)

QS(B)

there is no need to come back to sort a single element: so no need to come back (as it is a tail recursion ∴ it is said for better program it require logn stack space

You can write its equivalent non recursive program. no stack is required.



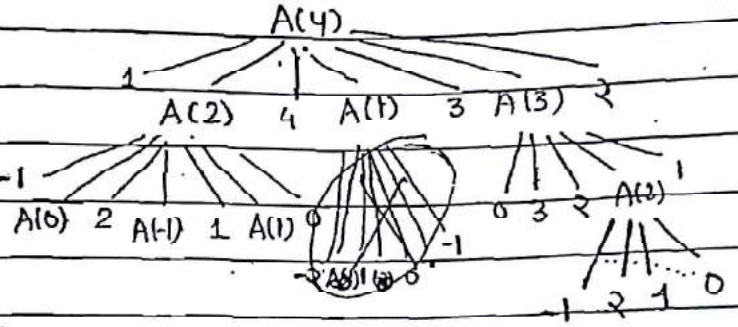
EXCELLENT

(n)

i/P = 4

o/P : ~~1, 1, 3~~

```
if (n<=1) return;
else
{
  bf (n-3);
  A(n-2);
  bf (n);
  A(n-3);
  bf (n-1);
  A(n-1);
  bf (n-2);
}
```




1 -1, 2, 1, 0, 4, -2, 0, -1, 3, 0, 3, 2, -1, 2, 1, 0, 1 2

Without using DP, TC = $3^n$ ... space = n ... = O(n)

with D.P. = O(n) space = n + n = O(n)

Indirect Recursion Ex:

```
A()          B()
{            {
==           ==
==           ==
B()          A()
==           ==
==           ==
}            }
```

A is calling B but B indirectly called A - Indirect Recursion.

EXCELLENT

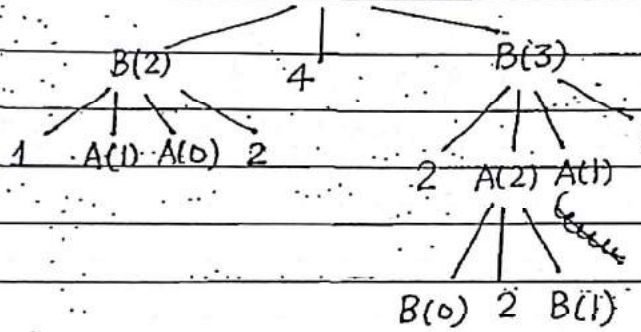
Page No.
Date : / /

Ex 2

```
A(n)
{
  if(n<=1) return;
  else
  {
    B(n-2);
    pf(n);
    B(n-1);
  }
}
```

```
B(n)
{
  if(n<=1) return;
  else
  {
    pf(n-1)
    A(n-1);
    A(n-2);
    pf(n);
  }
}
```

I/P: A(4)




Here Execution tree will have diff $f^n$ at different levels.

O/P: 1 2 4 2 2 3

No of Levels - n

$\therefore$ Complexity = $O(2^n)$ } without DP

Stack space = $O(n)$

with DP = n/2 for A & n/2 for B

$\therefore$ as if for A if 4 is there then for B 4 is not there

$= O(n)$

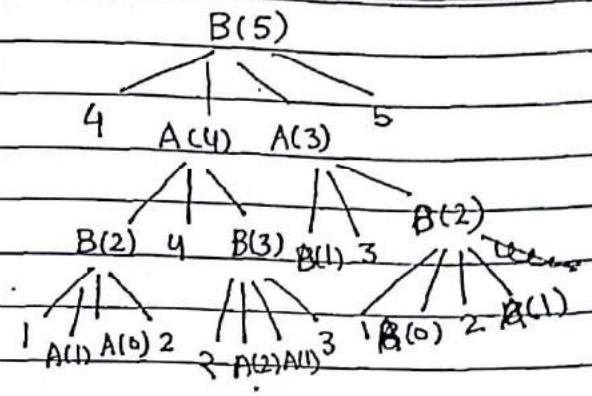
$\because$ n distinct $f^n$ calls

Space = $O(n+n) = 2n \approx O(n)$

If in Indirect Recursion, if A() has 2 fn call & B has 3 fn call then $tc = 3^n$ as upper bound will be taken to ternary (three).

Page No.
Date: / /

I/P B(5).




O/P - 4, 1, 2, 4, 2, 2, 3, 3, 1, 2, 5

Nested Recursion-

$$A(m,n) = \begin{cases} n+1 & ; \text{if } m=0 \\ A(m-1,1) & ; \text{if } n=0 \\ A(m-1, A(m,n-1)) & ; \text{otherwise} \end{cases}$$

When fn calling itself with calling itself in parameters as well as

```
A()
{
  ---
  A(A())
}
```

A(1, 5) → A(0, A(1,4)) = 1 + A(1,4)

= 1 + A(0, A(1,3))

= 1 + 1 + A(1,3)

= 1 + 1 + 1 + A(1,2)

= 1 + 1 + 1 + 1 + A(1,1)

= 1 + 1 + 1 + 1 + A(0, A(1,0))

= 1 + 1 + 1 + 1 + A(0, A(0,1))

O/P = 7

= 1 + 1 + 1 + 1 + ~~A(0,1)~~ + A(0,2)

= 5 1 + 1 + 1 + 1 + 3

= (7)

Page No.
Date : / /

With DP, $O(mn) \Rightarrow mn$ distinct fn call.

m n

Check once

$A(2,5) \Rightarrow A(1, A(2,4)) \Rightarrow A(1,13)$

$\Downarrow$

$A(1, A(2,3)) = A(1,11)$

$\Downarrow$

$A(1, A(2,2)) = A(1,9) = 11$

$\Downarrow$

$A(1, A(2,1)) \Rightarrow A(1,5) = A(0, A(1,4)) = A(0, A(1,3))$

$\Downarrow$ $\qquad$ $\Downarrow$ $\qquad$ $\Downarrow$

$A(1, A(2,0)) = A(1,3) \Rightarrow A(0, A(1,2))$

$\Downarrow$ $\qquad$ $\Downarrow$

$A(1,1)$ $\qquad$ $A(0, A(1,1)) = 4$

$\Downarrow$

$A(0, A(1,0)) = A(0,2) = 3$

$\Downarrow$

$A(0,1) = 2$

as $1,0 = 2$
$1,1 = 3$
$1,3 = 5$
$1,50 = 52$

(only two extra) Ackerman Relation

Ex- $A(3,3) = A(2, A(3,2)) \Rightarrow A(2,33) = 69$ — (try to find out any relation b/w given expression)

$\Downarrow$

$A(2, A(3,1)) \quad A(2,15) = 33$

$\Downarrow$

$A(2, A(3,0)) \quad A(2,6) = 15$

$\Downarrow$

$A(2,1) = 5$

O/P = 69

$2,1 = 5$
$2,2 = 7$
$2,3 = 9$
$2,4 = 11 \ (2\times4+3)$
$2,5 = 13 \ (2\times5+3)$
$2,6 = 15 \ (2\times6+3)$



EXCELLENT

A(1, 0 to n) = 1*n+2
A(2...0 to n) = 2*n+3
in Ackermann

Page No.
Date: / /

**Infix to Postfix-** Given a+b (Inorder)

ab+ (postfix)

+ab (prefix)

(order of operand cannot change)

**Ex1 - Infix-** a+b*c

Postfix- a+bc* = abc*+

Prefix- a+*bc = a +a*bc

**Note -** In all above 3 format, order of operands cannot be changed

**Ex2 Infix-** b-a+c

Postfix- ba-c+

Prefix - +-bac

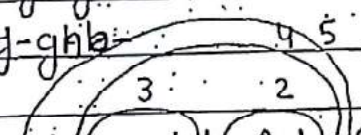
[operators with same priority getting solved acc. to their ~~precedence~~ associativity]

a+b+c+d-e-f-g-h

+, - > left associativity

*, /

ab+c+d+e-f-ghb




**Ex3 - Infix:** d-a*b/e↑f+c↑g

↑ - highest priority with right associativity.

a↑b↑c

abc↑↑

**Postfix -** d-a*b/e↑f+c↑g

d-a*b/e↑f+cg↑

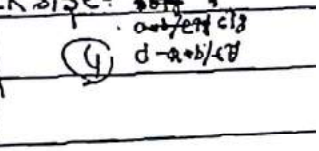
d-a*b/ef↑+cg↑

d-ab*/ef↑+cg↑

d-ab*ef↑/+cg↑

dab*ef↑/-cg↑+

Stack size- 1




EXCELLENT

Page No.
Date : / /

Prefix d-a*b/e↑f +↑cg

d-a*b/↑ef +↑cg

d-*ab↑/↑ef +↑cg

~~d-/a~~

d-/*ab↑ef +↑cg

-d/*ab↑ef +↑cg

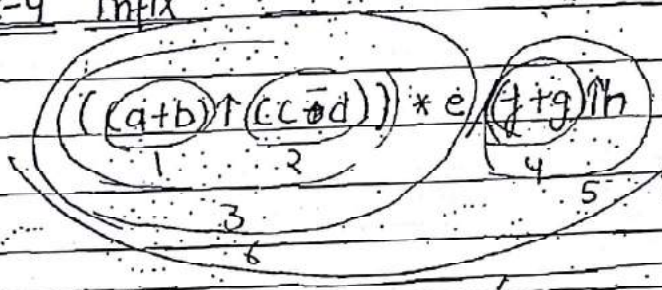
+-d/*ab↑ef↑cg

()-Left ~~Re~~ Associative:

if a question donot mention about priority of operator & you are not aiware, do left ~~to right~~.

Count outer brackets only.

Ex-4 Infix




ab+

Ex5 - ((a+b)↑(c-d))*(e/(f+g))↑h

1 2 3 4 5 6

Postfix ab+cd-↑ efg+/h↑*.

Prefix

~~*↑+ab-cd ↑/efg+h~~

*↑+ab-cd↑/e+fg h

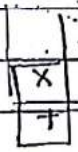
| operators | priority | Associativity |
|---|---|---|
| + - | (1)(Lowest) | L-R |
| * / | (2) | L-R |
| ↑ | (3) | R-L |
| () | (4) | L-R |

EXCELLENT

Page No.
Date : / /

Algorithm-
- if any operand
  - print operand
- if any operator
  - if has highest priority than one in stack
    - push into stack.
  - ~~else else~~
    - ~~pop the print the operator~~
  - if has ~~lesser~~ equal priority to the top of stack

a+b*c

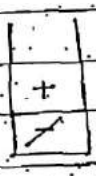



- 2 unit of operator stack.

abc*+

Ex 2 - b-a+c

b




O/P ba-c+

∴ You can not put two operator with same priority into stack.
You have to pop previous one acc. to associativity
if both have same priority, pop previous one ~~as it has not~~ in (LR) associativity.

- ~~When two operator wi~~

• Every operator ... stack at least once

Page No.
Date : / /

$a \uparrow_a b \uparrow_b c \uparrow_c d$ | $\uparrow_c$ $\uparrow_b$ $\uparrow_a$ | $abcd\uparrow_c\uparrow_b\uparrow_a$

$a+b*c\uparrow d-e/f*g\uparrow h$

| | |
|---|---|
| ~~↑~~ | $\uparrow abcd\uparrow*+ef/gh\uparrow*-$ |
| ~~*~~ | $**$ |
| ~~+~~ | $-$ |

In the algorithm, the pushed values are operators only so it is an operator stack.

Time Complexity = $n \times O(1)$ (as push, print, pop taking O(1) time)

$= O(n)$ (n - size of expression)

Ex - $d-a*b/e\uparrow f+c\uparrow g$

$dab*ef\uparrow/-cg\uparrow+$

| | |
|---|---|
| ↑ | stack size = 3 |
| * ~~↑~~ | 3 unit of operator stack. |
| / ~~*~~ | + |
| - | |

Note - Infix to postfix conversion uses operator stack.
To convert n-length infix exp into equivalent postfix, O(n) time required (for every symbol constant time (as pop, push, print is done only))

Page No.
Date : / /

When brackets are present in expression, they get pushed into stack. In stack, open bracket indicates the start of stack & closing bracket represent no one is there in stack. Whenever closing bracket came, pop until ~~closing~~ open bracket came.

$((a+b)\uparrow(c-d))*(e/(f+g))\uparrow h$

$ab+cd-\uparrow efg+/\uparrow h\uparrow *$

| | |
|---|---|
| ) | stack ends pop until another open bracket |
| + | |
| ( | new stack started |
| ( | new stack started |

brackets are also considered as operators

Prefix to Postfix:- to do this, you need 3 thing
operator
& two operand
} when you find them just circle them

1. $+ab \Rightarrow ab+$

2. $*+cd-ba \Rightarrow cd+\ ba-\ *$

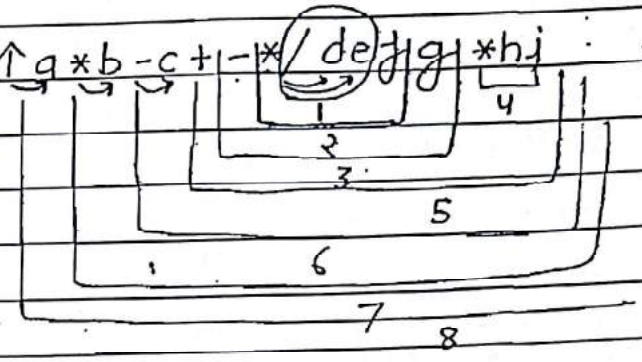
3. $*\uparrow/+ca*db\uparrow fg$

operands order cannot change.

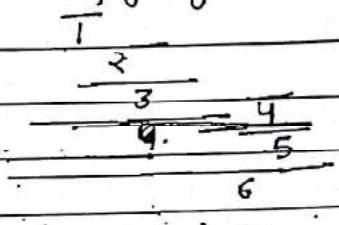
Page No.
Date : / /

Question

④ ↑a*b-c+d-e/f*g*h*i



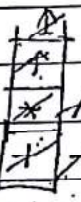

a b c de/f*g- hi*+- *↑




In this case, operand & operator both can be there in stack.
∴ Processor convert the exp. to postfix expression.

Postfix Evaluation- • Postfix expression if used to evaluate expression it take only one scan to execute the O/P as the operators are in order acc. to their precedence.
• fn pl are written in prefix.

ex - 2+3*4↑1↑5/2-10+3




Postfix

2 3 4 1 5 ↑ ↑ * 2 / + 10 - 3 +

O(n) time

When postfix Evaluation is done we use operand stack.

Operand- push into stack

an operator- pop two times
1st pop - operand2
2nd pop - operand1
take r = op1 op op2
push r in

EXCELLENT

()

Page No.
Date : / /

~~to p~~ for operand - push - O(1)
for operator - pop + pop + cal. + push = O(1)

∴ time complexity = O(n)

| | |
|---|---|
| ~~8~~ | |
| ~~1~~ | ~~1~~ |
| ~~4~~ | ~~4~~ |
| ~~3~~ | ~~12 2~~ ~~16 3~~ 1 |
| ~~2~~ | ~~6 8 2~~ |

O/P = 1

for infix - scan for each operator to whole expression to evaluate

[for postfix - O(n) only ↑ More suitable]

Ex2 Infix : 5 ↑ 1 * 2 ↑ 1 * 3 - 7 ↑ 1 * 2

Postfix - 5 1 ↑ 2 1 ↑ * 3 * 7 1 ↑ 2 * -

| | |
|---|---|
| ~~4~~ | ~~*~~ |
| ~~8~~ | ~~-~~ |

(2-Unit operator stack)

evaluation

| | |
|---|---|
| | |
| ~~4~~ | ~~4 4~~ |
| ~~2~~ | ~~2 2~~ |

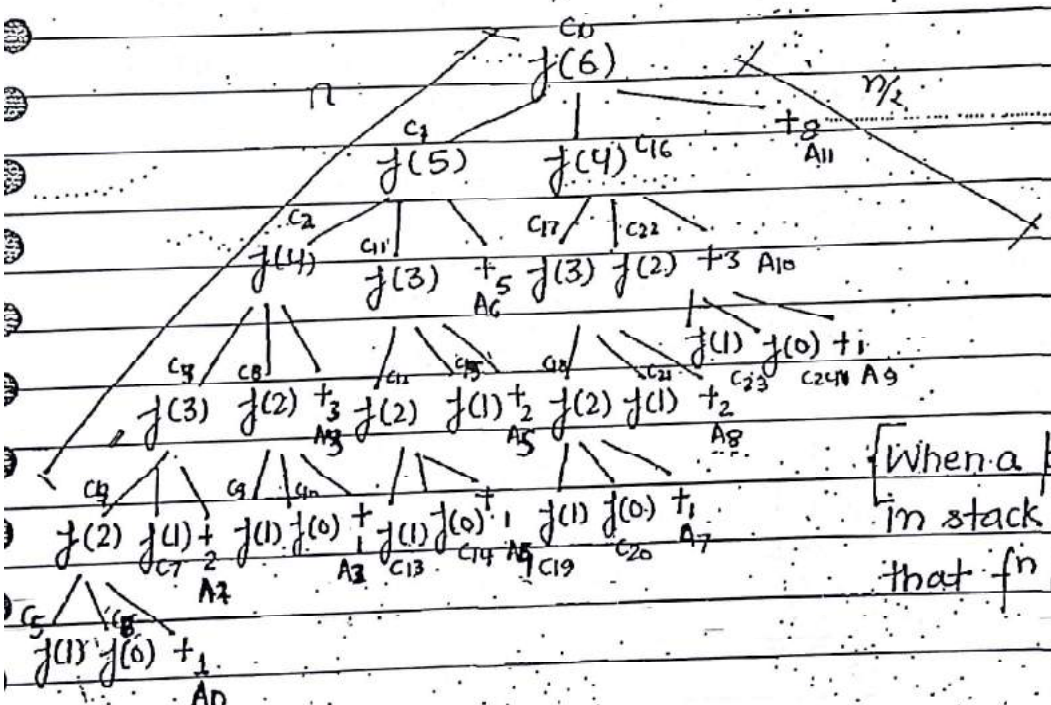
Answer = 16

Page No.
Date : / /

```
fibonacci(n)
{
    if(n==0) return 0;
    if(n==1) return 1;
    else (return (fib(n-1)+fib(n-2));
}
```

Time Complexity, $T(n) = \begin{cases} O(1) & ; \text{if } n=0 \text{ or } 1 \\ T(n-1)+T(n-2)+O(1) & ; \text{if } n \geq 2 \end{cases}$

No of add; $A(n) = \begin{cases} 0 & \text{if } n \leq 1 \\ A(n-1)+A(n-2)+1 & ; \text{if } n > 1 \end{cases}$




[When a particular $f^n$ is in stack, the parent of that $f^n$ is in stack.]

* In Every Programming language, $f^n$ calling seq: preorder & $f^n$ execution order postorder.

EXCELLENT

Rectify Daily

$2^n - 1 - 2^{n/2}$

f(2) 3

Page No.: 15

Date: / /

**Ques1-** In fibonacci of n, is there is any $f^n$ call after last $add^n$?

Ans- No $f^n$ call (After All total problem computed)

**Ques2-** In fibonacci of n, is there any $add^n$ after last $f^n$ call?

Ans - Yes, 3 Addition will be done $\lceil (n/2) \rceil$ no of $add^n$

↳ as it will be tracing the right most path of tree whose length is $\lfloor n/2 \rfloor$

**Ques3 -** In fibonacci of n, After how $f^n$ call first $add^n$ taken place?

Ans - it will occur when whole left most path get traced.

~~O(n)~~ = n $f^n$ call + 1 (as two $f^n$ call are there)

= (n+1) (after $C_7$, there is Ao.)

**Ques4 -** In Fibonacci of 6, After how many $f^n$ calls 9th addition takes place?

Ans - 22 $f^n$ calls (After $C_{22}$ there is A8).

**Ques5** In Fibonacci of n, how many $f^n$ calls are there?

$$f(n) = f(n-1) + f(n-2) + 1$$

($f^n$ call) ($f^n$ call) ($f^n$ call)

$f(0) = 1$

$f(1) = 1$

$f(2) = 3$

$f(3) = f(2) + f(1) + 1$

$= 5$

$f(4) = f(3) + f(2) + 1$

$= 9$

$$f(n) = \begin{cases} 1 & n \le 1 \\ f(n-1) + f(n-2) + 1 ; \text{if } n > 2 \end{cases}$$

for no of $f^n$ call, $f(n) = f(n-1) + f(n-2) + 1$

EXCELLEN

Page No.
Date : / /

Q(6) - In fibonacci of n, how many add$^n$ are there?

~~f(0)~~

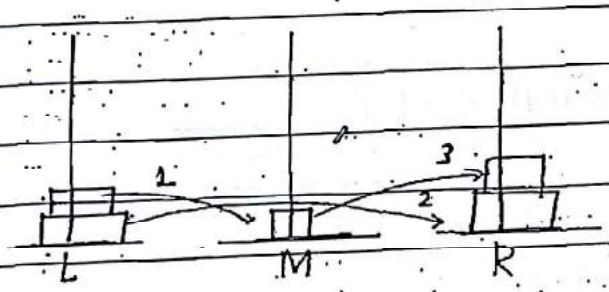
No of addition in $f(0) = 0$
$f(1) = 0$
$f(2) = 1$
$f(3) = 1+1 = 2$
$f(4) = 4$
$f(5) = 7$

$f(n) = f(n-1) + f(n-2) + 1$

No of Addn $f_n(n) = \begin{cases} 0 ; & \text{if } n \leq 1 \\ f(n-1)+f(n-2)+1 ; & \text{if } n > 1 \end{cases}$

## Tower of Hanoi

$n=2$




L-M
L-R
M-R

$n=3$

L-R
L-M
R-M
} to move top 2 to Middle

L-R } to move larger 1 to R

M-L
M-R
L-R
} to move top 2 to Right

$n=4$

| | |
|---|---|
| L-M | M-R |
| L-R | L-M |
| M-R | L-R |
| L-M | M-R |

R-L
R-M
L-M
L-R
M-R
M-L
R-L

EXCELLENT

n x, y, z
↓
n-1 x, z, y
~~x, y, z~~ print(x)
~~n-1 y, x, z~~

Page No.
Date : / /

ToH(n) = { ~~n-1~~ ToH(n-1, x, y, z);
ToH(1, x, y, z);
ToH(n-1, y, x, z);

ToH(4) = { ToH(3, L, R, M); // Move to Middle tower
Move(L→R) // Move to Right
ToH(3, M, L, R);

- for n=3, so n is obtained in the form of n=2
  - Move 2 to Middle tower
  - Move 3rd plate to destination tower.
  - Now Move 2 plates from Middle tower to destination

ToH(5) = { ToH(4, L, R, M)
Move(L→R)
ToH(4, M, L, R)

Total = 31

Total Moves, ToH(n) = 2 × ToH(n) + 1

Recursive Program-

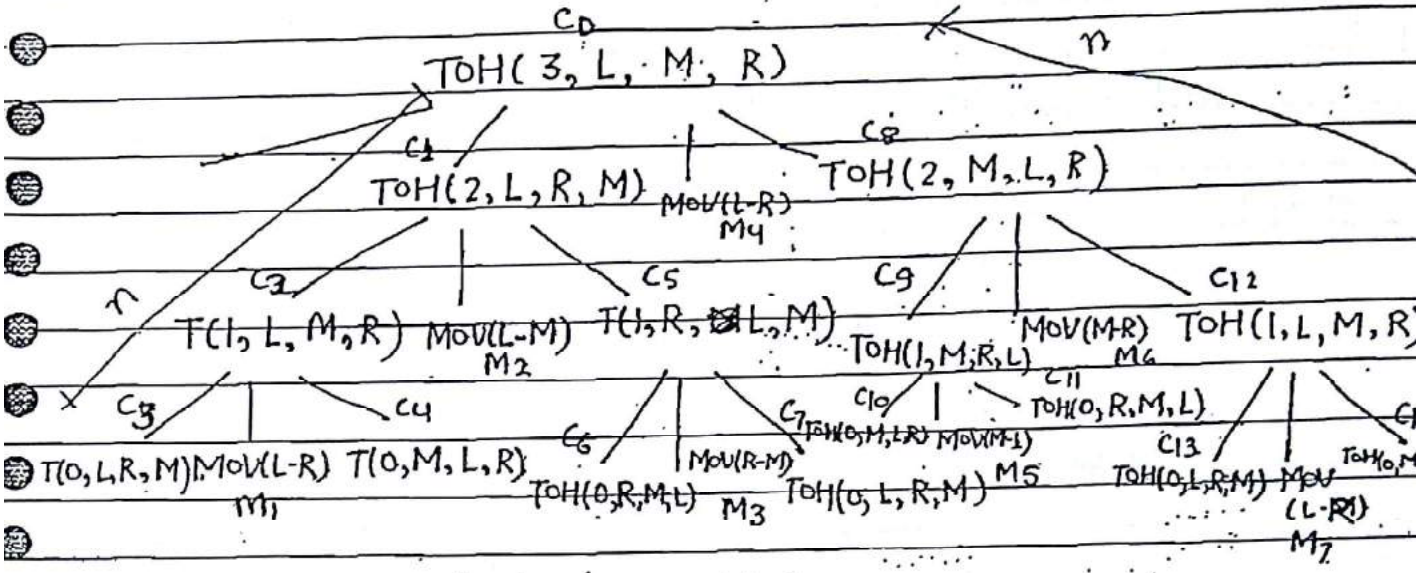
source, intermittent, destination

ToH(n, L, M, R)
{
if(n==0) return;
else
{
ToH(n-1, L, R, M)
Move(L to R)
ToH(n-1, M, L, R)
}
}

TC = $\Theta(2^n)$
↓ complete
binary tree

EXCELLENT

Without DP = $\theta(2^n)$ (improve count)




Stack space = $O(n)$

Ques1- In ToH(n), is there any $f^n$ call after Move?

Ans- 1 Mov $f^n$ call after $M_7$

$(n+1)$ $f^n$ call after last move

Ques2- In ToH(n), is there any move after last $f^n$ call?

Ans- No due to tail recursion.

(After $C_{15}$, in backtracking, no work)

Ques3- In ToH(n), after how many $f^n$ call 1st move takes place?

Ans- 4 $f^n$ call before $M_1$

$(n+1)$ $f^n$ call (left most path)

(After $C_4$, there is $M_1$)

Ques4- In ToH(3) How many $f^n$ call are there?

ToH(0) = 1 $\quad 2^{0+1}-1$

ToH(1) = 3 $\quad 2^{1+1}-1$

ToH(2) = 7 $\quad 2^{2+1}-1$

$\left[ ToH(n) = 2^{n+1}-1 \right]$

EXCELLENT

Page No.
Date : / /

No of $f^n$ call, $T(n) = \begin{cases} 0 & ; \text{if } n=0 \\ 2T(n-1)+1 \end{cases}$

as $T(3) = T(2) + T(2) + 1$

Left tree Right tree

$= 7+7+1 = 15$

Ques - IN ToH(n), How many Moves are there?

$\therefore$ TOH(0) = 0

ToH(1) = 1

ToH(2) = 3

ToH(3) = 7

$\therefore$ Moves, ToH(n) = 2 × ToH(n) + 1

or

$ToH(n) = 2^n - 1$

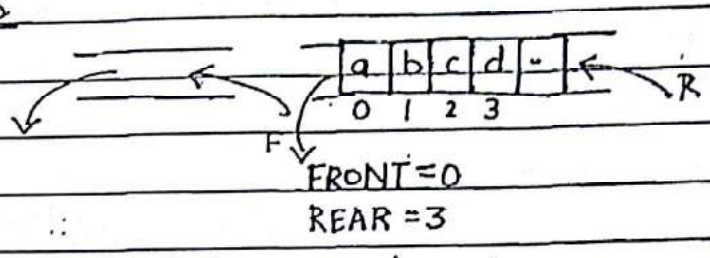
Moves, $ToH(n) = \begin{cases} 0 & \text{if } n=0 \\ 1 & \text{if } n=1 \\ 2*ToH(n)+1 & \text{; otherwise} \end{cases}$

EXCELLENT

Page No.
Date : / /

QUEUE open from both side, one side insertion, one time deletion

• FIFO



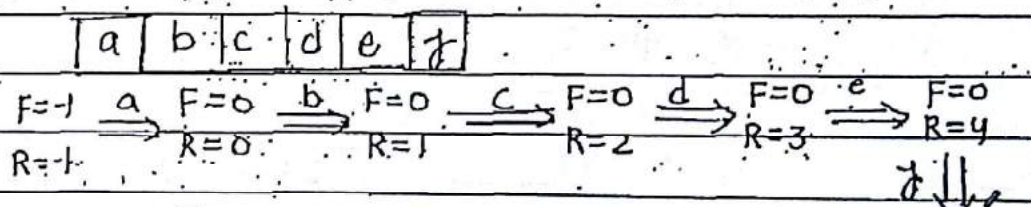

FRONT is a variable which ~~stores~~ stores the first posⁿ of the queue. it help to delete the elmt.

REAR is a variable which contain posⁿ of element newly inserted. it helps to insert the elmt.

ADT of a queue - • ENQUEUE()
• DEQUEUE()

Implementing Queue -

• Using Array -




```
int QUEUE_EMPTY()
{
    if (R == -1 && F == -1)
        return 1;
    return 0;
}
```

```
int FULL()
{
    if (R == N-1)
        return 1;
    return 0;
}
```

Date Date: / /

```
Void ENQUEUE(int x)
{
  if(R+1==N)
  { printf("QUEUE OVERFLOW"); exit(1); }
  R=R+1;
  Q[R]=x;
  if(F==-1)
    F=0;
}
```

{ Time complexity = O(1) (BC, AC, WC) }

```
int DEQUEUE()
{ int x;
  if(R==-1 && F==-1)
  { printf("QUEUE UNDERFLOW");
    exit(1);
  }
  x=Q[F];
  F=F+1;
  if(F>R)
  { F=R=-1;
  }
  return x;
}
```

{ Time Complexity = O(1) (BC, AC, WC) }

• Synchronization b/w rear & front is required here

It can also be written as

```
{ int x;
  if(R==-1 && F==-1)
  { printf("Queue Underflow");
    exit(1);
  }
  if(F==R)
  { x=Q[F];
    F=R=-1;
  }
  else
  { x=Q[F];
    F=F+1;
  }
  return x;
}
```

{ TC = O(1) }

EXCELLENT

[In Enqueue if you remove F==-1 && R==1 & use R==-1 both have same meaning]

Page No.
Date: / /

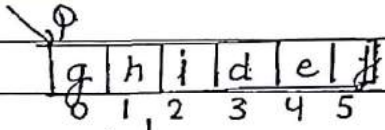
Circular Queue- Implementing queue Efficiently.

~~Linear Queue is eri~~ (Linear)

Note - Using above program, we can implement queue but it is not efficient because after rear reaching right most place it can not go back even though space is available.

To implement queue efficiently, we are moving toward circular queue.

Circular Queue-




F=3
R=5
g → in case of circular, use
{ R=(R+1)ModN }

F=3
R=2
i

F=3
R=1
1 mod 6
=(1)
h

F=3
R=0

if( F==(R+1)ModN)
then circular queue is full.

if ( F==-1 && R==-1)
circular queue is empty

void ENQUEUE(char x)
{

```
else
    R=(R+1) Mod N;
    Q[R] = x;
}
char DEQUEUE()
{
    char y;
    if( F==-1 || R==-1)
    {printf("Queue UnderFlow");
     exit(1);
    }
    else
    { y = Q[F];
      if(F==R)
      {
        F=-1; R=-1;
      }
      else
      {
        F=(F+1) Mod N;
      }
      return y;
}
```

⇒ T.C = O(1)