Notes

M-1 : Linked list

Date / /
Page No.

Data Structures

In gate →
✓ Linked list ✓ Stack ✓ Queue ✓ Tree ✓ Graph ✓ Hashing

(Revise C before proceeding)

★ **Introduction to Linked List:** It is a list of linked nodes, where prev. node contains an address of next node.

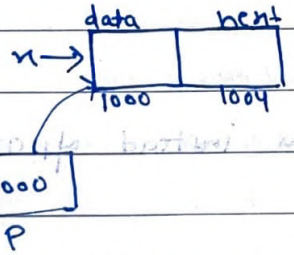
```
struct node {
    int data;
    struct node *next;
};
```

Here next is pointer to another struct node.

```
struct node *p = malloc (sizeof (struct node));
```

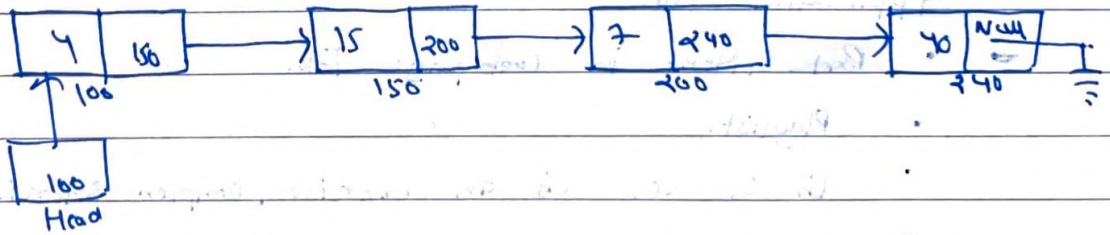
→ *p can be written here
↳ struct node * can't be written.

it will create memory in heap and assign addres to pointer p.



p → data ≡ x · data ≡ (*p) · data.

Ex:

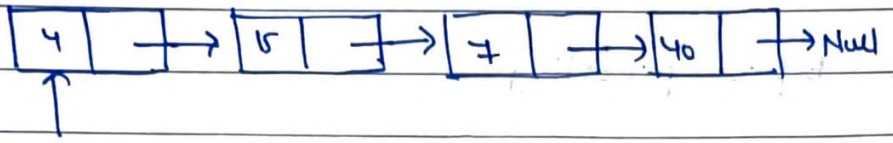


Head → data ≡ 4
Head → Next → data ≡ 15
Head → Next → Next → Next → Next ≡ NULL

Teacher's Signature _____

Notes

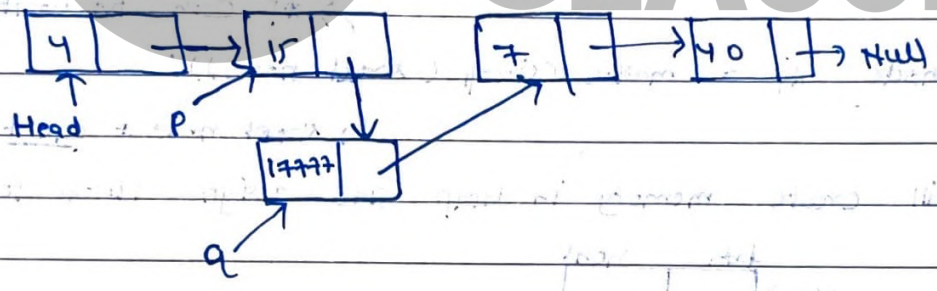
Ques: How the LL will look when we run following code on given LL?



Head

```
Struct node *p, *q;
p= head→ next;
q= malloc (size of (struct node));
q→ value = 17777;
q→ next = p→next;
p→next = q;
```


⇒



(My preference is using numbers instead of arrows).

→ Linked list is an imp. data structure. Some of its application are.

- Back & Next in web page Tab.
- Playlist.
- In OS (OS is an exercise, complex exercise of DS).
- Alt + Tab in windows.

Teacher's Signature _____

head = head→next; analogy to

Notes

→ Arrays v/s Linked list (Overview): Array is contiguous and linked list is not contiguous in memory.

	Array	LL
Cache Locality	✓	X
No. of Dynamic Elements	X	✓
Memory Usage (Volume)	✓ (less)	X (more)
Random Access	✓	X (need to traverse while LL)
Insertion / Deletion	X (need to shift all elements)	✓

★ Various Operations on Linked list.

a). Length of linked list:

```
int length (struct node *head)
{   int count=0;
    while (*head != null)
    {   count++;
        head = head→next;
    }
    return count;
}
```

b). Print Linked list:

```
Print List (struct node *head) {
    while (head != null) {
        printf ("%d", head→data);
        head = head→next;
    }
}
```

Teacher's Signature _____

Notes

Date / /
Page No

⇒ Another way to calculate length of LL given that LL has atleast one node.

```
int lenLLv2 (struct node *head)
{
    int count=1;
    while (head->next != null)
    {
        count++;
        head=head->next;
    }
    return count;
}
```

c) Insertion at beginning:

```
struct node * new = malloc (size of (struct node));
new->data = 100;
new->next = head;
head = new;
```

d) Insertion at End:

```
struct node *new = malloc (size of (struct node));
new->data = 100;
new->next = null;
while (head->next != null)   head = head->next;
head->next = new;
```


Teacher's Signature _____

Notes

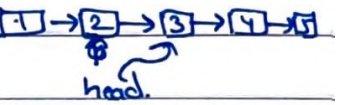
e). Insertion at middle (1,2,3,4,5) — Here

- new→data = 100
- ~~while (head→data != 3) head = head→next;~~ (call i++ karenge)
- new→next = head→next;

while (head→next→data != 3) head = head→next;

new→next = head→next;

head→next = new;



f). Deleting first node:

```
if (head == null) return;
Struct node *temp = head;
head = head→next;
free(temp);
```

g). Deleting Last Node:

```
if (head == null) return;
while (head→next→next != null) head = head→next;
free(head→next);
head→next = null;
```

h) Deleting Intermediate Node (1,2,3,4,5)

```
while (head→next→data != 3) head = head→next;
temp = head→next;
head→next = head→next→next;
free(temp);
```


Teacher's Signature _____

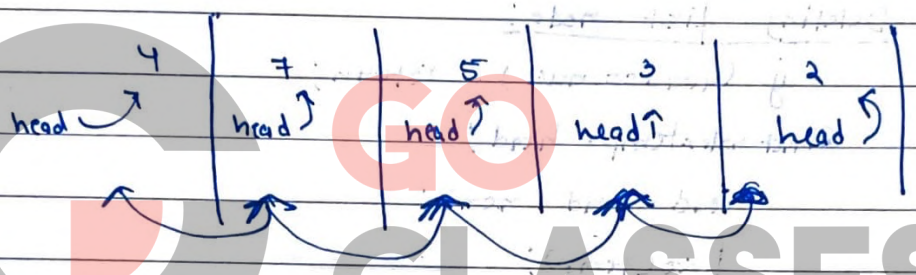
Notes

Tricky

★ Recursion In Linkedlist:

⇒ Printing Linkedlist using Recursion

```
Void printLL (struct node *head) {
    if (head != null) return;
    print (head → data);
    printLL (head → next);
}
```



⇒ Printing Ulti LL (Reverse)

```
void revLL (Node *head) {
    if (head == null) return;
    revLL (head → next);
    printf ("%d", head → data);
}
```

⇒ Sum of all nodes in LL

```
int sum (Node *head)
{
    if (head == null) return 0;
    return (head → data) + sum (head → next);
}
```

Teacher's Signature _____

Notes

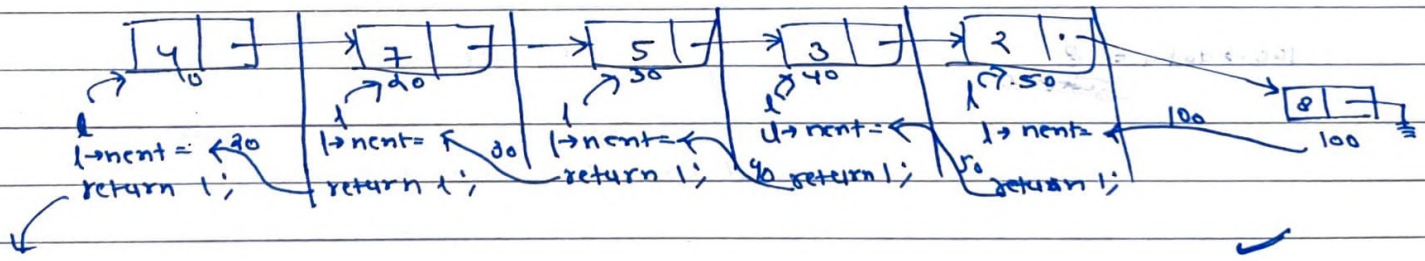
Date / /
Page No.

⇒ Length of Linked list

```
int lenLL (Node *head)
{
    if (head == null) return 0;
    return 1 + lenLL (head→next);
}
```

⇒ Insertion at end using recursion:

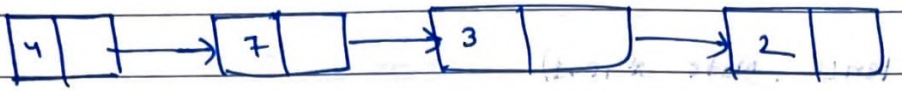
```
Node * insertRear (Node *l, int data)
{
    if (l == null)
        return CreateNode (data);   // (create new node & return address)
    else {
        l→next = insertRear (l→next, data)
        return l;
    }
}
```



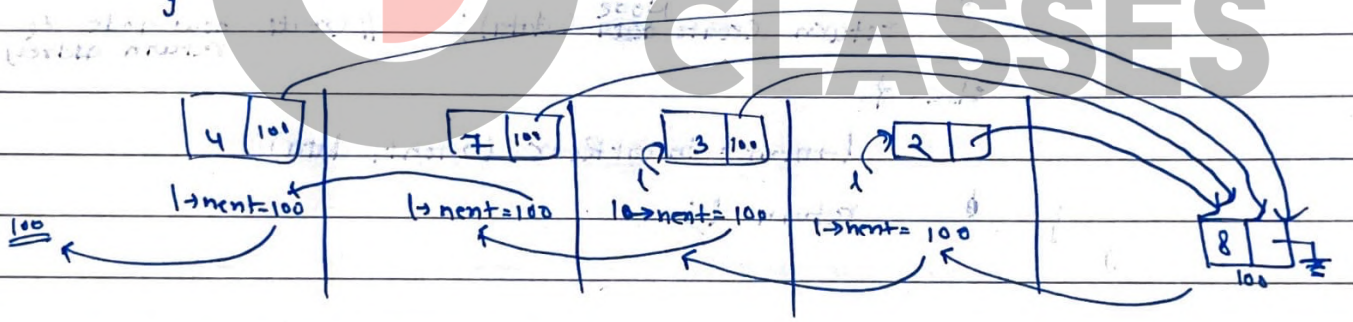
Teacher's Signature _____

Notes

Ques: fun(head, 8) 's output with following LL ?




```
Node *fun(Node *l, int data)
{
    if (l == null)
    {
        return CreateNode(data);
    }
    else {
        l->next = fun(l->next, data);
        return l->next;
    }
}
```



100 → data = 8

Teacher's Signature _____

Notes

Date / /
Page No.

Ques: What output can be expected from the function call removeFirst(head,2), removeAll(head,2) called on the provided linked list ?

$1 \to 2 \to 2 \to 8 \to 6 \to 2 \to 2 \to Null$

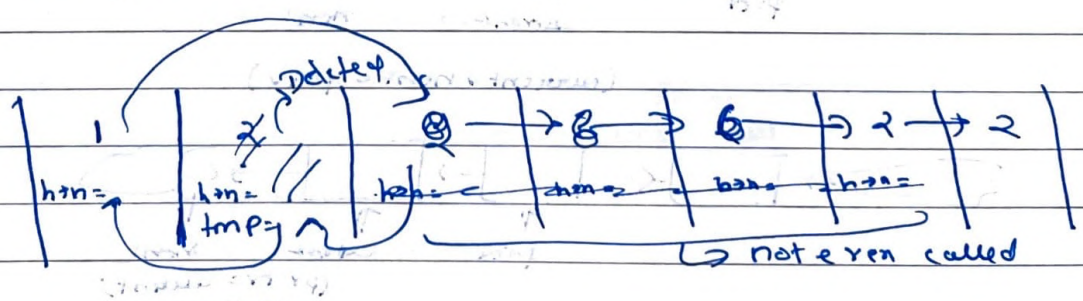
```
Node *removeFirst(Node *head, int x)
{
    if (head == null) return null;
    if (head->data == x) {
        Node *tmp = head->next;
        free(head);
        return tmp;
    }
    else {
        head->next = removeFirst(head->next, x);
        return head;
    }
}
```

o/p: $1 \to 2 \to 8 \to 6 \to 2 \to 2 \to Null$

```
Node *removeAll(Node *head, int x)
{
    if(head == null) return null;
    else{
        if (t->data == x)
            return removeAll(t->next, x);
        else{
            t->next = removeAll(t->next, x);
            return t;
        }
    }
}
```

o/p. $1 \to 8 \to 6 \to Null.$

(Not Tracing it, DIY with revision here 🙂)



Teacher's Signature _____

Notes

Leetcode Medium

Ques: Swap Nodes in pairs:

1 → 2 → 3 → 4 → NULL

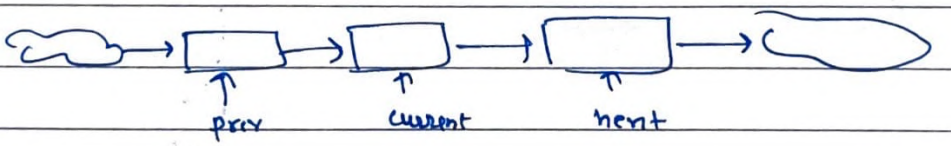
2 → 1 → 4 → 3 → NULL

```
Node * swapPairs (Node *l) {
    int temp = l->data;
    l->data = l->next->data;
    l->next->data = temp;
    swapPairs (l->next->next);
    return l;
}
```

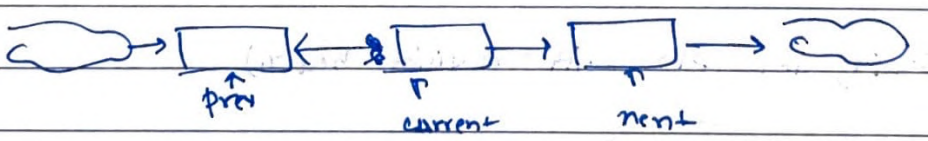
★ Reversing a Linkedlist:

Suppose initial LL:

Step 1:

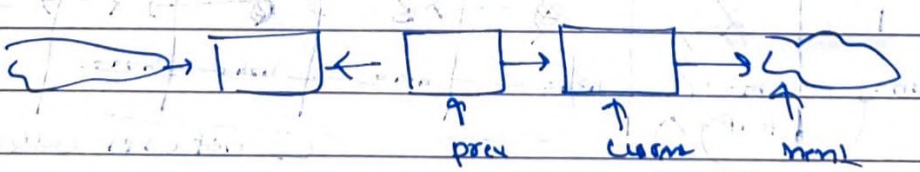


Step 2:



(current->next = prev)

Step 3:



(prev = current;
current = next;
next = next->next;)

Step 4: Repeat Step 2 & 3.

Teacher's Signature _____

Notes

Date / /
Page No.

⇒ Iterative Code for Reversing LL:

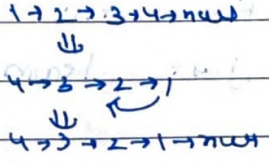
```
Node * rev (Node *l) {
    if (l== null || l->next== null) return l;
    Node *pp= null;                    // Initialisation
    Node *cp= l;
    Node *np= l->next;

    while (cp) {
        cp->next= pp;
        pp=cp;
        cp=np;
        np= np->next;
    }
    return pp;
}
```

Gate CSE 2022 Note ↔ Best algo. (i.e this above algo) is taking O(1) space & Θ(n) time in worst case.

⇒ Recursive Code for Reversing LL:

```
void reverse (Node *head)
{
    if (head== Null || head->next== null) return;
    reverse (head->next);
    head->next->next= head;
}
head->next= null;
```



This code is not able to return address of 4.

Teacher's Signature _____

Notes

Ver 3

```
Node *rev(Node *head)
{
    if(head==null || head->next==null) return;
    Node *n = rev(head->next);
    head->next->next = head;
    head->next = null;          // To make last node point null
    return n;                   // Head pointer new (last)
}
```

Ver 4

```
Node *rev(Node *head, Node *prev)
{
    if (head==null || head->next==null) return& prev;
    Node *n = reverse(head->next, head);
    head->next = prev;
    return n;
}  rev(head, null);
```

Ver 5

```
Node *rev(Node *head, Node *prev)
{
    if (head == Null) return prev;
    Node * nextP = head->next;
    head->next = prev;
    return reverse(nextP, head);
}
```

NOTE → for GATE, we need not to know to write code, just learn analysing the code.

Notes

Date / /
Page No.

★ **Circular Linked List**: This is a variation of LL, in which last node also points to first node, instead of pointing to null.

Ex: → 1 → 2 → 3 → 4 →

→ Method to find length of LL is: (Regular LL)

```
int len = 0;
for (struct node *c = head; c != null; c = c->next)
    len ++;
print (len);
```

→ But the below code will not work as condition will be false in first pass itself bcz cur == head in first pass

```
int len = 0;
for (struct node *c = head; c != head; c = c->next)   // issue
    len ++;
print (len);
```

→ To count the length and avoid this issue either we do while loop or start with len = 1 (for this atleast 1 node is req)

```
int len = 1;
for (struct node *c = head->next; c != head; c = c->next)
len ++;
print (len);
```

```
int len = 0;
Struct node *c = head;
do {
    c = c->next;
} len ++;
while (c != head);
print (len);
```

Teacher's Signature _____

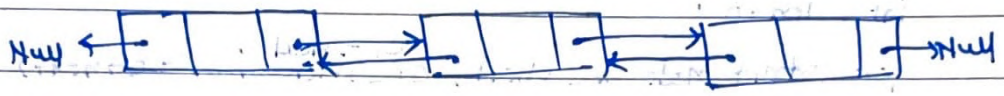
Notes

Date / /
Page No.

→ Similarly for printing circular LL, use do while loop.

→ Easily can insert node at end of LL and also at start (start = add at end + shift head to new node).

Doubly linked list: In this we can navigate both of the sides.




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

→ Easily do insertions and deletions, just take care of pointer adjustments.

Notes

M-2 Asymptotic Notations

Date / /
Page No.

→ Algorithm is a set of steps for a task.

→ To know the wellness of program/algo (i.e. will my algo. be able to solve a large practical input), we should be aware of following two questions:

a) Why is my program so slow? (Time Complexity)
b) Why does it run out of memory? (Space Complexity)

→ Various way to measure the time complexity:

a) Using Manual Clock & trying various inputs
   → Not at all practical.
b) Asymptotic Analysis
   → ✓

→ How can we determine the time it takes for an algo. to run if it can:

- Run of diff. H/w
- Clock Speed
- diff. Instruction Set
- diff memory access speed

} A Posteriori analysis (Not Practical).

So, instead of measuring time, we should measure something like no. of steps. — A priori analysis

→ Measuring Time → Posteriori
Counting steps. → Priori

Teacher's Signature _____

Notes

Date / /
Page No.

Ex: Sum of elements of an array

Algo. Sum (a, n)	T.C. (No. of steps)	S.C.
{	0	1 for sum
sum = 0	1	1 for i
for i = 1 to n do	n+1	1 for n
sum = sum + a[i];	n	n for a.
return sum	1	
}		
Total :	2n+3	(n+3) words

Instead of exact steps, we found some bound based on input size (n here).

★ **Asymptotic Analysis:** Our goal is to simplify analysis of running time by getting rid of details like rounding 1000001 to 1000000

So $3n^2 \simeq n^2$

So, it is a technique that focuses analysis on the most significant term.

It captures the essence, how the running time of an algorithm increases with the size of input.

→ for $\theta(m+n)$ we T(greater(m,n))

Notes

There are 5 asymptotic notations (Maths topic BTW):

- Big-Oh O (≤)
- Big-Omega Ω (≥)
- Theta Θ (=)
- little-oh o (<)
- little-omega ω (>)

It is matters for large inputs only.

★ **Big-Oh Notations:** $T(n)$ is $O(g(n))$ (order of big oh of $g(n)$) if there exists a constant $c>0$ & $n_0 \geq 0$ so that for all $n \geq n_0$:

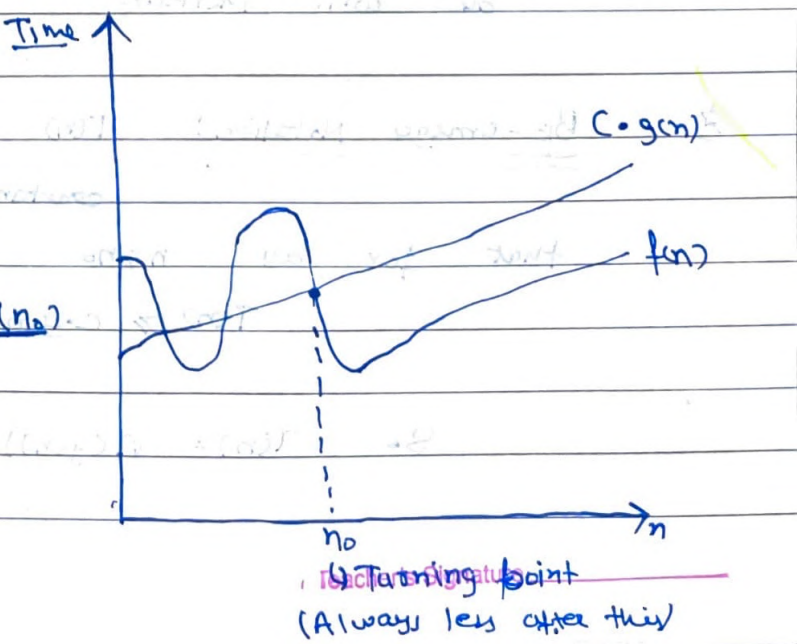
$$T(n) \leq c\,g(n)$$

So: $T(n) = O(g(n)) \equiv T(n) \leq c \cdot g(n)$

- Big-Oh provides asymptotic upper bound.
- $T(n) = O(n^2)$ means time for any input of size n is maximum n^2, i.e. $\leq n^2$. (This is upper limit)
- If $T(n) = O(n^2)$ then $T(n) = O(n^3)$ bcz if it is less than equal to n^2, then obviously less than equal to n^3.

- Ex: $f(n) = n^2$ $\quad c \cdot g(n) = 2^n$

1	1	<	2
2	4	=	4
3	9	>	8
4	16	=	16 (n_0)
5	25	<	32
6	36	<	64
7	49	<	128
↓	↓	↓	↓



Notes

Ex: a) $3n+2 = O(n)$ for $(c, n_0) = (1000, 1), (4, 3)$, etc.

→ In exam value of c & n_0 are not asked; just doing for sake of practice.

b) $1+n+n^2 = O(n^2)$

$\Rightarrow 1+n+n^2 \leq cn^2$

$c=3, n_0=2$, i.e. after this cn^2 is always greater

$c=4, n_0=5$,

$c=1000, n_0=1000$, etc.

c) $100n+6 = O(n)$

$c=101, n_0=5$

d) $2^{2n} = O(2^n) \Rightarrow 2^{2n} \leq c2^n \Rightarrow 2^n 2^n \leq c2^n \Rightarrow 2^n \leq c$

∴ False — Never

NOTE: In asymptotic analysis we always talk about the increasing function, bcz on increasing i/p size T.C. will never decrease either it will be constant or will increase.

★ Big-omega Notation: $T(n)$ is $\Omega(g(n))$ if there exists constant $c>0$ & $n_0 \geq 0$ so that for all $n \geq n_0$

$T(n) \geq c \cdot g(n)$

So $T(n) = \Omega(g(n)) \equiv T(n) \geq c\,g(n)$

Teacher's Signature _____

T.C ≡ Time Complexity

Notes

→ So, it provide asymptotic lower bound.

→ $T(n) = \Omega(n^2)$ means that T.C. for any input n is minimum n^2 (can't be less) i.e. $\geq n^2$ (This is lower limit) T.C. can't be less than n^2.

→ $T(n)$ is $\Omega(g(n))$ iff $g(n)$ is $O(T(n))$.

ie. if $T(n) \geq g(n)$, then $g(n) \leq \frac{1}{c} T(n)$

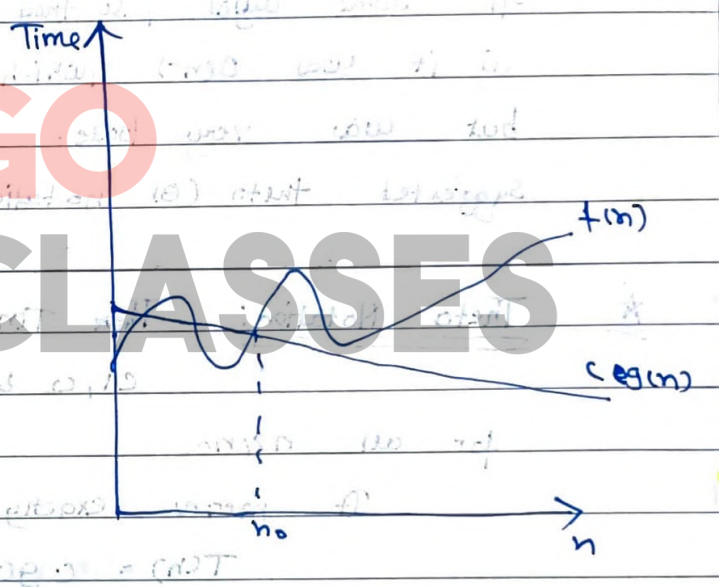
& if $g(n) \leq \frac{1}{c} T(n)$, then $T(n) \geq g(n)$

$f(n) \geq c\, g(n)$

$10n^2 \geq \Omega(100\, n^2)$

$10n^2 \geq c \cdot 100\, n^2$

$c = \frac{1}{100}$



→ if $T(n) = \Omega(n^2)$ then $T(n) = \Omega(n)$ bcz if it take min. n^2 time then obviously min. n time. (anything smaller than n^2).

→ $T(n) \geq c \cdot n^2 \equiv T(n) = \Omega(n^2)$ i.e. you will take at least n^2 time.

→ $T(n) \leq c \cdot n^2 \equiv T(n) = O(n^2)$ i.e. you will take at most n^2 time.

Teacher's Signature _____

Notes

Date / /
Page No.

→ $T(n) = \Omega(1)$ & $T(n) = O(\infty)$ is always True bcz at least constant time & atmost ∞ time. (Big no.)

Ex: (a) $3n+2 = \Omega(n)$

$\Rightarrow 3n+2 \geq c \cdot n$

$(c=1, n_0=1)$ (only one value req. for which, it's True)

→ Scientist donald Kruth asked his fellows to find O & Ω bounds of some algo, so they wrote $\Omega(1)$ & $O(n^{100})$ even if it was $O(n^2)$, which was mathematically correct but was very loose. So, to get rid of this he suggested theta (Θ) notation which meant =.

★ **Theta Notation:** $T(n) = \Theta(g(n))$ if there exist c_1, c_2 & n_0 such that $c_1 g(n) \leq T(n) \leq c_2 g(n)$ for all $n \geq n_0$.

It means exactly equal to.

$T(n) = c \cdot g(n)$

$T(n) = \Theta(g(n)) \equiv T(n) = c \cdot g(n)$

If $T(n) = \Omega(g(n))$ & $T(n) = O(g(n))$ then, $T(n) = \Theta(g(n))$,

If $T(n) = \Theta(g(n))$, then

then, $T(n) = \Omega(g(n))$ & $T(n) = O(g(n))$.

- $T(n) = \Theta(n^2)$ means T.C. is equal to n^2 for large n.

Teacher's Signature _____

Notes

Ex: $3n^2 = \Theta(n^2)$

bcz $2n^2 \leq 3n^2 \leq 4n^2$

$c_1 \qquad\qquad c_2$

for this $f(n) = \Theta(g(n))$ write

$c_1 g(n) \leq f(n) \leq c_2 g(n)$

- $3n+10 = \Theta(n) \neq \Theta(n^2) = O(n) = O(n^2) \neq O(\log n) = \Omega(n) \neq \Omega(n^2) = \Omega(\log n)$

Summary: a) Algo takes $O(n^2) \equiv$ for every input it takes $\leq n^2$

b) Algo takes $\Theta(n^2) \equiv$ for every input it takes $= n^2$

c) Algo take $\Omega(n^2) \equiv$ for every input it taking $\geq n^2$

Qui:

$f(n)$	$g(n)$	Ans:
n	n^2	$f(n)$ is $O(g(n))$
n	n^2	$g(n)$ is $\Omega(f(n))$
n^2	2^n	$f(n)$ is $O(g(n))$

linear ↑ quadratic ↑ exponential ↑

$\star$ $\log n < n < n\log n < n^2 < n^3 < 2^n < n^n$

$\log n$: logarithmic; $n\log n$: linearithmic; n^3: cubic

- $f(n) = \Theta(g(n)) \Rightarrow f(n) = \Omega(g(n))$
- $f(n) = O(g(n)) \not\Rightarrow f(n) = \Theta(g(n))$
- $f(n) = \Omega(g(n)) \not\Rightarrow f(n) = \Theta(g(n))$

→ Term asymptotically larger means we are ignoring constant (details) & looking at significant term.

i.e. $2n^2+3 \sim n^2$

Teacher's Signature _____

Notes

- $n^2 + 100000^{10000^{10000}} < n^3$

↳ this is constant. (ignore always, never matter size)

bcz, we talk about large ns.

- $n^2 < n^3 \Rightarrow 1 < n$ ✓

- $n^{5+n} \neq n^n$

(Can't ignore this constant 5)

$n^5 \cdot n^n > n^n$

$n^5 > 1$ ∴ True.

- $2^{2n} \neq 2^n$

$2^n \cdot 2^n \neq 2^n$

$\Rightarrow 2^n > 1$

→ So, the point is we can't always ignore Constants we can ignore if it is out of function. like c.e.d can be ignored but not k. in following

- $cf(n) \equiv f(n)$
- $f(n) + d \equiv f(n)$
- $f(kn) \not\equiv f(n)$

} Asymptotically.

Imp →

$(\log n)^k < n^\epsilon$ is always true, even if k is too large & ϵ is too small.

$\boxed{(\log n)^{100000} < n^{0.001}}$, $\epsilon > 0$

Imp →

$n! \not< n^n$ but $\log n! \equiv \log n^n \equiv n \log n$.

Teacher's Signature _____

Notes

Ques: Prove:

a) $2^n < n^n$

$\log 2^n < \log n^n$

$\log 2 < \log n$

b) $n^{\log n} > (\log n)^{\log n}$

$n > (\log n)^9$

$(n^k > (\log n)^k)$

Q. We can also do such que. by taking values but that value may be less than no and we may go wrong.

Imp.

If after taking log on $f(n)$ & $g(n)$, $\log f(n) > \log g(n)$ then surely $f(n) > g(n)$ but if $\log f(n) = \log g(n)$ then we cannot say $f(n) = g(n)$.

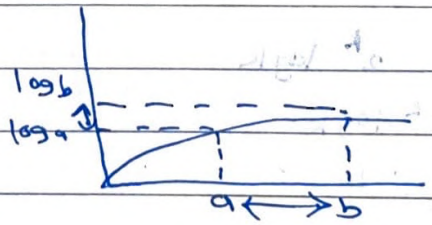
Ex: $f(n) = n^2$ $g(n) = n^3$

$f(n) < g(n)$

$\log f(n) = \log g(n)$ i.e. $\log n = \log n$

If log is small then no. is obviously small but if log is equal then no. can be small or equal or greater. So, we can say log is order preserving if $a > b$ then $\log a > \log b$ for real no. but asymptotically $\log a = \log b$ is possible.

If diff b/w a&b is too much then that b/w $\log a$ & $\log b$ will be small



Teacher's Signature _____

$a^{b^c} \equiv a^{(b^c)} \not\equiv (a^b)^c$

Notes $(a^b)^c \equiv a^{bc} \not\equiv a^{b^c}$

Date / / Page No

→ In asymptotic analysis if $a > b$, then $\log a \geq \log b$ or $\log a = \log b$ but $\log a \not< \log b$

→ If $\log a = \log b$ then $a < b$, $a = b$, $a > b$, any of these is possible.

→ So, now for asymptotic analysis always cancel out common terms.

$n^2 < n^3$ } With log this will not work.

$1 < n$ True.

→ $\log f(n) > \log g(n) \Rightarrow f(n) > g(n)$

$\log f(n) < \log g(n) \Rightarrow f(n) < g(n)$

$\log f(n) = \log g(n) \Rightarrow$ Can't say.

→ We may need to assume n as 2^k, 2^{k^2}, 2^{100^k}, 2^{2^k}, etc.

Ex: $(\log n)^{\log\log n} < (\log\log n)^{\log n}$

⇒ Let $n = 2^{2^k}$

$\log_2 n = 2^k$

$\log\log n = k$

So,

$(2^k)^k < (k)^{2^k}$

$2^{k^2} < k^{2^k}$

take log

$k^2 \log 2 < 2^k \log k$

$k^2 < 2^k \log k$

∴ ✓

Teacher's Signature _____

Notes

b) $n2^n < 4^n$

$n < \frac{4^n}{2^n}$ (/ by 2^n)

$n < 2^n$ ✓

c) $n^2 \log n < n^{1.0001}$

$\log n < n^{.0001}$ ✓

d) $(\log n)^k < n^\epsilon$

$k \log\log n < \epsilon \log n$

$\log\log n < \log n$

let $\log n = k$

$\log k < k$ ✓

e) $n^{\log n} < 2^n$

$\log n \cdot \log n < n \log_2 2$

$(\log n)^2 < n$ ✓

f) $\log_2 n = \log_{10} n$

$\log_2 n = \frac{\log_2 n}{\log_2 10}$ (Changing base)

$\log_2 n = \log_2 n$ ✓

g) $n^{\sqrt{n}} < 2^n$

$\sqrt{n} \log n < n \log 2$

$\sqrt{n} \log n < n$

$\log n < \sqrt{n}$ (/ by $\sqrt{n}$) ✓

g) $n^{\log_2 n} > n^{\log_{10} n}$

$\log_2 n \log n > \log_{10} n \log n$

$\log n = \log n$

(Can't say by this method)

$\log_2 n = 10$, $\log_{10} n = 3$ (Eg $n = 1024$)

but w.k.t. $\log_2 n > \log_{10} n$ for any n

and w.k.t. ~~$\log_a n$ for base~~ $n^a > n^b$ if $a > b$

So

$n^{\log_2 n} > n^{\log_{10} n}$ bcz. $\log_2 n > \log_{10} n$ ✓

Teacher's Signature _____

$n! > 2^n$ Take care of normal le asymptotic comparisons

⇒ $f(n) > g(n)$ is normal comparison ke if it's true

Notes then $f(n) = \theta(g(n))$ can also be true (take $2n$ & n).

Date / /
Page No.

Que: Arrange following function in order of asymptotic growth rate:

⇒ a) $\log n$ b) $(\log n)^{10}$ c) $\log\log n$ d) $(\log(\log n))^{10}$

⇒ Let $k = \log n$

$a = k$, $b = k^{10}$ c) $\log k$ d) $(\log k)^{10}$

$k^{10} > k > (\log k)^{10} > \log k$

$b > a > d > c$ ✓

⇒ a) 2^{2^n} b) $n!$ c) 4^n d) 2^n

w.k.t. b) c > d

now we need to put a

$\log n! = n\log n$

$\log 2^{2^n} = 2^n$

and $2^n > n\log n$

so

$a > b > c > d$ ✓

⇒ a) $2^{\log n}$ b) $(\log n)^2$ c) $\sqrt{\log n}$ d) $\log\log n$

Let $k = \log n$

$a = 2^k$, $b = k^2$, $c = \sqrt{k}$, $d = \log k$

$2^k > k^2 > \sqrt{k} > \log k$

$a > b > c > d$

Teacher's Signature _____

$n^2 < n! < n^n$

Notes

★ little-oh: $T(n)$ is $o(g(n))$ for all ($\forall c$) constant $c > 0$, there is $n_0 > 0$, so that for all $n \geq n_0$.

$T(n) < c \cdot g(n)$

↳ More strict ($\leq$ not allowed)

→ In O (big-oh), it was $\exists c, \exists n_0$. Here $\forall c, \exists n_0$.

⇒ $n^2 \neq o(2n^2)$

$n^2 < c \cdot 2n^2$ is not true for $c = \frac{1}{10}$, so it is not True

but

$n^2 = O(2n^2)$ bcz, there we need any one c (Ex. $c = 2$)

⇒ $n^2 = o(n^3)$

bcz $n^2 < cn^3$ bcz. for every c it is True

Ex: $c = \frac{1}{1000}$

$n^2 < \frac{n^3}{1000}$, taken $n_0 = 1000000$.

$f(n) = O(g(n)) \not\Rightarrow f(n) = o(g(n))$

$f(n) = o(g(n)) \Rightarrow f(n) = O(g(n))$

$f(n) = o(g(n)) \Rightarrow f(n) \neq \Theta(g(n))$

$O \Rightarrow \not<$, $o \Rightarrow <$

$2^n = o(n!)$

$n! = o(n^n)$

$\log n! \neq o(\log n^n)$

Teacher's Signature _____

Notes

Date / /
Page No.

★ **Little omega (ω):** $T(n)$ is $\omega(g(n))$ for all constant $c > 0$ there is $n_0 > 0$, so that for all $n \geq n_0$:

$$T(n) > c \cdot g(n)$$

- $T(n) = \omega(g(n))$ iff $g(n) = o(T(n))$.

So. $O \to \leq$, $\Omega \to \geq$, $\Theta \to =$, $o \to <$, $\omega \to >$

⇒ **Incomparability:** $n^{1+\sin n}$ is incomparable to n since $\sin n$ oscillates b/w -1 & 1, So. there is no n_0.

★ Some properties of asymptotic notations:

a) **Transitivity:**

$f(n) = \#(g(n))$ & $g(n) = \#(h(n)) \Rightarrow f(n) = \#(h(n))$

where $\# \in (\Theta, O, \omega, o, \Omega)$

b) **Reflexivity:**

$f(n) = \#(f(n))$

where $\# \in (\Theta, O, \Omega)$ & $\# \notin (o, \omega)$

c) **Symmetry:**

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

only for Θ

Teacher's Signature _____

→ Increasing function $t(n)$ & $g(n)$ will be either $t(n) \in O(g(n))$ or $t(n) \in \Omega(g(n))$. or both, is false bcz. sometimes, they are incomparable.

→ Note: $f(n) = \Theta(g(n))$ & $f(n) = \Omega(g(n))$, then $f(n) = \Theta(g(n))$ but $f(n) \neq g(n)$ ↳ bcz. normal comparison

d) Transpose Symmetry:

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

$f(n) = o(g(n))$ iff $g(n) = \omega(f(n))$

Not Req.

☆ Stirling's Approximation:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

→ Prove $n! < n^n$

Proof: $\frac{\sqrt{2\pi n}\, n^n}{e^n} < n^n$

$\frac{\sqrt{2\pi n}}{e^n} < 1$

$\sqrt{2\pi n} < e^n$ ⇒ $\sqrt{n} < e^n$ (Always True)

→ Prove $\log n! = \log n^n$

Proof: $\log\left[\frac{\sqrt{2\pi n}\, n^n}{e^n}\right] = n\log n$

$\log\sqrt{2\pi n} + \log n^n - \log e^n = n\log n$

$\log\sqrt{2\pi} + \log\sqrt{n} + n(\log n - \log e) = n\log n$

$\left(\frac{1}{2} + n\right)(\log n) = n\log n$

$n\log n = n\log n$

Teacher's Signature _____

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

$\Downarrow$ ~~$\Uparrow$~~ (one way)

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

Notes

Date / Page No.

Ques: Compare below functions

a) $n^{\sqrt{n}}$, b) 2^n , c) $n^{\log n}$ & $n!$

take log in all

a = $\sqrt{n}\log n$, b = n , c = $(\log n)^2$, d = $n \log n$

$b > a$,

$b > c$, & $a > c$

$d > b$,

So, $d > b > a > c$

NOTE While writing $f(n) = O(g(n))$ there is abuse of notation. Actually $O(g(n))$ is a set of functions such that $\{h(n) \mid h(n) \leq g(n)\}$. And here it must be $f(n) \in O(g(n))$ instead of $=$. But we abuse notation.

So, $f(n) = O(g(n))$ means $f(n) \in O(g(n))$ &

$O(g(n)) = \{h(n) \mid h(n) \leq (g(n))\}$

i.e. this foolish thing is wrong:

$n^2 = O(n^3)$

$n^3 = O(n^3)$

So $n^2 = n^3$ this is wrong due to notation abuse.

$O(1) = \{1, 1000, 20, \text{etc.}\}$

$O(n) = \{n, 3n, n+100 \ldots\}$

$O(n^2) = \{5n^2, n^2+100, n, 20 \ldots\}$.

Teacher's Signature _____

Notes

(Not writing PYQs) (Easy) (DIY) (Video - 6b)

↳ 94, 96, 00, 01, 03, 04, 08, 11, 15, 17

★ Analyzing loop Complexities:

- $1+2+3+\dots+n = \frac{n(n+1)}{2}$
- $1^2+2^2+3^2+\dots+n^2 = \frac{n(n+1)(2n+1)}{6}$
- $\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\frac{1}{3}+\dots+\frac{1}{n} \approx \log n$
- $\sum_{k=1}^{n} \log k = \log 1+\log 2+\log 3+\dots+\log n \simeq n\log n$ (= log(n!))

Ex:1
```
for (i=0 to n)
    n=y+2;
```
T.C = O(n) = Θ(n) = Ω(n)

big-oh is more popular, so we will use this.

However Θ is more strict & informative, but O is more popular

Ex:2
```
for (i=0 to n)
    for (j=0 to n)
        n=y+2;
```
O(n²)

Ex:3
```
for (j=0 to n)
    n=y+2;
for (i=0 to n)
    for (j=0 to n)
        n=y+2;
```
$n+n^2$ = O(n²)

Teacher's Signature _____

Notes

Ex:4

```
for(i=0 to n)          2n
    n=y+z;             O(n)
    a=b+c;
```

Ex:5

```
stmt 1;                k
stmt 2;                O(1)
 |
 |
stmt k;
```

Ex:6

```
if (cndtn)             1
    stmt;              O(1)
else
    stmt;
```

Ex:7

```
for(i=0; i<n; i++){
    for(j=i+1; j<n; j++){
        stmt;
    }
}
```

i=0 i=1 i=2 ... i=n-1 i=n

$\Rightarrow n + (n-1) + (n-2) \dots + 1 + 0$

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

Ex:8

```
for (i=0 to m);
    a=b+c
for (j=0 to n);
    b=a+c
```

$\Rightarrow O(m,n)$ → whichever is greater

Teacher's Signature _____

Notes

Ex:9

```
for(i=1; i<n; i+=2) {
    stmt;
}
```

=

```
for(i=n; i>=1; i-=2) {
    stmt;
}
```

⇒ Let no. of iterations = k.

Iteration	i	
1	1	
2	1+2	1+(2×1)
3	1+2+2	1+(2×2)
4	1+2+2+2	1+(2×3)
⋮	⋮	⋮
k	$1+\underbrace{2+2+\dots+2}_{(k-1)}$	1+(2*(k-1))

And $1+(2*(k-1)) = n$

$k = \frac{n-1}{2}+1$

$$\boxed{k = \frac{n}{2}}$$

So. $O(n)$

Ex:10

```
for(i=1; i<n; i+=a)
    stmt;
```

=

```
for(i=n; i>=1; i-=a)
    stmt;
```

⇒ Total iteration = $\frac{n}{a}$

$O(\frac{n}{a}) = O(n)$

Teacher's Signature _____

Notes

$i = 1 \xrightarrow{+1} 2 \xrightarrow{+1} 3 \xrightarrow{+1} 4 \quad \_ \_ \_ \_ \quad N$ — N steps

$i = 1 \xrightarrow{+2} 3 \xrightarrow{+2} 5 \xrightarrow{+2} 7 \quad \_ \_ \quad N$ — $\frac{N}{2}$ steps

$i = 1 \xrightarrow{+a} 1+a \xrightarrow{+a} 1+2a \xrightarrow{+a} 1+3a \quad \_ \_ \quad N$ — $\frac{N}{a}$ steps

$i = N \xrightarrow{-1} N-1 \xrightarrow{-1} N-2 \quad \_ \_ \quad 3 \xrightarrow{-1} 2 \xrightarrow{-1} 1$ — N steps

$i = N \xrightarrow{-a} N-a \xrightarrow{-a} N-2a \quad \_ \_ \_ \_ \quad 1+a \xrightarrow{-a} 1$ — $\frac{N}{a}$ steps

Ex:1

```
for(i=1 ; i<=N ; i=i*2)        ≡   for(i=N ; i>=1 ; i=i/2)
    stmt;                          stmt;
```

Iteration	i
1	1
2	2
3	2^2
4	2^3
⋮	⋮
k	2^{k-1}

$2^{k-1} = N$

$\boxed{k = \log_2 N}$

$O(\log N)$

Teacher's Signature _____

Notes

Date / /
Page No.

Ex:12 `for(i=1 ; i≤N ; i=i*a)`
`stmt;`

No. of steps = $\log_a N$

$O(\log N)$

≡ `for(i=N ; i>=1 ; i=i/a)`
`stmt;`

Ex:13 `for(i=1 ; i²<N ; i++)`
`stmt;`

iteration	i
1	1
2	2
3	3
.	.
.	.
k	$\sqrt{n}$

$k = \sqrt{n}$

$O(\sqrt{n})$

Ex:14 `for(i=1 ; i≤2^N ; i=i*2)`
`stmt;`

iteration	i
1	1
2	2
3	2^2
.	.
.	.
k	2^k

$2^k = 2^n$

$k = n$

$O(n)$

Ex:15 `for(i=2 ; i<=n ; i=i^c)`
`n=y+2;`

iteration	i
1	2
2	2^2
3	$(2^2)^2 = 2^4$
4	$(2^4)^2 = 2^8$
.	.
.	.
k-1	2^{2^k}

So, $(i=k)$

$2^{2^k} = N$

$2^k \log_2 = \log N$

$k = \log \log N$

$O(\log \log N)$

Teacher's Signature _____

Notes

Ex:16 `for(i=n; i>=2; i=sqrt(i))`
`stmt;`

⇒ O(loglogn)

Ex:17 `for(i=n^2; i>=2; i=sqrt(i))`
`stmt;`

⇒ 1+loglogn
O(loglogn)

(loglog n^2 ⇒ log2 + loglogn ⇒ 1+loglogn)

Ex:18 `for(i=n/2; i<=n; i=i*2)`
`stmt;`

⇒ O(1)

Summary:

i=i+a	⇒	O(n/a)
i=i*a	⇒	$O(\log_a n)$
i=i^a	⇒	$O(\log_a \log_a n)$

Ex:19
```
for(i=1; i<=N; i++)          → N
    for(j=1; j<=i; j=j+i)    → i/i
        stmt;
```

O(n)

Teacher's Signature _____

Notes

Ex: 20

```
for(i=1 ; i<=N ; i++)          → N
    for(j=1 ; j<=i² ; j=j+i)   → i²/i = i
        stmt;
```

$1+2+3+\dots+N$

$O(n^2)$

Ex: 21

```
for(i=1 ; i<=n ; i++)          n
    for(j=1 ; j<=n ; j=j*i)    n/i
        stmt;
```

$\frac{n}{1}+\frac{n}{2}+\frac{n}{3}\dots+\frac{n}{n}$

$n\left(1+\frac{1}{2}+\frac{1}{3}\dots+\frac{1}{n}\right)$

$O(n\log n)$

Ex: 22

```
for(i=1 ; i<=n ; i++)          N
    for(j=1 ; j<=n ; j=j*2)    logN
        stmt;
```

⇒ $O(n\log n)$

Ex: 23

```
for(i=1 ; i<=n ; i++)          N
    for(j=1 ; j<=i ; j=j*2)    log i
        stmt;
```

⇒ $\log_1+\log_2+\log_3\dots+\log n$

$\log(1\cdot2\cdot3\cdot4\cdot5\dots n)$

$\log(n!)$

$O(n\log n)$

Teacher's Signature _____

Notes

Ex:24

```
for(i=1; i<=n; i*=2)        log n
    for(j=1; j<=n; j++)     n
        stmt;
```

$\Rightarrow O(n \log n)$

Ex:25

```
for(i=1; i<=n; i*=2)        log n
    for(j=1; j<=i; j++)     i
        stmt;
```

$\Rightarrow 1 + 2 + 4 + 8 \ldots + n$

$\Rightarrow 2^0 + 2^1 + 2^2 + 2^3 \ldots + n$

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

Ex:26

```
for(i=2; i<=n; i=i^2)       } log log n
    for(j=1; j<=n; j++)     } n log n
        for(k=1; k<=n; k=k+j)
            stmt;
```

for inner to loop

$\frac{n}{1} + \frac{n}{2} + \frac{n}{3} \ldots + 1$

$O(n \log n \cdot \log\log n)$

Teacher's Signature _____

Notes

Ex: 27 for (int i=0 ; i<n; i=i*2)
stmt;

⇒ Infinite loop bcz i is always 0.

- **Best Case Time:** Minimum time for any input.

Ex: if element is 1st element during searching, then it is best case and for it time is $O(1)$, $\Theta(1)$ as $\Omega(1)$

Even if algo takes n time for one input & n^2 for all inputs then, best case T.C. is $\Theta(n)$

- **Worst Case T.C.:** Maximum time for any input.

Ex: if element is not there during searching, then it is worst case & for it, time is $\Theta(n)$ as $O(n)$ & $\Omega(n)$

Even if algo taken n^2 time for one i/p & n for all inputs then, worst case T.C. is $\Theta(n^2)$.

- **Avg. Case:** Some prob. measurment to calculate the avg. time taken. It can be $\Theta(n^2)$, $O(n^2)$, $\Omega(n^2)$, etc.

(Popular)
→ Best analysis is worst case analysis bcz it tells us how bad our algo can perform.

Teacher's Signature _____

Notes

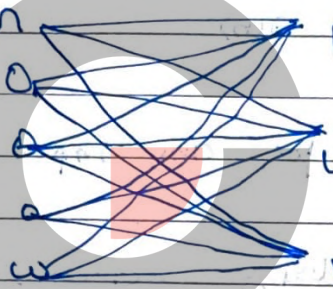
Date / /
Page No

→ Algo. has T.C. of $O(n^2)$ means algo takes $\leq n^2$ time in all cases. (best, worst, avg.)

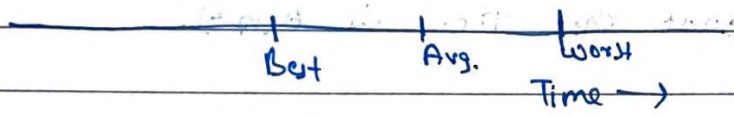
→ Algo is taking $O(n^2)$ in worst case means. algo takes $\leq n^2$ in worst case.

→ Major Miconception is that that Ω is for best case & O is for worst case. There is nothing like.

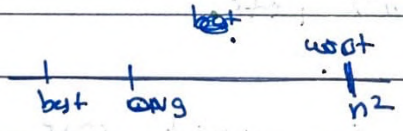
→ All 5 notations can be used to describe any case.
i.e.



→ Always, this order is there:



→ If $O(n^2)$ in worst case (then it is for all cases)



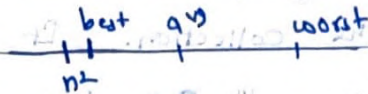
→ If $O(n^2)$ in best case

then avg & ~~worst~~ worst case can take more than n^2.

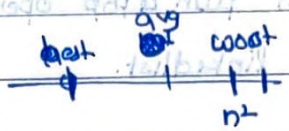
Teacher's Signature _____

→ Only thing we can say ~~ki~~ in best case min ~~has T.C. $\Omega(n)$ then in best case it's possible only.~~
Similarly, ~~in worst case, max time ki taken i.e. ki algo has T.C~~
Notes $O(n)$, then in worst

- $\Omega(n^2)$ in best case. (then $\Omega(n^2)$ for all cases)



- $\Omega(n^2)$ in worst case



how best & avg can be anywhere.

So. ~~ki~~ $O(n^2)$ in worst case then $O(n^2)$ in any case
~~ki~~ $\Omega(n^2)$ in best case then $\Omega(n^2)$ in any case.
~~ki~~ $O(n^2)$ in best case then can't say
~~ki~~ $\Omega(n^2)$ in worst case then can't say.

Que: Consider an algo A, which takes $\Theta(n)$ in best case & $\Theta(n^2)$ in worst case. Then which of the following are ~~true~~ True

~~a)~~ Algo T.C. is $O(n^2)$
~~b)~~ Algo T.C. is $\Omega(n)$
~~c)~~ Algo T.C. is $O(n^3)$
d) Algo. T.C. is $\Omega(n^2)$
~~e)~~ Algo T.C. is $\Omega(1)$
~~f)~~ Algo best case T.C. is $O(n)$
~~g)~~ Algo. worst case T.C. is $\Omega(n^2)$
~~h)~~ Algo. best case T.C. is $O(n^2)$

⇒ only d) is false

Notes

M-3 Stack and Queues

Date 06/07/23
Page No

→ **Abstract Data Types**: It specifies operations that can be performed on the collection. It does not talk (hides) the actual implementation. It can have multiple implementations.

Ex: Stack is ADT with push & pop operations. It can be implemented by array or linkedlist.

★ **Stack**: ADT with FILO or LIFO property. Insertion (pushing) and deletion (popping) happens at one end, another end is always closed.

Some of it's applications:

- Reversing word
- Undo Mechanism
- Expression Evaluation
- Function Calls
- Parenthesis Balancing

ISRO 2015-41

Ques: If the sequence of operations – push(1), push(2), pop, push(1), push(2), pop, pop, pop, push(2), pop are performed on a stack, the sequence of popped out values is:

⇒ 2 2 1 1 2

A)

	✗
✗	✓
✓	✗

⇒ **Stack Permutation**: A permutation that can be obtained from the stack. (No method, use intuition).

Teacher's Signature _____

Notes

Date / /
Page No.

Qu: Given sequence of numbers 1,2,3,4. Which of the following is not a valid stack permutation?

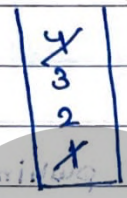
A. 1,2,3,4 ✓

B. 1,3,2,4 ✓

C. 1,4,2,3 ✗

D. 2,1,4,3 ✓

⇒



1 4 Now 2 cannot come before 3. So, invalid.

Qu: Which of the following can be obtained in the output using a stack assuming that input is the sequence 5,7,8,4,6 in that order?

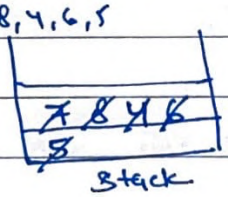
a) 6,8,4,7,5 ✗

b) 6,4,5,7,8 ✗

c) 6,4,7,8,5 ✗

d) 7,8,4,6,5 ✓

⇒ 7,8,4,6,5



→ Total no. of stack permutation for n elements is

$C_n = \frac{1}{n+1} {}^{2n}C_n$ (Catalan Number) $= \frac{(2n)!}{(n!)(n+1)!}$

Teacher's Signature _____

Notes

Date / /
Page No

★ Array based implementation of Stack:

We are following the following approach:

→ Top always points to top element, we need to increase it before adding or decrease it after deleting.

→ If stack is empty top = -1.

→ We will check if stack is full before pushing and if stack is empty before popping.

top = -1

```
void push (value)
{
    if (top == capacity - 1) {
        print ("stack full");
        return;
    }
    top = top + 1;
    array [top] = value;
}
```

} array [++top] = value;

```
int pop()
{
    if (top == -1) {
        print ("stack empty");
        return;
    }
    top = top - 1;
    return array[top+1];
}
```

} return array[top--];

Teacher's Signature _____

Notes

Date
Page No.

→ In this implementation, as we are just playing with value of top with each insertion and deletion, T.C. for push & pop is O(1).

→ Limitation of array based implementation is that capacity of stack need to be defined first & it cannot be changed.

★ Linked list based implementation of stack:

Here, we have two options i.e. either both operation from head (front) or tail (back).

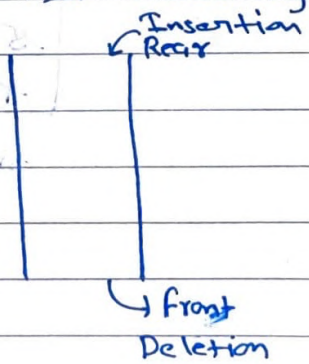
It is efficient to do from front as we have head pointer and we can easily add & remove from start of linked list in $\theta(1)$ time. If we do other implementation then T.C. would be $O(n)$.

✓ Code can be easily written. So skipping.

★ Queue: ADT with FIFO or LILO property. In this element is inserted from one end (rear) and removed from other end (front)

Applications can be:

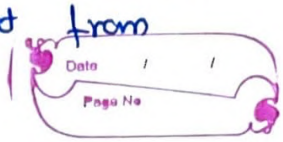
- In OS
- In networking
- Bank Queue, etc.



Teacher's Signature

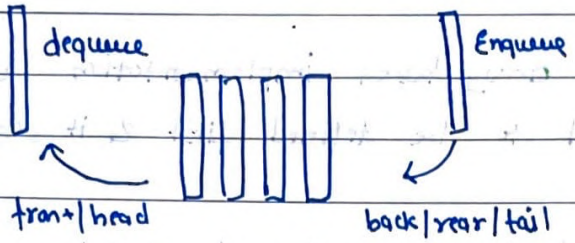
Insertion & Deletion analogy can be taken as go in bus from rear (piche se) and come out from front (aage se).

Notes



There are mainly two operation in queue:

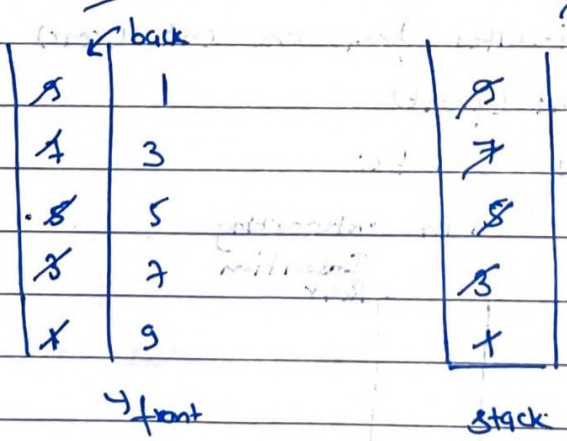
- Enqueue (data): Insert data at rear of queue
- int dequeue: Remove data from the front of queue



→ In this if char go as A,B,C,D, then they will come as it is i.e A,B,C,D. But in case of stack they will come in reverse i.e. D,C,B,A.

Ques: Given a 5 element queue (front to back: 1,3,5,7,9). & an empty stack. Remove elements from queue and insert them to stack and then remove them one-by-one from stack and reinstert them into queue. The queue now looks like: (from front to back)

⇒ 9,7,5,3,1



Teacher's Signature _____

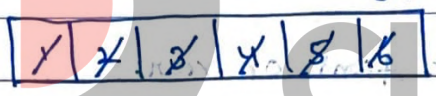
Notes

→ So, from the previous question we can infer that we can reverse element of queue using one stack.

★ Array based implementation of queue: We will have two approaches for array based implementation:

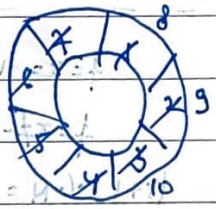
Before that, let's look at an issue. If we directly use array and after some enque and deque, we cannot use queue, even if it is empty because we reach till end of array.

Ex: front rear



Now what??

So, to overcome this, we can come up of solution that we will treat array as circular array by using mod n (%n).

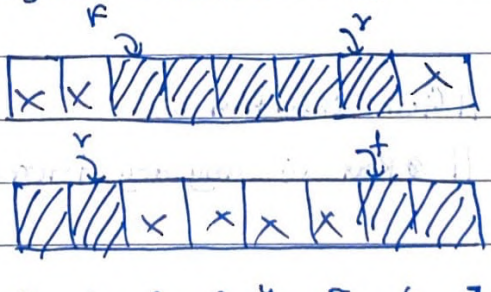


We will pretend that array is circular.

No issue in this.

→ Elements are always b/w Front to rear. Not b/w rear to front.

Ex: (Highlighted elements only)



Teacher's Signature _____

Notes

Date / /
Page No

Approach 1:

→ Front & rear will point to the front and rear element so we will increase rear before insertion and ~~decrease~~ increase front after deletion.

→ Initial values of front & rear are -1. i.e. queue is empty.

→ Elements are b/w front & rear.

→ front or rear never crosses each other.

→ we never decrement front or rear.

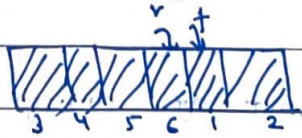

```
#define N 15
int front, rear, data;
front = rear = -1;
int array[N];
```

Initialisation.

f=r=-1 → queue is empty

f=r≠-1 → one element.

(r+1)%N = f → queue is full i.e. rear is just behind front.




```
void enqueue (data)
{
  if (rear+1%N == front) {
      print ("Queue is full");
      return;
  }
  if (front == -1);      // first Insertion
      front = 0          // @Rear is anyway increased

  rear = (rear+1) % N;
  Array [rear] = data;
}
```

Teacher's Signature _____

Notes

Date / /
Page No.

So two things we take care of during:

a) Enqueue:
- if queue is full
- if first insertion

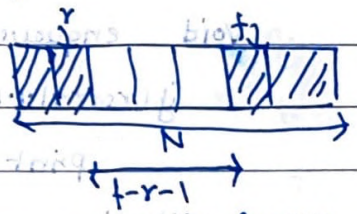
b) dequeue:
- if queue is empty.
- if last element in now queue will be empty.

```
int dequeue(){
    if (front == -1) {
        print("queue is empty");
        return -1;
    }
    data = array[front];
    if (front == rear)          // Last Element.
        front = rear = -1;
    else
        front = (front+1) % N;
    return data;
}
```

```
int getsize(){                  // returns number of elements
    if (front == -1)
        return 0;
    else if (front > rear)
        return N - front + rear + 1;
    else
        return rear - front + 1;
}
```

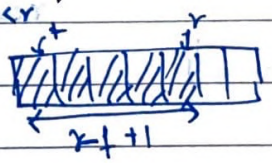
↳ condition of one element covered here.

if f > r



filled = N - (f-r-1) i.e. empty.

if f < r



if f = r = -1 ;
0

Teacher's Signature

if f = r ;
1

Notes

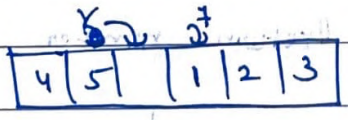
★ Approach-2

→ In this implementation, we start with front = start = 0 instead of -1. This means we already point to the place where we need to insert.

→ In this implementation, our old methods to check full or empty queue will not work because here f=r=0 can be empty or it can be full also. However f=r≠0 is always full. but f=r=0 can be full or empty. So, then how do we know if it is full or empty.

→ To avoid the issue we will stop one cell before and waste this one cell. we can't insert in one cell. Now if only one cell remain then it is full i.e. (r+1) mod n == f and if f=r, then it is empty.

→ full



→ full

```
void enqueue(data){
    if(rear+1%N == front){
        print("full queue");
    else:
        array[rear] = data;        // Note that first insert then incr.
        rear = (rear + 1) %N;
}
```


Teacher's Signature _____

Notes

Date / /
Page No.

```
int dequeue() {
    if (front == rear) {
        print("Empty Queue");
        return;
    }
    else {
        data = array[front];
        front = (front + 1) % N;
    }
    return data;
}
```

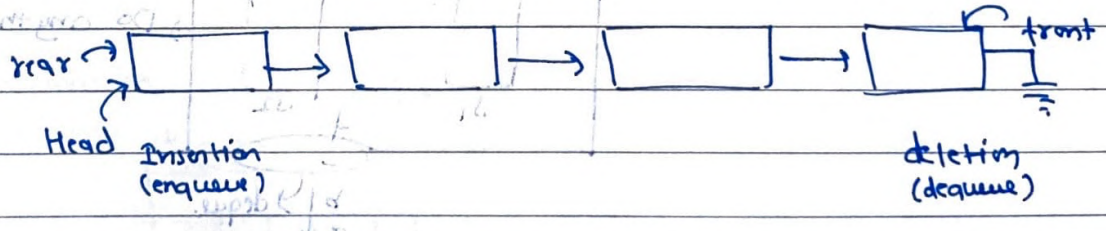
→ Here one cell is wasted i.e. actual capacity is N-1 only. But it is worth it.

→ Earlier approach is much more popular as compared to this.

★ **Linked list implementation of queue:**

Here also we have two options:

→ **Option 1:** Insertion from front & deletion from back of LL

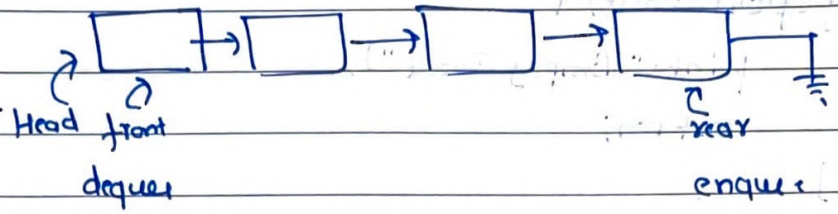


This is not a good option because insertion will be done in O(1) time but dequeue will take O(n) time as we will have to traverse till end.

Teacher's Signature _____

Notes

→ Option 2: Insertion from last of LL & deletion from starting.

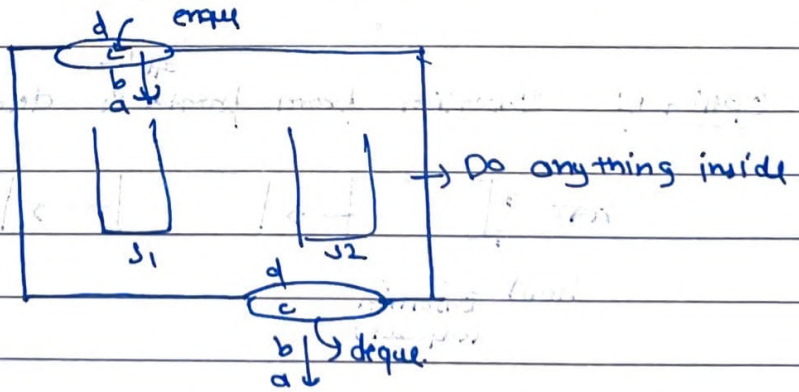


This is best option as we can perform both deque and enque in O(1) time. enqu in O(1) as:

```
rear → next = new;
rear = new;
// done
```

→ front = rear = null → Empty queue.
→ front = rear → One Element
→ No size full issue in LL implementation.

★ Implementing queue using two stacks: we need to perform basically enque and dequeue operations using push & pop of two stacks.



This is what we need to do.

Teacher's Signature _____

Notes

Date / /
Page No.

Solution:

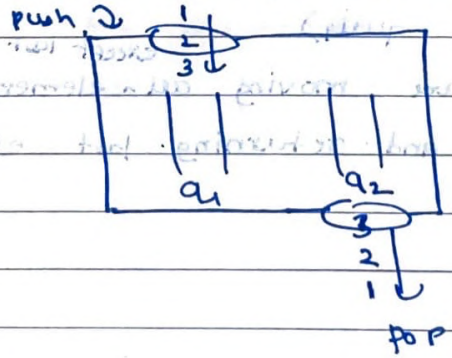
```
Enque (data) {
    push (S1, data);
}

int dequeue () {
    if S2 is not empty {
        data = pop (S2);
        return data;
    }
    else {                    // S2 is empty
        Transfer all elements from S1 to S2
        data = pop (S2);
        return data;
    }
}
```

What basically we are doing is in enqueue, ~~popping~~ pushing in S1 and in dequeue (first) moving all data to S2 & popping from S2 then in later dequeues just popping from S2 till S2 is empty.

Total Cost for m enque & n deque is O(m+n).

☆ **Implementing stack using two queues:** Here, we need to perform push & pop operation using enque & deque on two queues.



Teacher's Signature _____

Notes

Solution:

```
void push (data) {
    // insert into any non-empty queue
    if q1 is notempty:
        Enqueue (q1, data);
    else:
        Enqu (q2, data);
}

int pop() {
    if q1 is not empty {
        Transfer n-1 elements (all except last) to q2;
        data = dequeue (q1); // last element of q1
        return data;
    }
    else {
        Transfer n-1 elements (all except last) to q1;
        data = dequeue (q2); // last element of q2
        return data;
    }
}
```

We are for pushing, just enquing to the non empty queue (if first push then any queue)
And for popping, we are moving all except last elements from filled queue to empty queue and returning last element.

Teacher's Signature _____

Notes

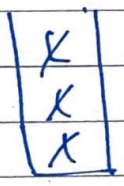
Date / /
Page No.

★ **Balancing Symbols:** Ex: [()] is ✓ but [(]) is X.

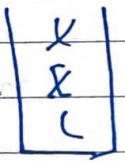
Algorithm for this:

```
Create a stack.
while (input is available)
  {
    if i/p is opening delimiter like (, {, [
        push i/p to stack.
    if i/p is closing symbol like ), }, ]
        pop the stack
        if stack is empty, then error
        if popped symbol is not the corresponding
        delimeter, then error.
  }
if stack is not empty, then error
```

Ex: ((())) ([()])



Valid

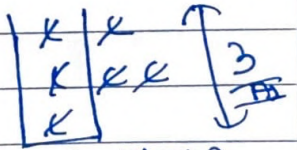


→ Not corresponding.

Invalid

Que: Consider the usual algorithm for checking validity of paranthesis. What is the max. no. of paranthesis that will appear on the stack at any one time when the algorithm analyses: (()(())(())).

a) 4 b) 3 c) 2 d) 6



(It is asking depth of stack)

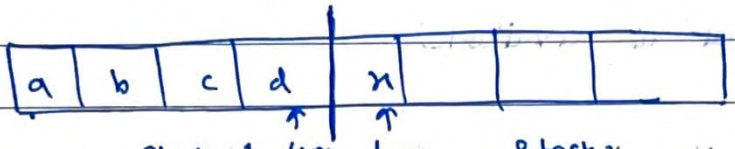
Teacher's Signature _____

Notes

Date / /
Page No.

☆ Implementing 2 Stacks using one array:

a) Divide in two halves: Not efficient Ex: one stack has 4 elc. & one has 1 element and total array is of 8, then too we can't add in first stack.



b) Grow stack by your own need: Efficient (Space wise)

No Hard boundary. (Grow in any direction acc to your need)

Time wise both are equally efficient i.e O(1), but Space wise second approach is more efficient.

☆ Ways of writing an expression:

- Infix Notation: x+y
- Prefix Notation / Polish Notation: +xy
- Postfix Notation / Reverse Polish Notation: xy+

Teacher's Signature _____

Notes

Date / /
Page No.

Precedence :	↑	(−, +) unary	>	(↑)	>	(*, /)	>	(+, −) binary
Associativity :		LR		RL		LR		LR

- **Infix to Postfix:**

→ **Simple human algorithm:**

Step 1: Apply appropriate brackets based on precedence and associativity

Ex: x+y*z → (x+(y*z))

Step 2: Convert each inner bracket to postfix separately, treat this as one box. & repeat the process.

Ex: (x+(y*z)) → (x+(yz*)) → xyz*+ Postfix.

Ex: a) a+b*c-d/e*f

→ a+(b*c)-((d/e)*f)

a+(bc*)-((de/)*f)

((a+(bc*))-(de/f*))

(abc*+)-(de/f*)

abc*+de/f*-

b) (a+b*c-d)/(e*f)

→ (a+bc*-d)/(ef*)

→ ((abc*+)-d)/(ef*)

→ (abc*+d-)/(ef*)

→ abc*+d-ef*/

- **Infix to prefix:** Similarly we can do from infix to prefix.

Ex: a+b*c-d/e*f

((a+(b*c))-((d/e)*f))

⇒ (a+(*bc))-((/de)*f)

→ (+a*bc)-(*/def)

⇒ -+a*bc*/def.

Teacher's Signature _____

Notes

Date / / Page No

Que: Convert following infix notation to prefix & postfix

a+b*c*d↑e↑f-g/h+a*b/c

a) Postfix:

a+(b*c)*(d↑(e↑f))-(g/h)+((a*b)/c)

⇒ a+(bc*)*(def↑↑)-(gh/)+(ab*)/c)

⇒ a+(bc*def↑↑*)-(gh/)+(ab*c/)

⇒ (abc*def↑↑**)-(gh/ab*c/+)

⇒ abc*def↑↑**+gh/ab*c/+

b) Prefix:

(a)+((*bc)*(d↑(↑ef))-(/gh)+(*ab)/c)

a+((*bc)*(↑d↑ef))-(/gh)+(/*abc)

a+(**bc↑d↑ef)-(/gh)+(/*abc)

+-+a**bc↑d↑ef/gh/*abc

Ex: 7*-5+3

↓

Here this is unary minus having highest priority.

(7*(-5))+3

Postfix: (7*(5-))+3

⇒ (75-*)+3

⇒ 75-*3+

prefix: (7*(-5))+3

(*7-5)+3

+*7-53

Ex: +(a,b,c)

⇒ prefix:

+(,ab,c)

+(,abc)

+,a,bc

postfix:

+(ab,,c)

+(ab,c,)

ab,c,+

Teacher's Signature _____

Notes

Ex Sin(x+y) *z

Sin(xy+) *z

xy+Sin *z

xy+Sinz*

postfix.

Sin(+xy) *2

Sin +xy *2

* Sin +xy 2.

prefix.

* Infix to postfix using stack

Algo:

if i/p is '(':

push in stack

if i/p is ')':

pop until left paranthesis is popped.

if i/p is operator):

1: priority of i/p is less than stack top, then pop.

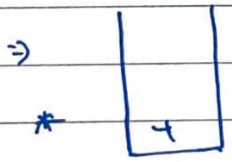
2: " " " " higher " " " , then push

3: " " " " same as " " , then pop (except '^')

(Remember gareeb ke upar ameer baithega)

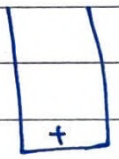
Case i)

i/p: +



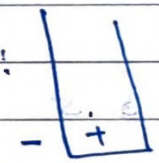
Case ii)

i/p: *



Case iii)

i/p: -



Teacher's Signature _____

Notes

Ex: 3+2*5-7 ↑ 9↑ 4

325*+79 4↑↑ -

→ Evaluating postfix using stack

if i/p is binary operator :

//pop 2 elements

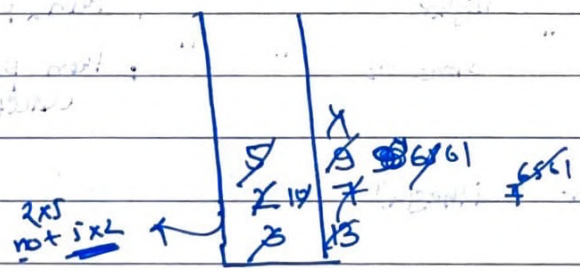
op2 : pop()

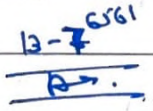
op1 : pop()

res = op1 <> op2

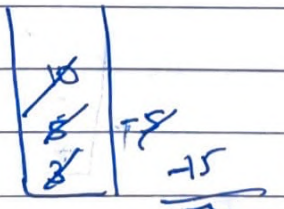
push result

Ex: 3 2 5 * + 7 9 4 ↑ ↑ -





Ex:2 3 5 10 - *

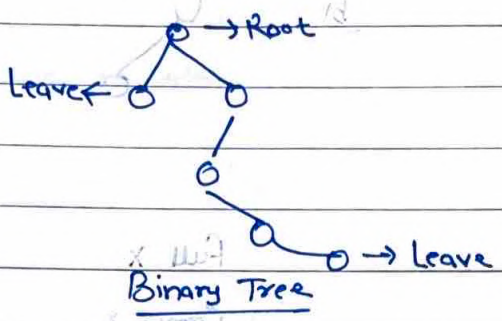


Notes

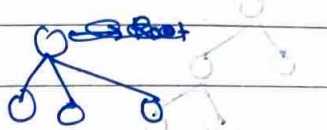
Date / /
Page No.

Module-04 Binary Trees (BST & AVL)

Binary → 0, 1, 2 children



Binary Tree

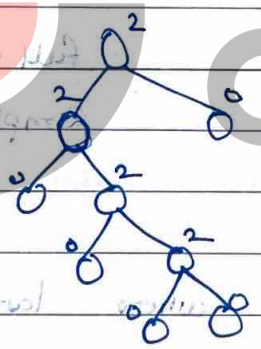


Not Binary Tree

⇒ Some Standard binary trees:

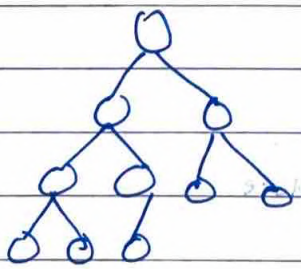
★ **Full Binary Tree:** All nodes have either 0 or 2 child.

Ex:

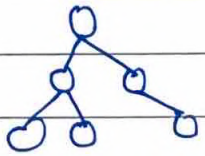


★ **Complete Binary Tree:** All levels except last are full. Last level is left filled

Ex:



Left filled



Not CBT

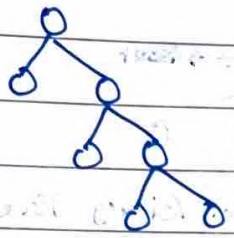
Teacher's Signature _____

Notes

Date / /
Page No

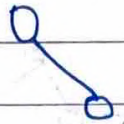
Ques: Mark which one is full or complete or both or none.

a)



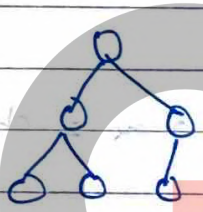
full ✓
Comp ✗

b)



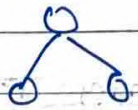
full ✗
Comp ✗

c)



full ✗
Complete ✓

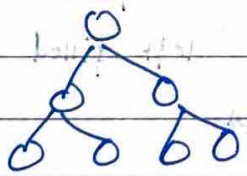
d)



full ✓
Complete ✓

★ **Perfect Binary Tree:** C.B.T where last level is also full.

Ex:

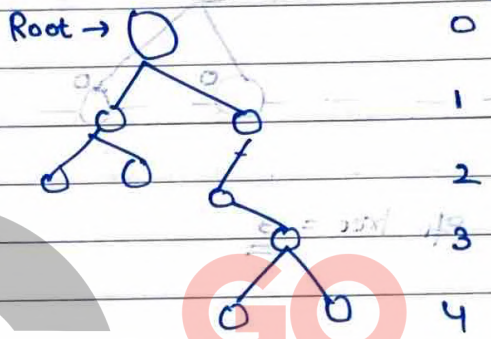


Perfect ⇄ full + Complete

Notes

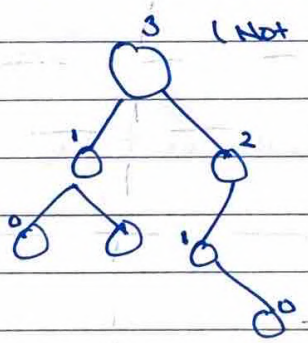
Date / /
Page No.

★ **Depth of a node:** The depth (or level) of a node is the number of edges from the node to the root. Root node's level is 0 by default, Untill stated otherwise.



Depth → Looking down (kitna gehra hai)

★ **Height of a node:** The height of the node is the number of edges (in longest path) from the node to the leaf. Height of leaf node is 0.



(Not 2 as longest path).

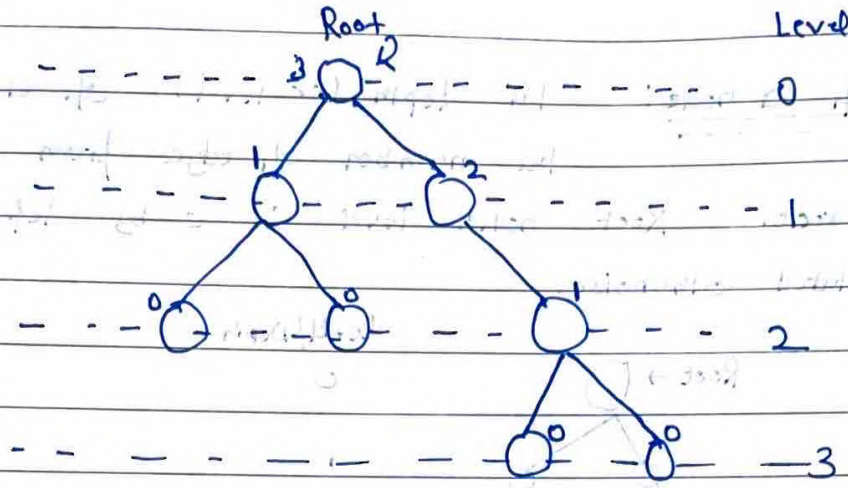
Height → looking upwards (kitna upar hai)

★ **Height of a tree:** It is equal to height of root node

(3 in above ex)

Teacher's Signature _____

Notes

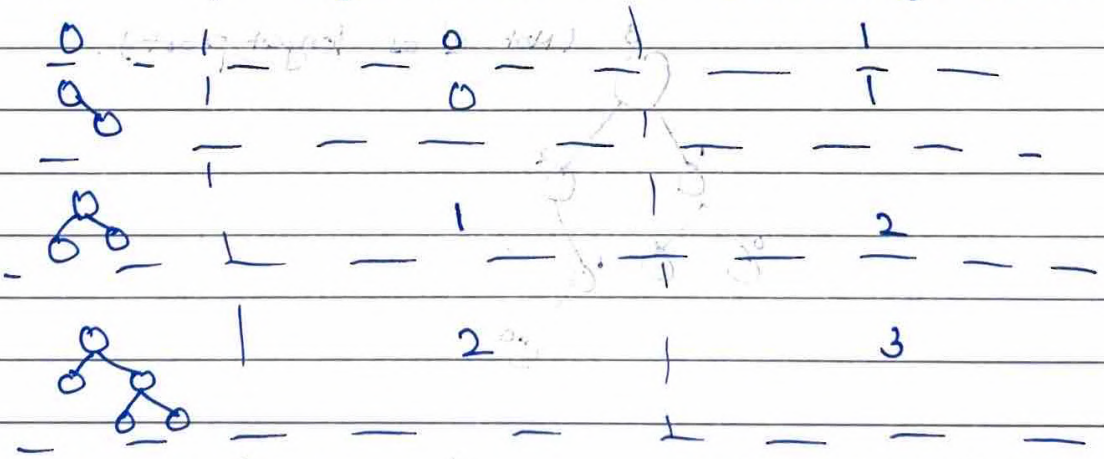


Height of tree = 3

★ In a binary tree, no. of leaf nodes is always equal to one more than nodes with two children.

No. of leaves = 1 + No. of degree 2 nodes

Ex: Tree | Degree 2 Nodes | Leaves = 1 + degree 2 nodes



Teacher's Signature _____

N → All nodes
L → Leaves
I → Internal Nodes
D_2 → Degree 2.
D_1 → Degree 1

Notes

Now, in a full binary tree (i.e. either 2 or 0 child), every internal node is a degree 2 node.

$I = D_2$

$\boxed{L = I + 1}$ → Only in full binary tree.

So, from this, we can say

Total nodes in full binary Tree (N) = $I + L$

$= D + I + 1$

$\boxed{= 2I + 1}$ ✓

Que: $I = 15$ in full BT, then $n = ?$

⇒ $n = 31$

Que: $L = 32$ in full BT, then $n = ?$

⇒ $n = I + L$

$n = L + (L-1)$

$n = 63$

Gate CSE 2015

Que: A binary tree has 20 leaves, then 2° nodes are ____ ?

⇒ 19

Que: Consider a perfect B.T. if root is at level 0, then no. of nodes at level i?

⇒ 2^i

Teacher's Signature

Height of Tree ≡ Depth of tree.

Notes

Date / /
Page No.

Que: Consider a perfect binary tree of height h, where root is at level 0.

a) Leaves are at level h.
b) No. of leaves are 2^h.
c) No. of I $2^h - 1$
d) Total no. of nodes $2^{h+1} - 1$.

Que: Consider of perfect binary tree of total n nodes.

a) No. of leaves is: $\frac{n+1}{2}$
b) Height: $(\log_2(n+1)) - 1$ or $\log(l)$

Que: Consider a complete B.T. of height h:

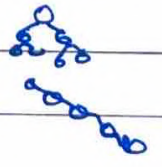
a) Min. no. of nodes: 2^h
b) max. no. of nodes: $2^{h+1} - 1$

Que: Consider a binary tree of height h:

a) Min. no. of nodes: $h+1$
b) max. no. of node: $2^{h+1} - 1$
c) Min. no. of leaves: 1
d) Max. no. of leaves: 2^h

Que: Consider a binary tree of n nodes:

a) Min height: $O(\log n)$
b) Max. height: $O(n)$



Teacher's Signature _____

Notes

Date / /
Page No.

☆ Representation of Trees:

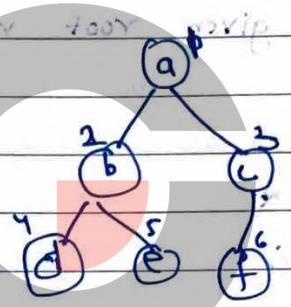
a) **Array Representation:** Suitable for CBTrees only.

At i, if there is some node, then (1 starting index)

parent : i/2

right child : 2i

left child : 2i+1



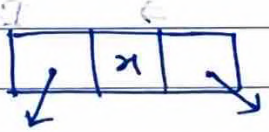
1	2	3	4	5	6
a	b	c	d	e	f

(We can find formulae for child & parent index even if index starts at zero. Do during Revision)

Next Time

b) **Linked list Representation:** More useful / More General

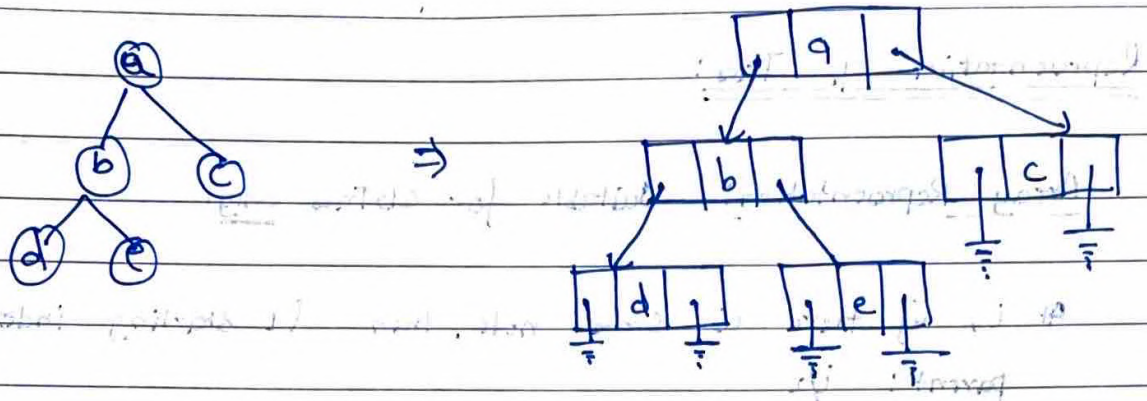
Similary to doubly LL.




```
struct BTnode {
    int data;
    struct BTnode *left;
    struct BTnode *right;
};
```

Teacher's Signature _____

Notes



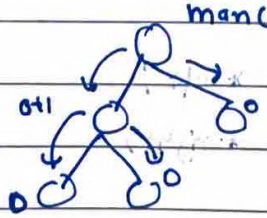
Linked list Representation

Que: What does fun() do for a given root node of non-empty binary tree?

```
int fun (struct node * root) {
    int a=0, b=0, ans;
    if (root == null)
        return 0;                          → leaf
    if (root→left == null && root→right == null)
        return 0;
    if (root→left)        → // means left != null
        a = fun (root→left);
    if (root→right)
        b = fun (root→right);
    return  max(a,b) + 1;
}
```

⇒ It is calculating height of tree.

max(0+1,0) + 1 = 2 (Height)



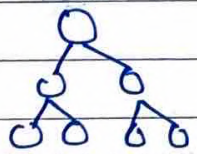
Teacher's Signature _____

Notes

Date / /
Page No.

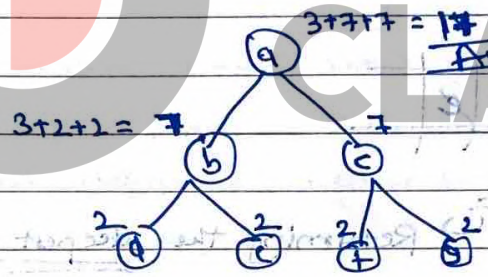
Ques: Below code is executed on the given tree. What will be the output?

```
int func (Node *t) {
    if (t==null) return 0;
    if (t->left==null && t->right==null) return 2;
    else return (3+ func(root->left) + func(root->right));
}
```



(Here assume at else, fun(left) is called before right, however uncertain in C programming).

⇒



Here function calls are a → b → d → e → c → f → g

Ques: Analyse the given function: (pg. 66)

```
struct node * checkdeep (struct node *root)
{
    struct node *temp;
    if (!root) return null;
    enqueue (root);
    while (! isEmptyqueue ()) {
        temp = dequeue ();
        if (temp->left)
            enqueue (temp->left);
```

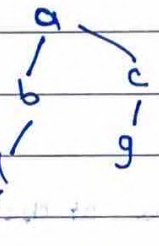
Teacher's Signature _____

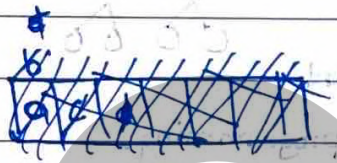
Notes

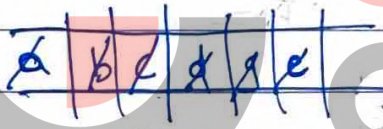
Date / /
Page No.

```
if (temp->right)
        enqueue (temp->right);
}
return temp;
}
```

⇒ Let's analyse on this tree







temp=c

↳ Returning the deepest node..

DFS

Qu:

```
int findlevel (root) {
    level=0;
    if(root==null) return 0;
    enqueue (root);
    enqueue (null);
    while (!isEmptyQueue()) {
        root= dequeue();
        if (root == null) {
            if (!isEmptyQueue()) Enqueue (null);
            level++;
        }
        else {
            if (root->left) Enqueue (root->left);
            if (root->right) enqueue (root->right);
        }
```

Teacher's Signature

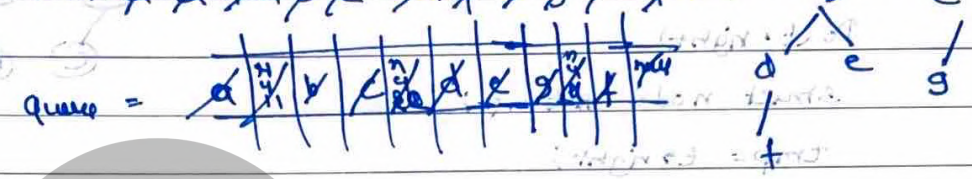
Notes

Date / /
Page No.

return level;
}

⇒ level = 0 1 2 3 4

root = a null b c null f g null null



So it will return 4 (i.e. level / Depth of the tree).

Que: checkSomething (root1, root2)
{
if (root1 == null && root2 == null)
return 1;
if (root1 == null || root2 == null)
return 0;

return (root1→data == root2→data && checkSomething(root1→left, root2→left) && checkSomething(root1→right, root2→right));
}

⇒ It is returning 1 if Tree 1 & Tree 2 are exactly same i.e. structure and the data. If we remove dotted line then it'll not compare data & will only compare structure.

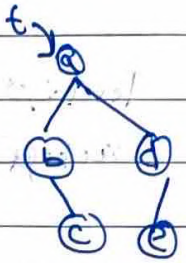
Teacher's Signature _____

Notes

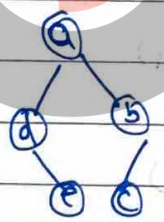
Date / /
Page No.

Ques: What does Do() do for a root node of given tree?

```
void Do(struct node *t)
{
    if(t)
    {
        Do(t->left);
        Do(t->right)
        struct node *tmp;
        tmp= t->right;
        t->right = t->left;
        t->left= tmp;
    }
}
```



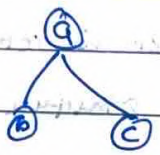
⇒



So it is making mirror img. of the given tree.

★ Void postorder (struct node * root)

```
{
    if(root)
    {
        postorder(root->left)
        postorder(root->right)
        print(root->data);
    }
}
```



b c a

Teacher's Signature _____

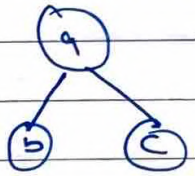
Post
LR Root

Pre
Root LR

In
L Root R

Notes

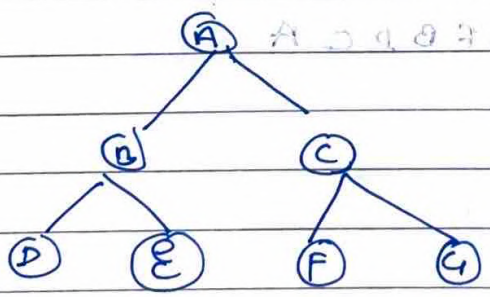
```
* void inorder ( struct node *root)
{
  if (root)
  {
    inorder ( root->left) ;
    print(root-> data) ;
    inorder (root -> right);
  }
}
```



b a c

```
* void preorder (struct node *root)
{
  if (root)
  {
    print (root -> data) ;
    preorder (root -> left) ;
    preorder (root -> right);
  }
}
```

a b c



IN : D B E A F C G

PRE : A B D E C F G

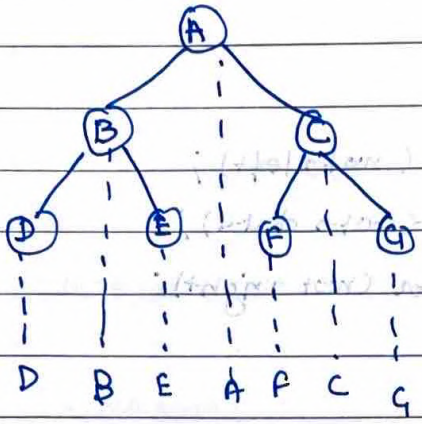
POST : D E B F G C A

Teacher's Signature

Notes

Date / /
Page No.

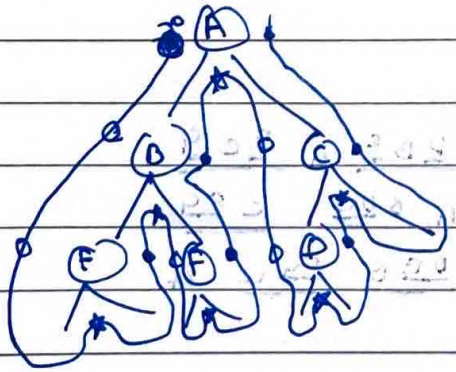
⇒ One Trick of Inorder Traversal: (Projection / Squishing)



(only for inorder).

So inorder Traversal is D B E A F C G

★ Traversal Trick: Go from root of tree from top to bottom & Left to Right & print the node for preorder while traversing first time, inorder " " II " , postorder " " III " .



Pre: A B F F C D
In: F B F A D C
Post F F B D C A

✓

Teacher's Signature _____

Notes

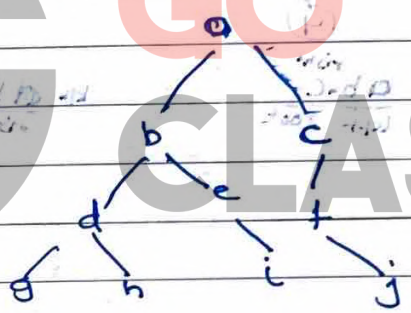
Date / /
Page No.

☆ Binary tree Construction (Given several Traversals):

a) Inorder & Preorder: A unique tree will be made in this case.

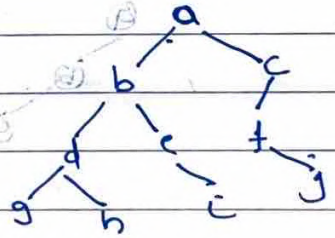
Ex: in: gdhbeiafjc
pre: abdeghecfj

From preorder get to know root & from inorder get to know left & right subtrees & go on recursively.



b) Inorder & Postorder: Unique binary tree here also.

Ex: In: gdhbeiafjc
Post: ghdiebjfca



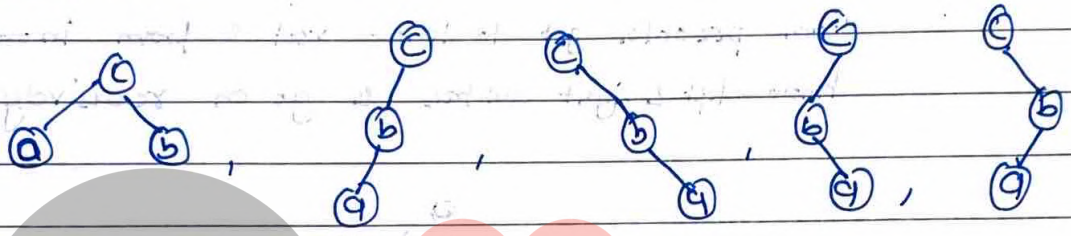
Teacher's Signature _____

Notes

c) Only Postorder or Preorder or inorder (any one): Catalan number of trees can be made exactly $\left[\frac{1}{n+1} {}^{2n}C_n\right]$ (Also eq. to stack permutation)

Ex: n=3 then 5 trees can be made.

Postorder: abc

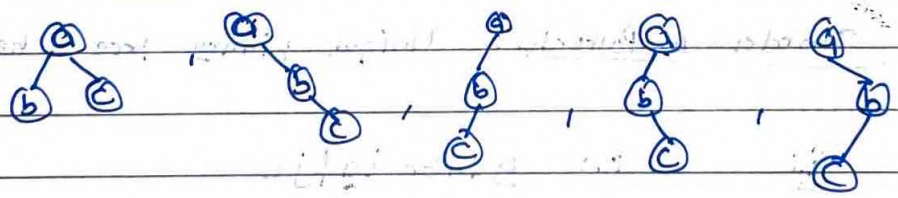


a b c — root, right

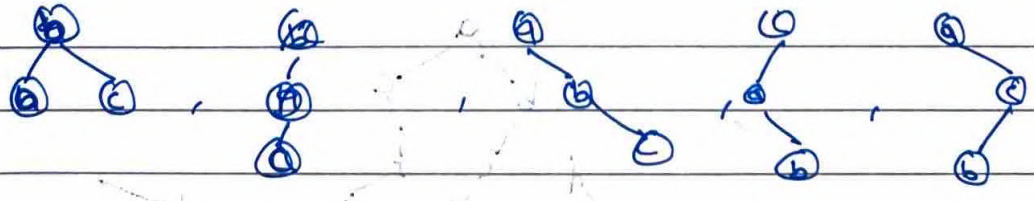
a b c — root, left

a b c — root, left

Preorder: abc



Inorder: abc



So, we can infer that, from any one only, all structures can be modified to structures as catalan no.

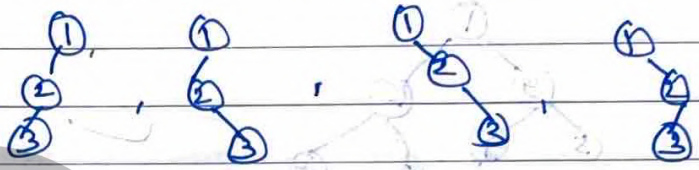
Teacher's Signature _____

Notes

Date / /
Page No.

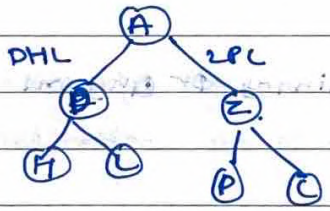
d) Using Pre order & Post order (both): May or may not form.

Ex: Pre: 1 2 3
Post: 3 2 1

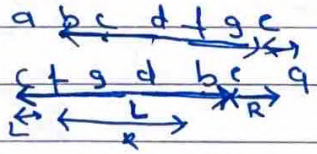


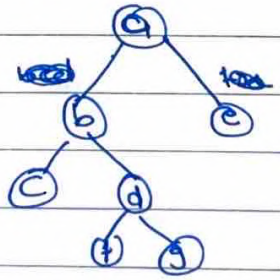
* c) But, given preorder & postorder we can form a full binary, unique tree.

Ex: Pre: A D H L Z P C
Post: H L D P C Z A



Ex 2: Pre: a b c d f g e
Post: c f g d b e a





Check for root in Post (RM) then divide post by checking LL(M) of in preorder & so on.

Teacher's Signature _____

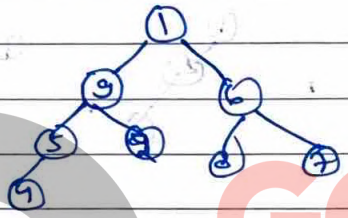
Notes

Date / / Page No

We can Construct a unique complete binary tree, using any one of the traversal only.

(complete binary tree → L to R Level wise)

Ex: pre: 1 9 5 4 2 6 8 7



So for a unique tree we need:

General Tree : Inorder + (Pre | Post)

Full Binary : General or Pre + Post (Any 2 of 3)

CBT : Full Binary ~~or~~ any one i.e. (any 1, 2 or 3)

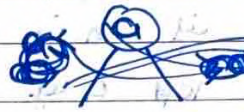
~~Thats~~

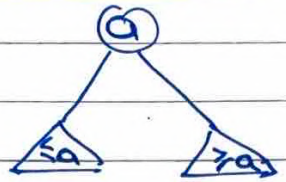
Teacher's Signature _____

Notes

Date / / Page No.

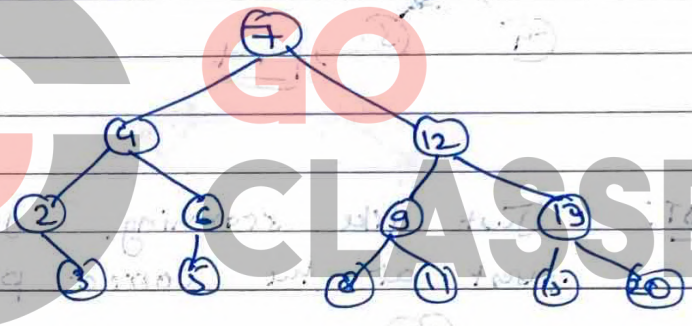
☆ **Binary Search Tree (BST):** In this, all the data of all node in left subtree of root should be ≤ than the data of root & in right subtree, it should be > data of root.



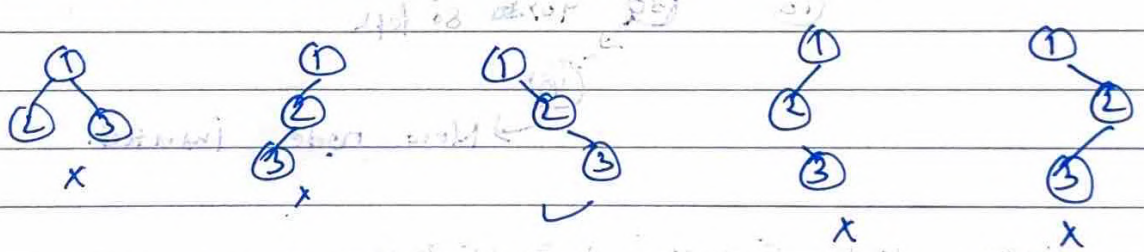


Smaller than a — Larger than a

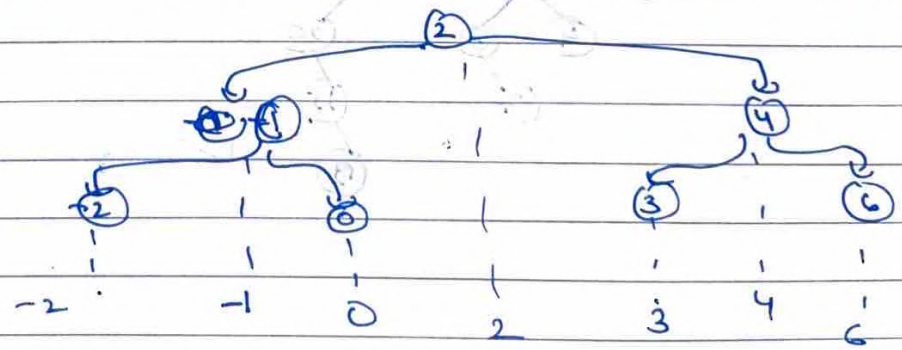
Ex:



We can check if given tree is BST or not by checking the above defination at each node.



⇒ Inorder traversal of BST in always sorted (increasing order)



→ Sorted

Teacher's Signature _____

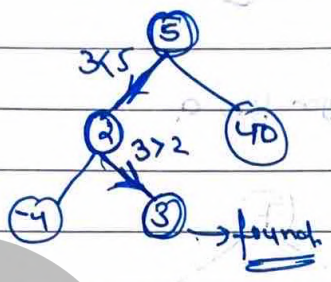
127 ⇒ Notes

Date / /
Page No

✱ Searching in BST:

Go from root, if n is greater then go right else if it is lesser go left & if equal then found. Repeat till you reach leaf.

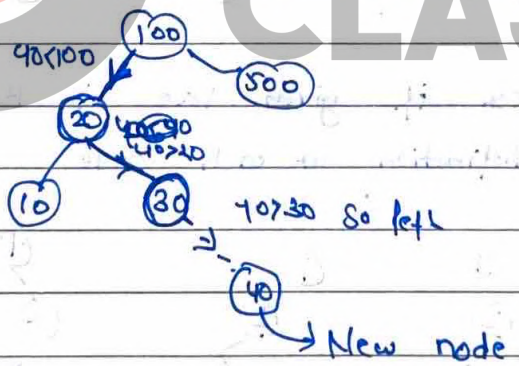
Ex: n=3



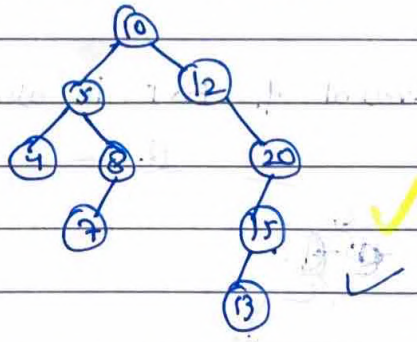
✱ Insertion in BST:

Just like searching, go left or right & insert at the correct place.

Ex: n=40



Ex:2 Insert 10, 12, 5, 4, 30, 8, 7, 15 & 13



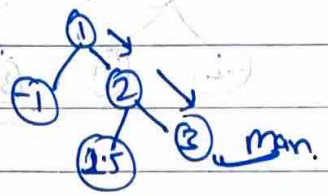
Teacher's Signature _____

Notes

Date / /
Page No.

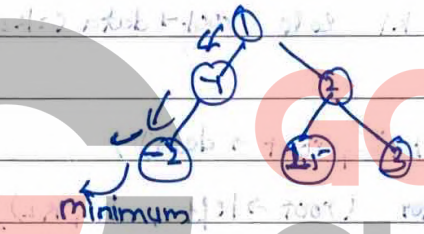
★ **Maximum Element in BST:** for this keep on going right of BST

Ex:



★ **Minimum Element in BST:** for this, keep on going left of BST.

Ex:



★ **k^{th} maximum or minimum:** for this do the inorder traversal ($\Theta(n)$) & then return k^{th} element for k^{th} minimum & $(n-k)^{th}$ element for k^{th} maximum ($\Theta(n)$).

Total T.C. = $\Theta(n) + \Theta(n)$

$= \Theta(n)$

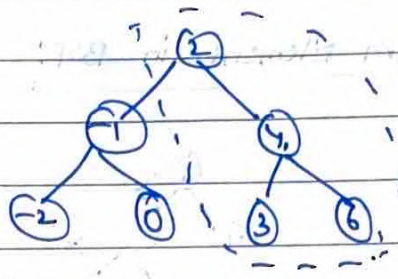
NOTE: In all (In, Pre, Post) order traversals T.C. is $\Theta(n)$ as we are visiting each node once.

★ **Range Search in BST:** It means finding all the nodes in the tree between a particular range.

Teacher's Signature _____

Notes

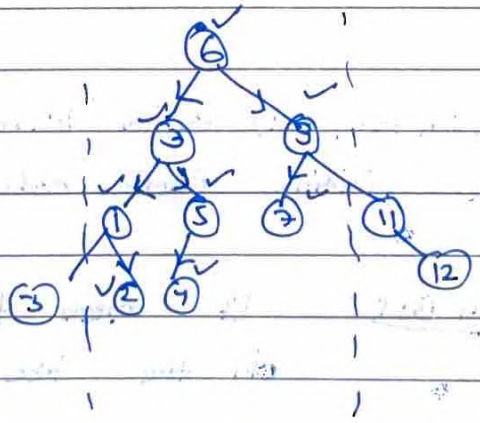
Ex: [1,9] is range & tree is



then o/p is 2,3,4,6.

```
void RangePrinter (node *root, int k1, int k2)
{
    if (root == null) return;
    if (root->data >= k1 && root->data <= k2)
    {
        printf("%d", root->data);
        RangePrinter (root->left, k1, k2);
        RangePrinter (root->right, k1, k2);
    }
    else if (root->data < k1)
        RangePrinter (root->right, k1, k2);
    else
        RangePrinter (root->left, k1, k2);
}
```

Ex: [1,10]



Teacher's Signature _____

Notes

Basically T.C. of this depends on how many nodes we find (let's say m), then T.C. = O(m)

In any tree of height h,
2h nodes (h+h) are on the border (may or may not be in range)

Total = 2h + m if how on border are in range or
m if all of border are in range

T.C. = O(2h+m)

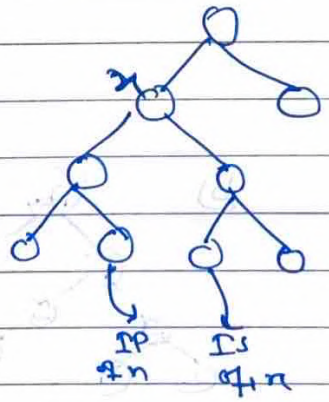
T.C. = O(h+m) ← Or. or T.C. = O((log n) + (m))

where h → height (eq. to log n)
m → no. of node in range.

Imp ★ Deletion in BST: This is not direct if node to be deleted is not leaf.

⇒ Inorder Predecessor: no. printed just before n is IPred. of n. = greatest no. in Left sub Tree.

⇒ Inorder Successor: no. printed just after n is ISuccessor of n = least no. in Right sub Tree.



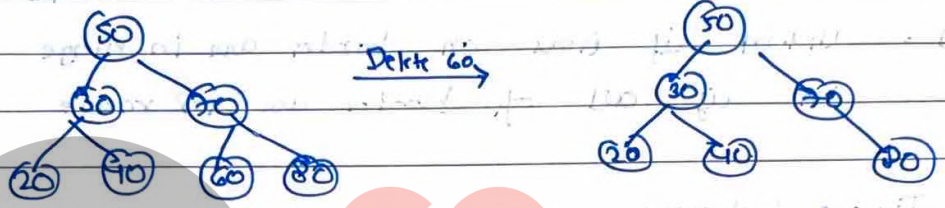
Teacher's Signature

Notes

Now, to delete node in BST, we first search the node in BST. Now, we have three cases:

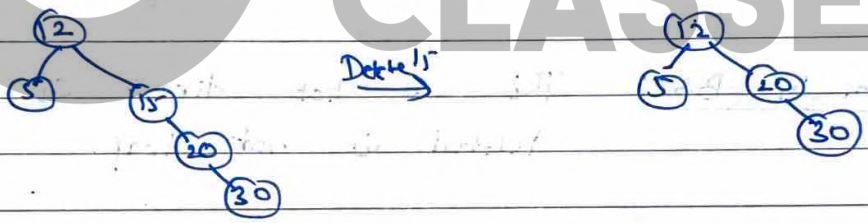
Case 1: Leaf Node → Just Delete it

Ex:



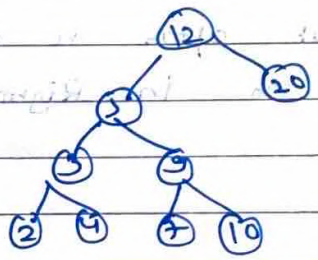
Case 2: One Child → Delete & connect child to parent

Ex:



Case 3: Two children → Replace node with LP or LS. & then the case will be change to (1) or (2)

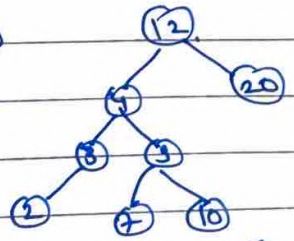
Ex:



Delete 5

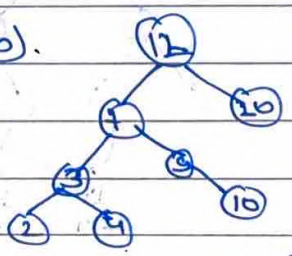
Now two options a) & b). Replace 5 with 4 or 7.

a)



or

b).



Teacher's Signature _____

Notes

Date / / Page No.

★ BST Time Complexities:

warmup:

- Best Case: Time to complete on optimal i/p.
- Worst " " " " pessimal "
- Avg. " : Run algo. many times, with many i/p & compute avg.

In data structures, while analysing best, worst or avg. case, there are two aspects:

- Internal state of the D.S. Ex (skewed or Balanced)
- Data Element on which operation is performed.
  Ex: (leaf or root in search)

Like Searching root element in BST
Balanced Tree : O(1)
" " " Skewed " : O(1)
" " leaf " " " : O(n)
" " " " Balance " : O(log n)

As we cannot commit that BST will be balanced, we will consider skewed as worst case.

So.

	Best Case	Avg. Case	Worst Case
Search	O(1)	O(log n)	O(n)
Insert	O(1)	O(log n)	O(n)
Delete	O(1)	O(log n)	O(n)

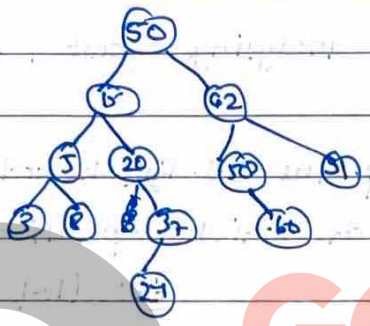
Teacher's Signature _____

Notes

Some BST PYQs:

CSE 96
Ques: A BST is generated by inserting following integers
50, 15, 62, 5, 20, 58, 91, 3, 8, 37, 60, 24.
No. of nodes in Left Sub Tree & Right Sub tree is:

⇒ Method 1 Lengthy.



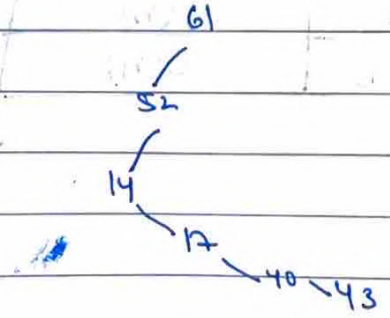
⇒ 7 in left subtree & 4 in Right Sub Tree.

→ Another approach can be counting no. less than 50 & greater than 50, bcz. All lessers will be in left subtree & all greater will be in right sub tree.

CSE 96
Ques: A BST is used to locate number 43. Which of the following prob sequence is possible.

(A) 61 52 14 17 40 43

⇒ One approach can be



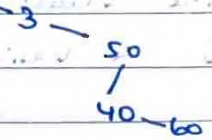
No issues possible.

Teacher's Signature _____

Notes

(c) 2 3 50 40 60 43

⇒



Cannot be on left of 50.
So, not possible.

(d) 10 65 31 48 37 43

⇒ Another approach ki ⓐ Divide seq. in two parts i.e. > 43 & < 43

(10, 31, 37,)
(65, 48)

Check if they are increasing & decreasing if yes then possible else not. So Yes, here possible.

(e) 81 61 52 14 41 43

⇒ (81, 61, 52,) ✓
(14, 41) ✓

(f) 17, 77, 27, 66, 18, 43

⇒ (17, 27, 18) ✗
(77, 66) ✓

So, not possible.

Note: If x are > no. & y are < no. then total probe sequences that are possible are

$$\frac{(x+y)!}{x!\,y!}$$

Eg: $\frac{5!}{2!\cdot 3!} = 10$

Easy from Comb. div. rule

Teacher's Signature

Notes

97 CSE

Que: A BST contains the value 1, 2, 3, 4, 5, 6, 7, 8. The tree is traversed in pre order and the values are printed out. Which of the following is a valid o/p?

(a) 5 3 1 2 4 7 8 6

⇒ Root Left Right is seq. in preorder.

(b) 5 3 1 2 6 4 8 7

(c) 5 3 2 4 1 6 7 8

(d) 5 3 1 2 4 7 6 8

⇒ 5 is root ie we know in BST 1234 are on left only as BST ie 6,7,8 are on right only

So option b not possible.

Now, 3 is root of LST now 1,2 are on its left in any order and 4 on right. So option C eliminated.

Now at RST, 7 is Root so 6 on left ie 8 on right. So 7,6,8 is correct. So D eliminated.

D

So, solve this recursively.

07 ET

Que: When searching for the key value 60 in a BST, nodes containing the key value 10, 30, 34, 40, 50, 70, 80, not necessarily in the order given. How many probe seq. are possible.

⇒ >60 : 2

<60 : 5

Prob seq: $5! \cdot 21 = \frac{7 \times 6}{2} \Rightarrow 21$

Teacher's Signature

Notes

2016 CSE

Qu: The no. of ways in which the no. 1,2,3,4,5,6,7 can be inserted in an empty BST, so that height is 6. (Height starts from 0).

⇒ As with each structure can have unique positions of no. to give resultant sequence. And total structures are 2^6. (bcz at each node we have two choices (other than root)).

So $2^6 \times 1 = 2^6$

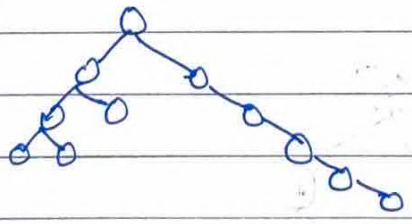
→ As we can observe that BST can be skewed (O(n) height), which may lead to operations inefficient. To overcome the problem we require a balance condition that ensures height is always O(log n). Now, for this we came up with many suggestions:

Suggestion 1: Right and Left subtree of root have equal no. of nodes

⇓

Problem: Still can be skewed.

Ex:

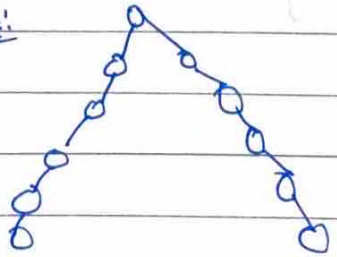


Suggestion 2: R & L ST of root have equal no. of nodes & height.

⇓

Problem: Still can be skewed.

Ex:



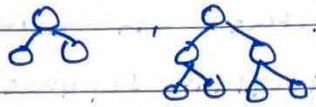
Teacher's Signature _____

Notes

Suggestion 3: Right & LST of every node have equal no. of node

⇓

Problem: Too strong, only perfect trees.

Ex: 

Suggestion 4: AVL Tree.

⇓

Problem: No Problem. Perfect Solution.

★★ AVL Tree: It is a self balancing BST. i.e. BST with balance condition.

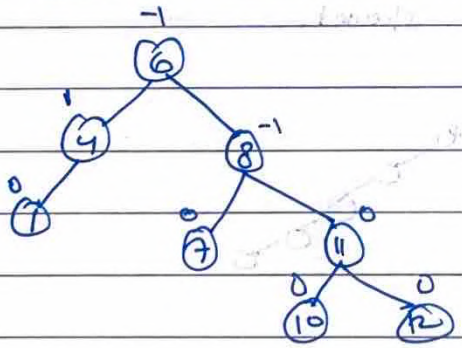
⇓

Balance Condⁿ: Balance of every node is b/w -1 & 1

i.e. balance ∈ {-1, 0, 1}

when balance(node) = height(node→left) − height(node→right)

Ex:



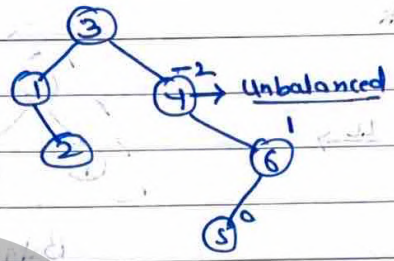
AVL → BST

BST ↛ AVL

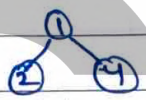
Notes

Date / /
Page No.

Qu: Which are BST & AVL:

a) 

BST ✓
AVL ✗

b) 

BST ✗
AVL ✗

```
Struct BinaryTreeNode {
    Struct BinaryTreeNode *left;
    Struct BinaryTreeNode *right;
    int data;
}
```

```
Struct AVLTreeNode {
    Struct AVLTreeNode *left;
    Struct AVLTreeNode *right;
    int data;
    int height;        — Additional
}
```


★ AVL Tree Operations:

a) AVL Search : Same as BST Search

b) AVL Insert : BST Insert + Balancing

c) AVL Delete :

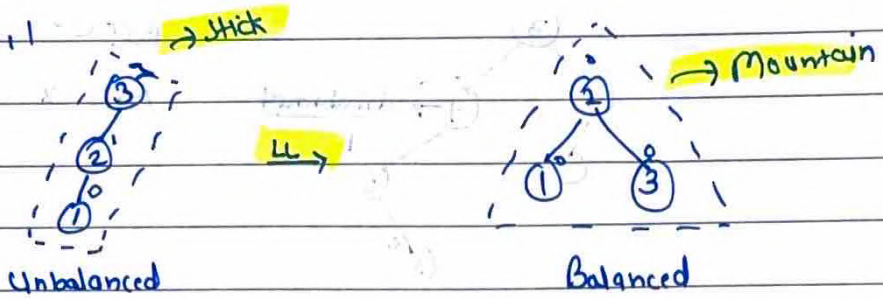
Teacher's Signature _____

Notes

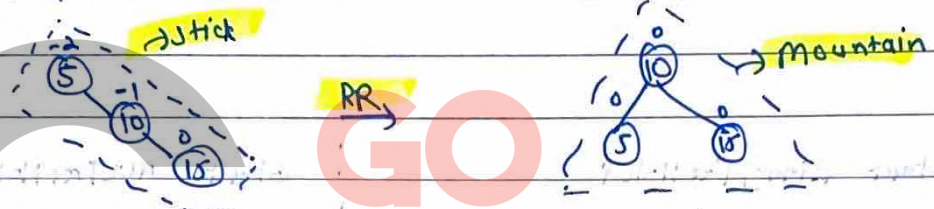
Date / /
Page No

★ AVL Insertion:

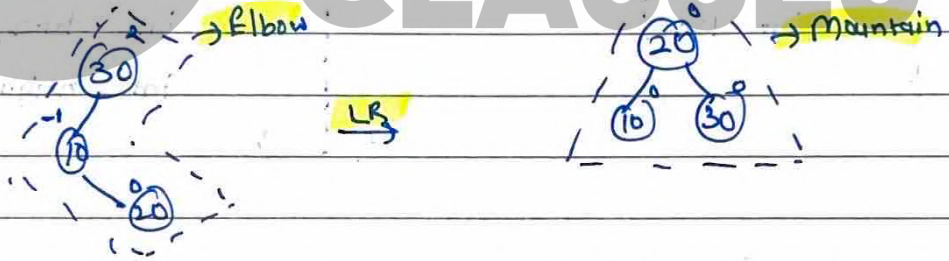
a) Insert 3, 2, 1



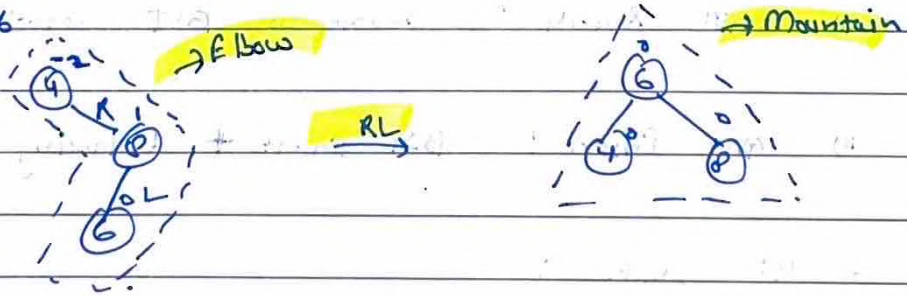
b) Insert 5, 10, 15



c) Insert 30, 10, 20



d) Insert 4, 8, 6

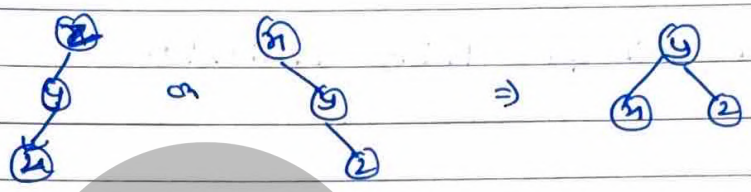


Teacher's Signature _____

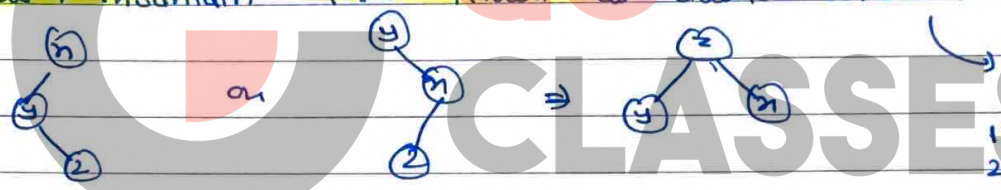
Notes

So these 4 are the std. disbalancing conditions, which we need to take care of during insertions. We have to create Mountain.

⇒ Stick → mountain (Single Rotation)



⇒ Elbow → mountain (Also known as double rotation)

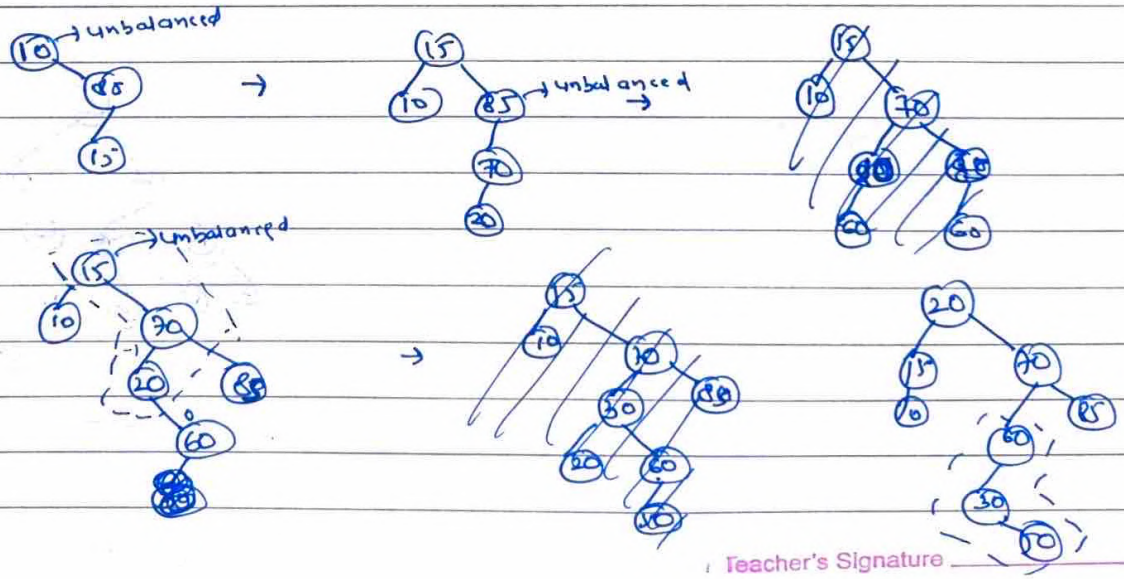


↳ bcz generally done in two parts
1: elbow → stick
2: stick → mountain

These rotations are just simple pointer manipulations which takes constant time.

Ex: Insert in order

10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55



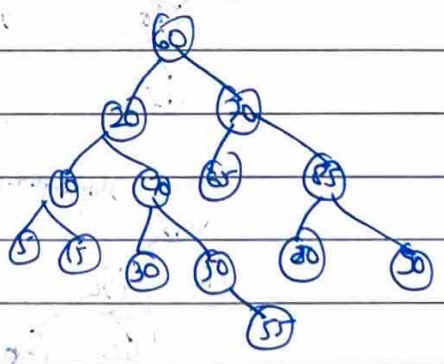
Teacher's Signature

Notes

Date / /
Page No.

→ We start checking BF (balancing factor) from bottom and do rotation at the very first encountered Unbalanced node. Then we balance that node and then we do not need to recheck it till next insertion bcz it will be balanced till root automatically

Continue the tree on this pg. during revision :)



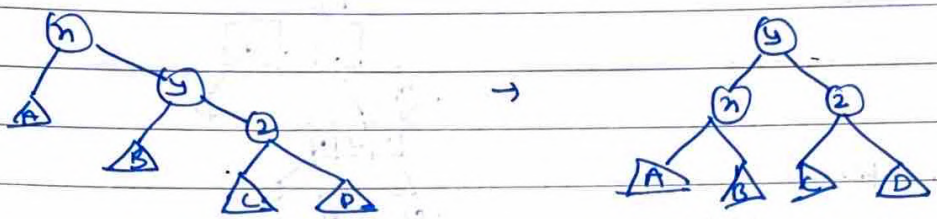
Teacher's Signature _____

Notes

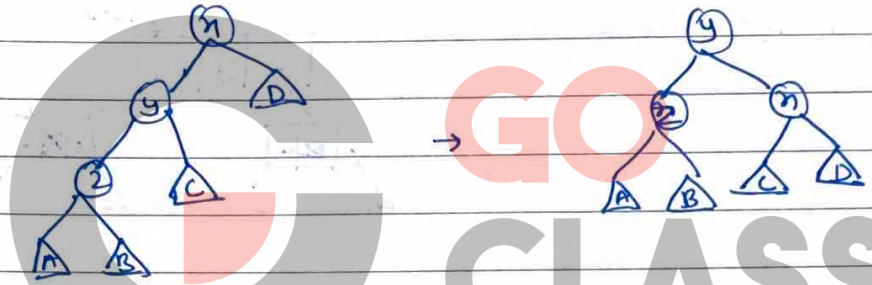
Date / /
Page No.

→ Situations we encounter: → Soln we go to it is balanced

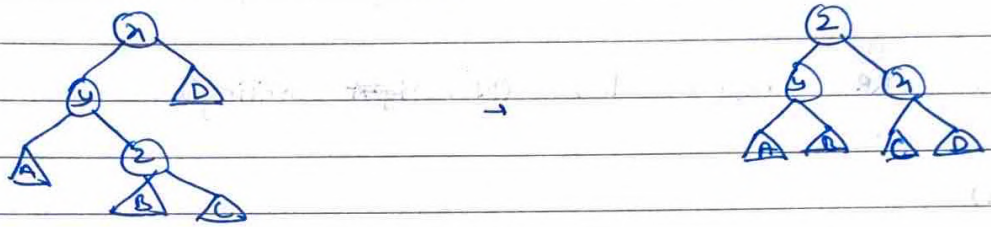
a)



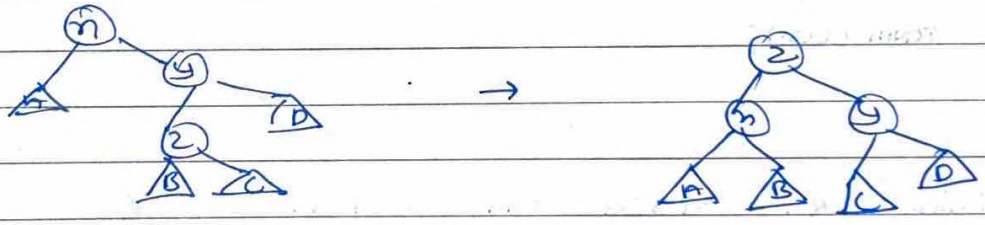
b)



c)



d)



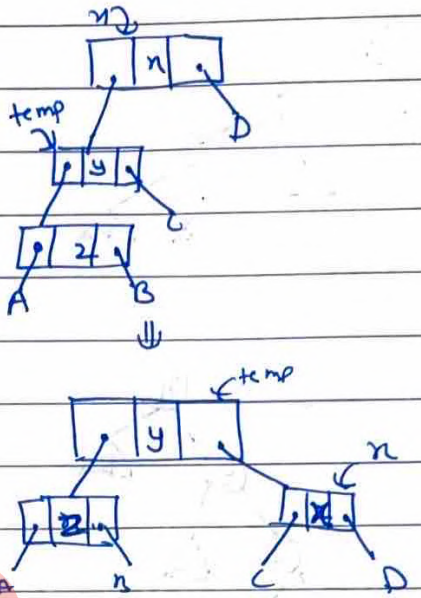
So, in case of insertion we have atmost one (single or double) rotation ⇒ O(1) rotations.

Teacher's Signature _____

Notes

⇒ Code of LL Rotation may look like:

```
LL (x)
{
    temp = n->left;
    n->left = temp->right;
    temp->right = n;
    return temp;
}
```



Also we may need to update the heights of temp & n in structure as we are maintaining this.

⇒ Similarly RR may be done (Not ~~right~~ writing).

⇒ LR (n)

```
{
    n->left = RR (y);   // n->left
    return LL(n);
}
```

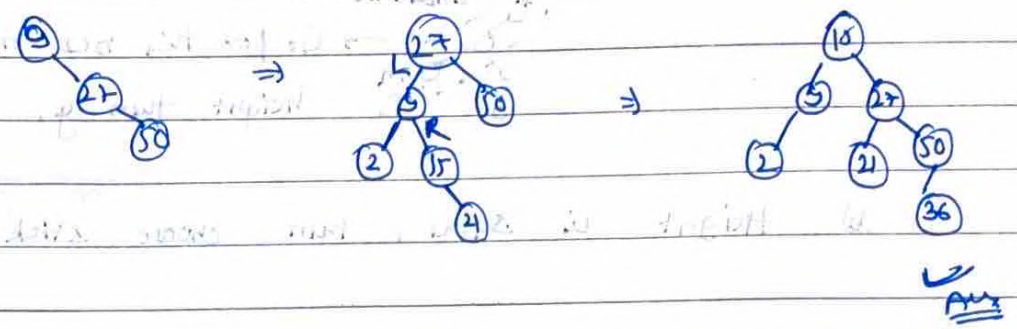
⇒ Similarly RL rotation can be done.

Notes

Date / /
Page No.

Spiggo

Que: Make AVL tree by inserting 9, 27, 50, 15, 2, 21, 36.



So, now we know about insertion. Now lets move on to deletion.

★ AVL Deletion: Delete like BST (3 cases) & then do balancing.

Ex: delete (7) from below AVL Tree:

Important points about deletion:

Rule 2
a) We may need to do balancing till root, i.e. at each level, after single deletion. So, while insertion we cannot guarantee in this that after first balancing whole tree is balanced. We need to chek till root.

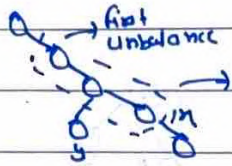
Rule 1
b) During balancing, start checking from bottom & balance first imbalanced node. Now there can be two cases:

Teacher's Signature _____

Notes

a) Height is not same, then go for higher height.

Ex:

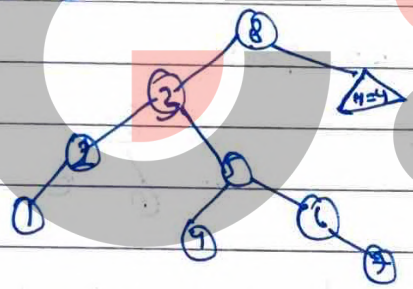


→ Go for this bcz x has higher height than y.

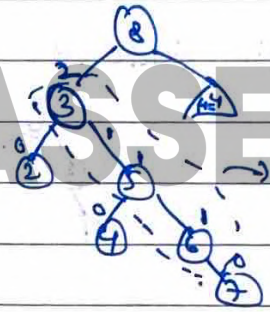
b) Height is same, then choose stick over elbow.

Going against these rules are strictly wrong ie tree may remain unbalanced at that level. So these are not recommended rules, these are mandatory.

Ex:1 Delete 1.



⇒

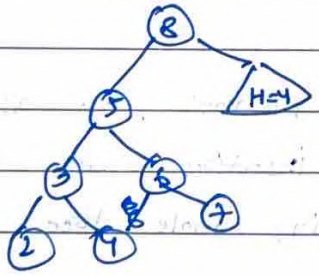


→ Not elbow bcz. G has more height than y

Balanced

UB

⇒

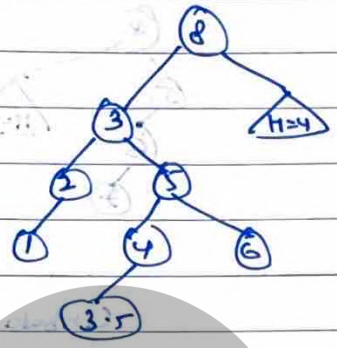


Balancing elbows instead of stick will make this any wrong (Not proving this).

Teacher's Signature _____

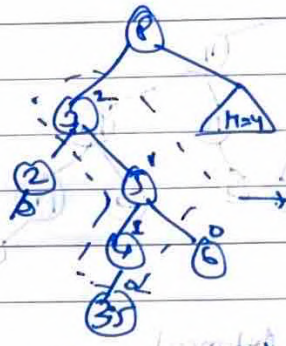
Notes

b) Delete 1



Balanced

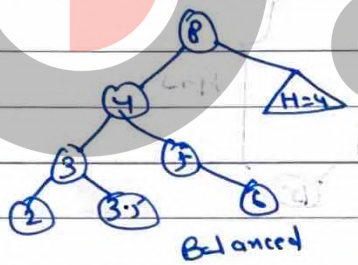
⇒



Choosing elbow bcz 4 has more height than 6.

Unbalanced

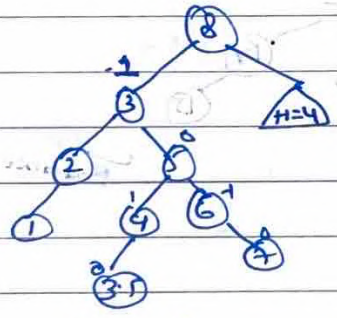
⇒



Balanced

(Now, till root all three a, b, c are balanced, but this may not be the case every time)
↳ like in d

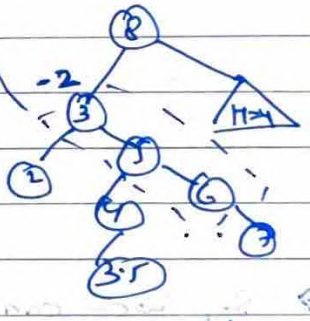
c) Delete 1



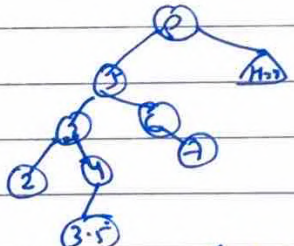
Balanced

Choosing thick bc same height

⇒



Unbalanced

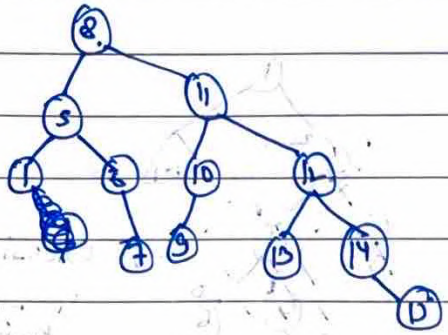


Balanced till root

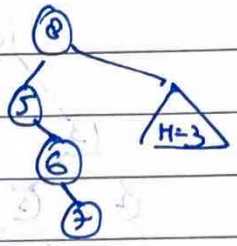
Teacher's Signature

Notes

d) Delete 1

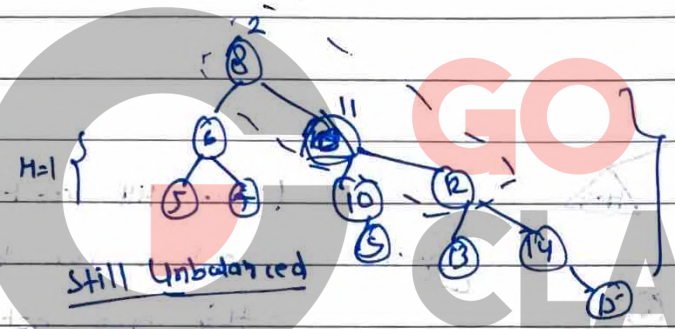


⇒

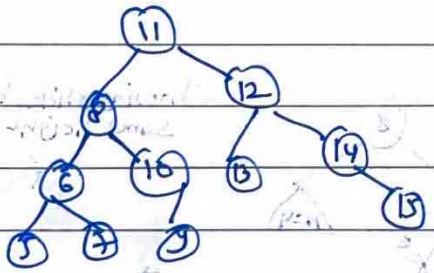


Unbalanced

Balanced



⇒



Balanced now

→ So, we can say that in case of deletion, rotation may propagate to root. ⇒ O(log n) rotation per deletion in worst case.

Notes

Date / /
Page No.

★ Maximum and minimum no. of nodes in an AVL tree of height h:

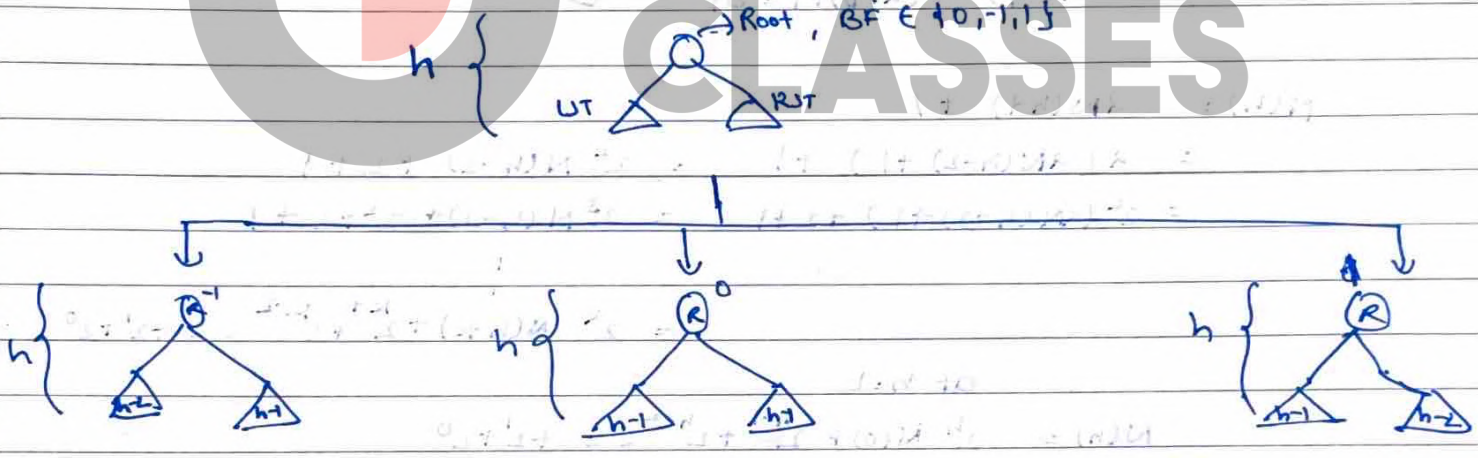
→ For BST it was easy:

Max: $2^{h+1}-1$

Min: h

→ In AVL tree maximum remains same like BST: $2^{h+1}-1$
i.e. each level is full. But let's solve this recurrently.

Ques: max. no. of nodes in AVL tree of height 'h'.



→ Can't be h-2 both side as $h = \max(LST, RST)+1$. So height will be h-1 not h in that case.

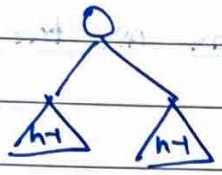
for max. no. of nodes we will consider the case with BF = 0, as we need max. height both side for maximum nodes.

Let N(h): max. no. of nodes in AVL Tree with height h.

Teacher's Signature _____

Notes

Date / /
Page No

So, in 

we can say

$N(h) = N(h-1) + N(h-1) + 1$ → Root

$\boxed{N(h) = 2N(h-1) + 1}$ → Recurrance relation

Base case $N(0) = 1$

now $N(1) = (2*1)+1 = 3$

$N(2) = (2*3)+1 = 7$

$N(3) = (2*7)+1 = 15$

$N(4) = (2*15)+1 = 31$ ✓

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

$= 2(2N(h-2)+1) + 1 = 2^2 N(h-2) + 2 + 1$

$= 2^2(2N(h-3)+1) + 2 + 1 = 2^3 N(h-3) + 2^2 + 2 + 1$

$= 2^k N(h-k) + 2^{k-1} + 2^{k-2} \dots + 2^1 + 2^0$

at $h = k$

$N(h) = 2^h N(0) + 2^{h-1} + 2^{h-2} \dots + 2^1 + 2^0$

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

$\boxed{N(h) = 2^{h+1} - 1}$ ✓

↳ Directly by recurrence → Same.

Now we may be given ques. in 2 forms:

a) Height is given ⇒ Max. nodes we can fill in given height

↕ Same

Counter Invertible

b) Nodes are given ⇒ Min. height possible with given nodes.

Notes

height $\propto$ (no. of nodes per level)

Ex: 15 Node ⇒ Min Height ? 3
16 " ⇒ " " ? 4
30 " ⇒ " " ? 4

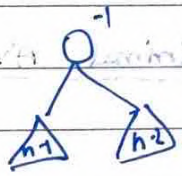
3 height ⇒ max. Node ? 15
4 " ⇒ " " ? 31

} in AVL tree. (Also same in BST)

Que:2 How many minimum no. of nodes are possible in AVL tree of height h?

→ In BST it was h+1 bcz it could be skewed, but in AVL Tree, it needs to be balanced. So we need to do it in our old fashion.

Let N(H) = min. no. of nodes for AVL tree of height h.



for min. nodes we will have case when root's BF = {-1, 1} to have min. height

So:

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

Ie

base case $N(0) = 1$
$N(1) = 2$ Ans
$N(2) = 4$
$N(3) = 7$ (4+2+1)

Teacher's Signature

Notes

Here also que. can be in two forms:

a) height is given ⇒ min. nodes we can fill in given height?

⇕ same

b) node are given ⇒ max. height possible with given nodes?

Ex: 12 node ⇒ Max. height? 4
16 " ⇒ " " ? 4
20 " ⇒ " " ? 5

3 height ⇒ min. nodes ? 7
4 " ⇒ " " ? 12
5 " ⇒ " " ? 20

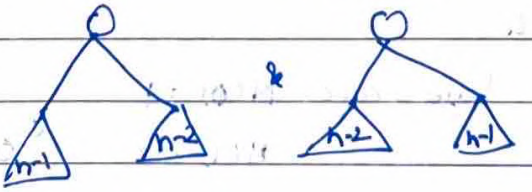
GATE 2009

Que: What is the max. height of any AVL tree with 7 nodes?

⇒ 3

* No. of diff shapes of minimal AVL tree: Let NS(h) represent this for height h.

NS(0) = 1 ○
NS(1) = 2  ,  (Not bcz. minimal)

NS(h) = 

These will be minimal & would be mirror img.

Teacher's Signature _____

Notes

So, we can just find any one type & multiply by two.

So, $NS(h) = NS(h-1) * NS(h-2) * 2$.

$NS(0) = 1$ & $NS(1) = 2$.

$NS(2) = 4$

$NS(3) = 16$

$NS(4) = 128$

So, this is how we can divide & conquer the problem.

Teacher's Signature

Heap is not related with BST.

Date / / Page No.

Notes

Module-5

★ **Heaps:** for a tree to be heap, these two properties need to be satisfied by the tree:

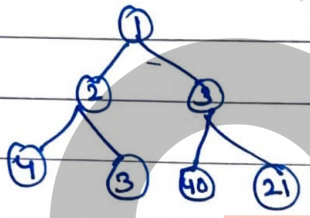
a). **Structural Property:** All levels except last are full. Last level is left filled (Complete Binary Tree)
↳ CBT

b) **Heap Property:**

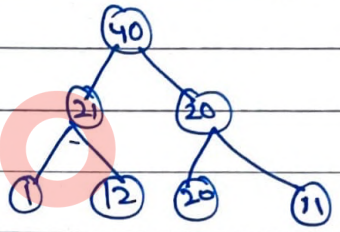
min-heap: parent ≤ child

max-heap: parent ≥ child

Ex:



Min Heap



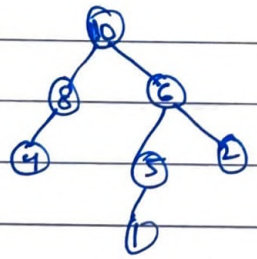
Max. Heap.

NOTE ⟷ ~~BBT~~ CBT → ~~Heap~~. i.e if a tree is not ~~BST~~ CBT, it can't be heap.

Gate GATE 2011

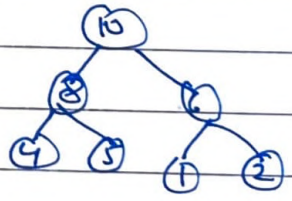
Que: which of the following is a max-heap?

(A)

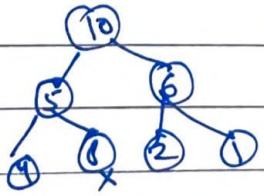


Not even CBT

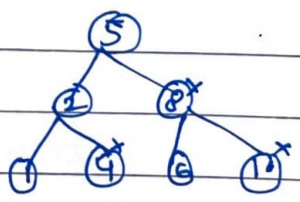
(B)



(C)



(D)



It is BST

Teacher's Signature _____

Notes

Date / /
Page No.

★ Heap in an Array:

Since it is CBT, we can store heap in an Array efficiently.

for a 1-starting index of Array:

Parent (i) : $i/2$

left child (j) : $2j$

right child (k) : $2k+1$

Similar formulas can be created for 0 starting array by using own logics.

★ Max & Min elements in heap:

In a min heap:

→ Min. element is always present on root.

→ Max. element is always present on leaf.

In a max heap:

→ Min. element is always present on leaf.

→ Max. element is always present on root.

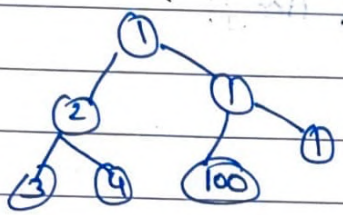
NOTE: Min & Max can also be at other places in case of duplicates but still these elements are also at these places.

Teacher's Signature _____

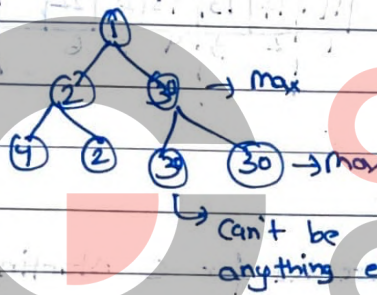
Notes

Date / /
Page No.

Ex: In following min heap, minimum element is also present on leaft, but also need to be at root so it is there, So, our definition is correct.



Similary, max is at internal node too, but also in leaf.



Same things can be applied on the max heap.

★ **k^{th} minimum in min heap:** It is minimum possible n such that there are atleast k elements in the array $\leq n$.

Ex: 1, 2, 3, 4, 5 ..... 99, 100.

min, 2nd min

4^{th} min = 4 bcz {1, 2, 3, 4} → 4 elements are $\leq n.(4)$

In sorted list A[k] is k^{th} minimum.

Teacher's Signature _____

Notes

Page No.

Ex: 1, 1, 1, 1, 1, 1

Here 3rd min is 1 as atleast three no. are less than 1.

→ So, our definition remains good for duplicates also.

→ In sorted array, a[k] is the (k+1)th minimum.

→ Ex: 1, 1, 2, 3, 4, 4, 5, 5, 5, 7, 7, 8, 9, 9, 10, 11

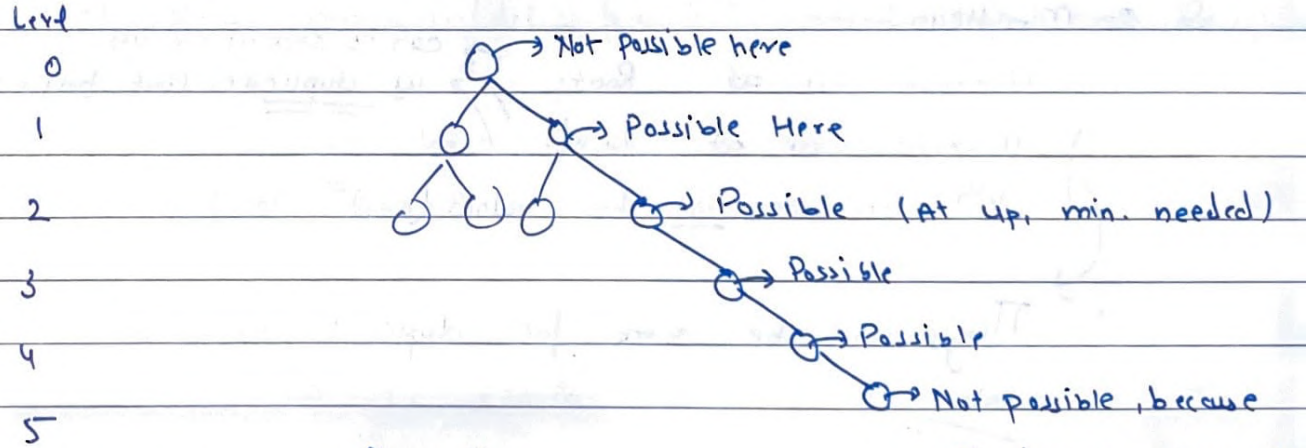
8th min is 5 bcz there are 8 no. ≤ 5.

(1, 1, 2, 3, 4, 4, 5, 5), 9th min. is also 5.

* Kth minimum in Heap:

In case of distinct elements, kth min. can't be at root if it is not minimum.

Ex: 1, 2, 3, 4, 5, ____ 5th min. can be at:



Assume Heap is full and, we have enough elements for this structure

Teacher's Signature _____

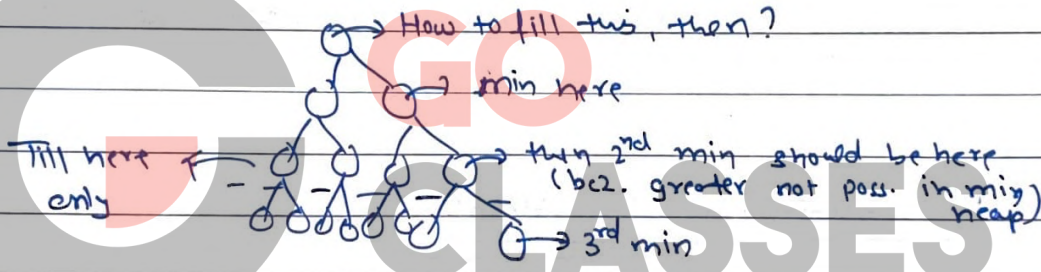
Notes

→ So, 5th min can not be at level 0, and at level greater than 4. It can be at level 1, 2, 3 & 4 only and that too iff we have that level possible by CBT property.

→ NOTE: K^{th} minimum can not be below $K-1^{th}$ level.

Proof by Counterex:

Assume 3rd min. is at level, 3.



So, Not possible

So, In Min Heap:

- Minimum is at Root.
- Maximum is at Leaf. → can be same where also iff duplicate. (but here it's)
- K^{th} min can not be below $(K-1)^{st}$ level.

Things will be same for duplicate elements also.

Teacher's Signature _____

Notes

Date / /
Page No.

Ex: 1, 2, 3, 3, 4, 4.

Here min is 1.
2nd min is 1.
& so on.

So min is at root ✓
2nd min can't be at level 2 ✓
3rd min can't be at level 3 ✓
4th min & 5th min. are same, but check def.
↳ It can't be at level 4th
↳ If it is then that is 5th min.
It can't be at level 5.

and so on.

→ Same analysis/def. for kth maximum in max heap.

GATE CS 03
Qw: In a min-heap with n-elements, the 7th smallest element can be found in the time:

⇒ O(1)
↳ bcz. search atmost $2^7 - 1$ elements.
↳ can be at upto 6 levels only.

Good

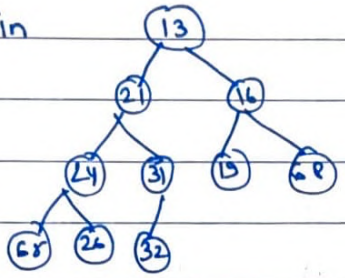
Teacher's Signature _____

Notes

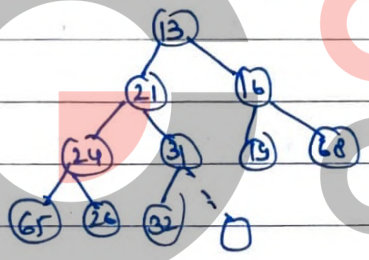
Date / /
Page No.

★ **Insertion in heap:** We already know the structure of tree after each insertion, we just need to label them appropriately.

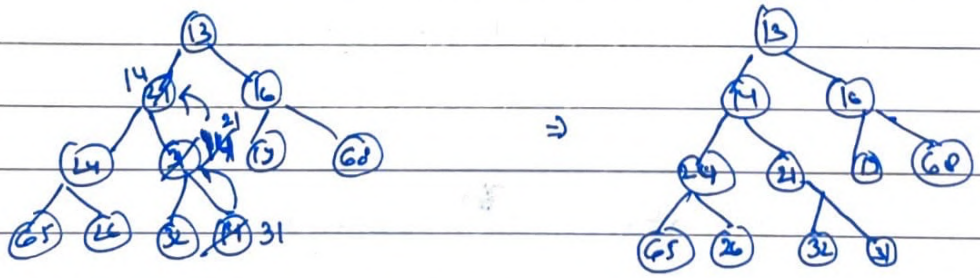
Ex: Insert 14 in



→ W.K.T new tree will look like



Now insert 14 in the tree & compare with its parent if it is greater than parent (lesser than parent in case of max heap), then OK, else swap. Repeat this till root is reached.



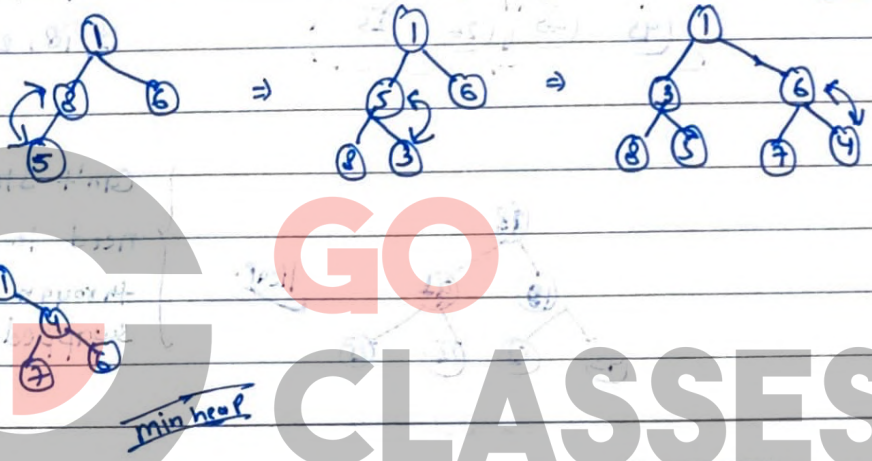
Teacher's Signature _____

Notes

This is how we can do insertions easily. Here one insertion may take log n swaps in worst case. So T.C = O(log n) in worst case & O(1) in best case.

★ We can build heap using repeated insertions.

Ex: min heap by inserting 1, 8, 6, 5, 3, 7, 4. → Top-Down Approach



Time Complexity: W.K.T almost half of the nodes are in leaves & half are internal. And each insertion is taking time equal to its height i.e. log n for leaf node.

T.C :- $\frac{n}{2}(\log n) + \frac{n}{2}\left(\frac{1}{2}(\log n - 1) + \frac{1}{2^2}(\log n - 2) \cdots + \frac{1}{n}(\log 1)\right)$

↳ This + Something $\rightarrow$ $n/2^2 = n$

$\boxed{T.C = O(n \log n)}$ → Build Heap using repeated insertions.

So leaves are taking nlogn & internal nodes n time.

We can reduce the time if we are able to skip the leaves somehow (see later).

Teacher's Signature _____

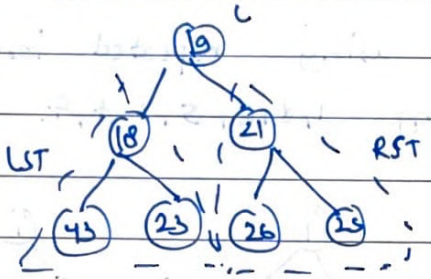
Notes

Date / /
Page No.

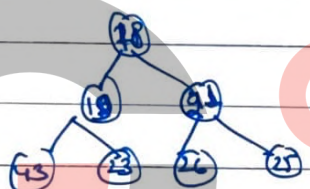
★ Heapify: Assume a node 'i' whose left & right subtrees are already heap. Then can we fix i.

Yes, we can compare i, i→left & i→right & swap the minimum with i.

Ex:



as LST & RST of i are heap, we can compare 19, 18, 21.



Heap.

Can't stop here we will need to go down till leaf through the path we swapped.

~~the way is~~

This process is known as Heapify(i). Remember the necessary condition i.e. LST & RST should be heap & leaves are obvious heap by default.

After heapifying 'i', we may need to heapify its child to which the node are swapped with.

So T.C. of heapify(root): O(log n) in worst case & O(1) in best case.

So, there is another way to build heap:

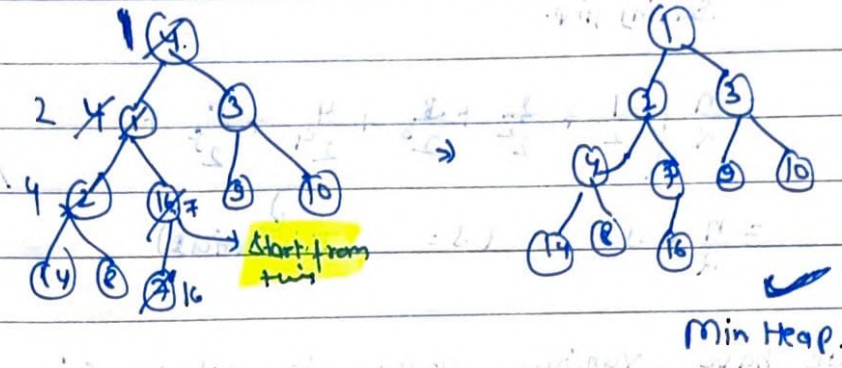
```
for (i = n/2 to 1)      // is internal nodes
    heapify(i)
```


Teacher's Signature _____

Notes

Ex: Build heap: using heapify

4, 1, 3, 2, 16, 9, 10, 14, 8, 7



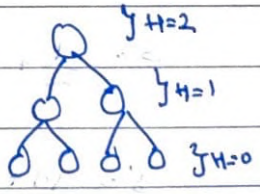
We can also build using top down approach (using repeated insertion) in O(nlogn) time.

The current approach of heapify on all internal nodes is called bottom up approach.

⇒ T.C. of bottom up approach:

W.K.T.

height	nodes
0	$n/2$
1	$n/2^2$
2	$n/2^3$
k	$n/2^{k+1}$



Teacher's Signature _____

Notes

$\log n$ ↳ (max. swaps in worst case)

$\sum_{h=0}^{\log n} h * \text{no. of nodes at height } h$

Total Time = $\left(\frac{n}{2}*0\right) + \left(\frac{n}{2^2}*1\right) + \left(\frac{n}{2^3}*2\right) + \left(\frac{n}{2^4}*3\right) + \left(\frac{n}{2^5}*4\right) + \dots$

↳ (last) Saving time

$= \frac{n}{2}\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \frac{5}{2^5} + \dots\right)$

$= \frac{n}{2} * S$ (S = This series)

Now we have various ways to solve S:

Good Method M1: $S = \sum_{i=0}^{\infty} x^i \cdot i$ where $x = \frac{1}{2}$

W.K.T.

$\sum_{i=0}^{\infty} x^i = \frac{1}{1-x}$ — (1)

Diff. eq (1)

$\sum_{i=0}^{\infty} i\,x^{i-1} = \left(\frac{1}{1-x}\right)^2$ — (2)

Mult. (2) * x

$\sum_{i=0}^{\infty} i\,x^i = \left(\frac{1}{(1-x)}\right)^2 * x$

which is equal to S

So $S = \left(\frac{1}{1-x}\right)^2 * x$

$S = 2$

AGP M2: $S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} \dots$

$S = x + 2x^2 + 3x^3 + \dots$ $(x = 1/2)$

$xS = \quad x^2 + 2x^3 + \dots$

$S - xS = x + x^2 + x^3 + \dots$

$S(1-x) = \frac{x}{1-x} \Rightarrow S = \frac{x}{(1-x)^2} \Rightarrow S = 2$

Teacher's Signature _____

Notes

So as $s=2$

$\therefore T.C = \frac{n}{2} * 2$

$= n$

⇒ So T.C of build heap using heapify is $O(n)$.

Top Down	Bottom up
→ Repeated Insertion	Heapify
→ $O(n \log n)$	$O(n)$
→ $n \log n$ time on leaf	No time on leaf
→ No need of all element in adv. (Book in library, you get some daily)	All elements are req. in Adv. (Mostly used when new data comes in bunches)

★ Findmin & Deletemin operations on Min Heap:

Findmin: Easy just return root value - i.e. A[1]

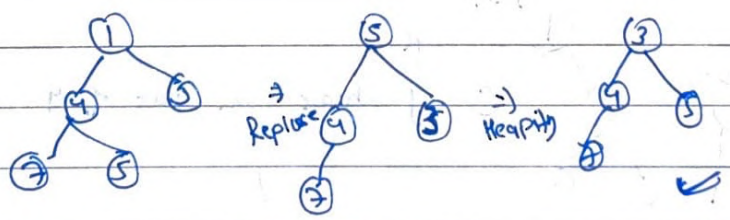
$T.C = O(1)$

Deletemin: In this we want to delete & return value at root. It can be done in two steps.

A[1] = A[N--] , replace root with leaf

heapify (A[1]) , heapify the root. (bcz LST & RST are Heaps)

Ex:



→ Replace → Heapify

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

$= O(\log n)$

Teacher's Signature _____

${}^nC_r$ → Choosing r out of n.

${}^nC_r = {}^nC_{n-r}$

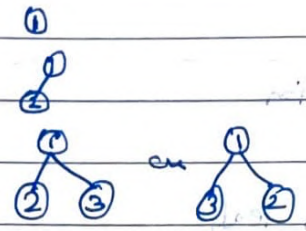
Notes

Date / /
Page No.

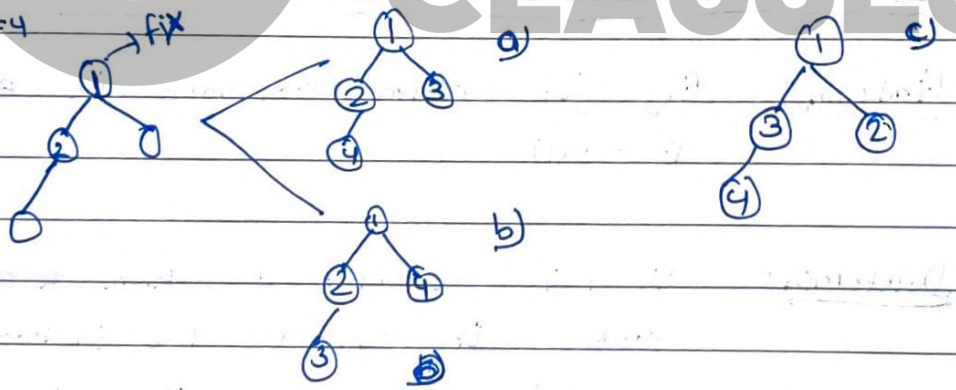
☆ **Minimum Number of Heaps:** Here, Structure is fixed, we can play with labels only

(In BST structure was not fixed, labels were fixed i.e. one label per structure).

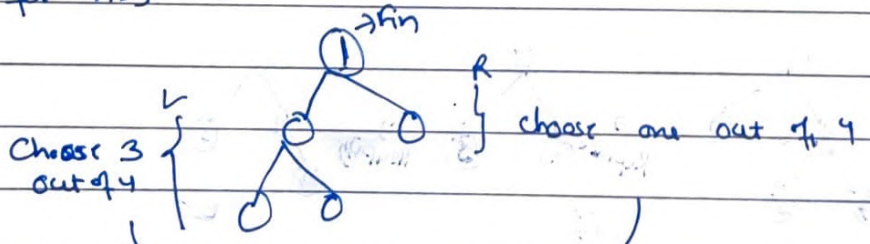
No. of Nodes	No. of Min Heap
1	1
2	1
3	2
4	3
5	?
6	?



for n=4



for n=5



$\rightarrow {}^4C_3$ or ${}^4C_1$

$= {}^4C_3 \times L \times R$ (n=3 → n=1)

$= {}^4C_3 \times 2 \times 1$

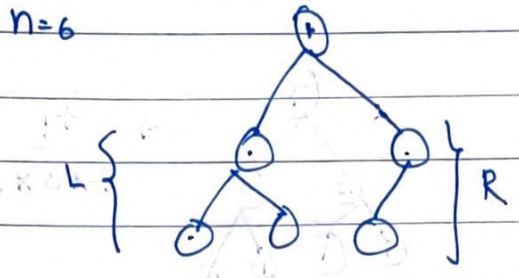
$= 8$

Teacher's Signature _____

Notes

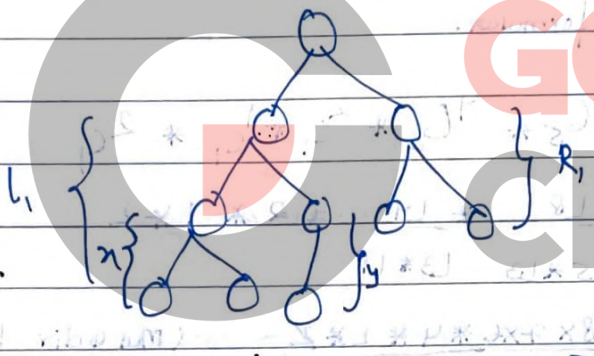
Date / /
Page No.

$n=6$



$$^5C_3 * {}^2C_1 * {}^1C_1$$

$n = 10$



$$^9C_6 * [{}^5C_3 * \underbrace{{}^2C_1}_{x} * \underbrace{{}^1C_1}_{y}]_{L_1} * [{}^2C_1]_{R_1}$$

$n = 15$

$$^{14}C_7 * {}^6C_3 * {}^2C_1 * {}^2C_1 * {}^6C_3 * {}^2C_1 * {}^2C_1$$

So, the gen. formulae can be

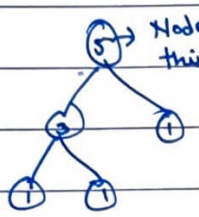
$$a_n = {}^{n-1}C_{L \text{ or } R} * a_L * a_R.$$

where a_i means no. of min. heaps with i nodes.

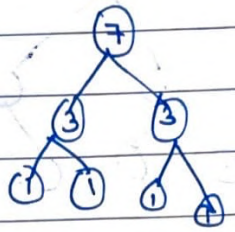
Teacher's Signature _____

Notes

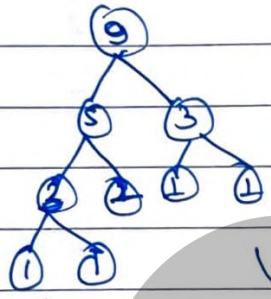
Shortcut for this



$\Rightarrow \frac{5!}{3*5}$



$\Rightarrow \frac{7!}{7*3*3}$



$\Rightarrow \frac{9!}{9*5*3*3}$ } No. of min heaps.

↳ From formulae.

$${}^8C_5 * {}^4C_3 * {}^2C_1 * {}^1C_1 * {}^2C_1$$

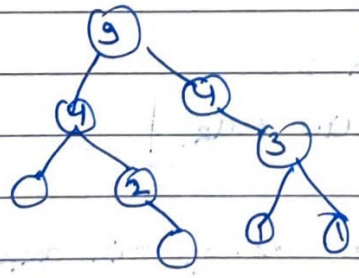
$$\Rightarrow \frac{18}{15*13} * \frac{14}{13*11} * 2 * 1 * 2$$

$$\Rightarrow \frac{8 \times 7 \times 6 * 4 * 1 * 2}{3 * 2}$$ (Multiply by 9, 5, 3)

$$= \frac{9!}{9*5*3*3}$$ Same

This method also works fine for heaps w/o structural property.

Must follow heap property



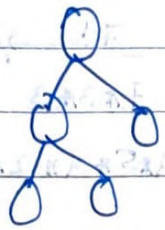
$\frac{9!}{9*4*4*3*2}$

Notes

Date / /
Page No.

⇒ No. of ways to fill a tree with n nodes: $n!$ (labels)

Que: Given this structure, what is the prob., that tree is heap.



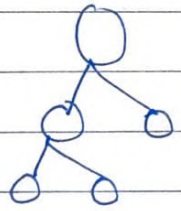
⇒ No. of heaps: $\frac{5!}{5*3}$ } From the given structure

All trees = $5!$

Prob: $\frac{\text{favourable Cases}}{\text{All cases}} = \frac{\frac{5!}{15}}{5!} = \frac{1}{15}$

So 1 out of 15 trees can be heap for above structure.

Another way:



for this to be heap:
root should be min
&
left is heap

P = P(min. root) * P(left is heap) → i.e min. among 3.

$= \frac{1}{5} * \frac{1}{3} = \frac{1}{15}$

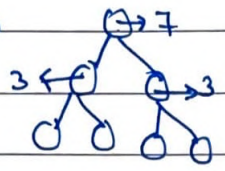
Teacher's Signature _____

Notes

CE 18

Qu: No. of possible min-heaps containing each value from {1, 2, 3 . . . , 7} exactly once is _____?

⇒ Structure for 7 nodes

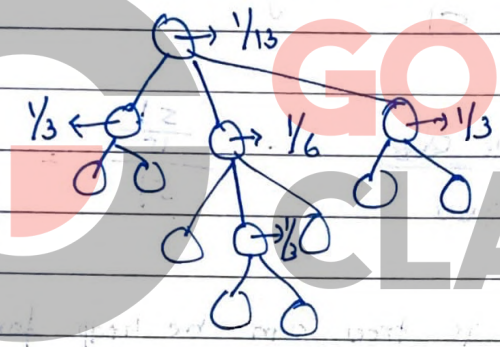


$$\Rightarrow \frac{7!}{7*3*3} \Rightarrow \frac{\cancel{7}^{\cancel{6}^{2}} * 5 * \cancel{4}^{2} * \cancel{3} * \cancel{2} * 1}{\cancel{7} * \cancel{3} * \cancel{3}}$$

$$\Rightarrow 1*5*4*2*1 \Rightarrow 80$$

TIFR 14

Qu: Consider the following tree with 13 nodes.



Suppose nodes are labelled distinctly, each perm. being equally likely. Prob. that the labels are min heap.

$$\frac{1}{13} * \frac{1}{6} * \left(\frac{1}{3}\right)^3$$

Ans.

Teacher's Signature _____

Notes

Date / /
Page No.

☆ No. of shapes of binary tree possible:

Nodes — Trees

1 =

2 =

3 =

n = Catalan No. $= \frac{1}{n+1}\left({}^{2n}C_n\right) = \frac{2n!}{(n+1)!\,n!}$ → Unlabbeled trees

→ So, No. of shapes of binary trees equals Catalan number.

☆ No. of labelled binary trees possible:

As u.k.T, each structure gives n! labelled binary trees.

So, total labelled binary trees $= \frac{2n!}{(n+1)!\,n!} * n!$

n! * Cat. No.

$= \frac{2n!}{(n+1)!}$ → Labeled Trees

☆ No. of labelled BSTs:

As each structure can give one unique BST.

So, total BST $= \frac{2n!}{(n+1)!\,n!} * 1$

Cat. No.

$= \frac{2n!}{(n+1)!\,n!}$ → labelled BST

Teacher's Signature _____

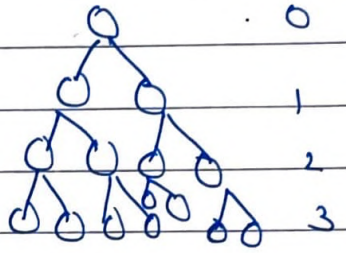
Notes

Date / /
Page No.

Insert Avg. Case Analysis (Not req. for GATE):

If we pick a random node from a tree, then prob. that it is:

(3) leaf Node = $\frac{\frac{n}{2}}{n} = 1/2$



level 0 = $\frac{1}{n} \rightarrow \frac{1}{2^4}$

level 1 = $\frac{\frac{n}{2^3}}{n} \rightarrow 1/2^3$

Level 2 = $\frac{1}{2^2}$

Now, if we insert a new node, then prob. that it will be at leaf node = 1/2

at level 2 = $\frac{1}{2^2}$

at level 1 = $\frac{1}{2^3}$

at level 0 = $\frac{1}{2^4}$

So, Avg. time to insert a node into heap:

time at level 3 → time @ level 2

$$T.C. = \frac{1}{2} * 0 + \frac{1}{2^2} * 1 + \frac{1}{2^3} * 2 + \dots$$

$$= \frac{1}{2}\left(\frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \dots\right)$$

$$S$$

Heap Insertion ↔ Percolate Up
Heap Deletion ↔ Percolate Down

Notes

$$S = \frac{1}{2} + \frac{2}{2^2} + \frac{3}{2^3} + \frac{4}{2^4} + \dots$$

Let $x = 1/2$

$$S = x + 2x^2 + 3x^3 + 4x^4 \dots$$

$$xS = x^2 + 2x^3 + 3x^4 + \dots$$

$$S - xS = x + x^2 + x^3 + x^4 + \dots$$

$$S - xS = \frac{1}{1-x}$$

$$S(1-x) = \frac{1}{1-x}$$

$$S = \frac{1}{(1-x)^2}$$

$$S = \frac{1}{(1-1/2)^2} \Rightarrow 4$$

$$T.C = \frac{1}{2} * 4 = \frac{2}{} = O(1)$$

So Avg. T.C = $\Theta(1)$

Best T.C = $\Theta(1)$

Worst T.C = $\Theta(\log n)$

This way not req. for Gate. Only to explain how to calculate Avg. time complexity.

It was O(1) bcz half of the nodes are at leaf so, 50% chance of no replacements.

Teacher's Signature _____

Notes

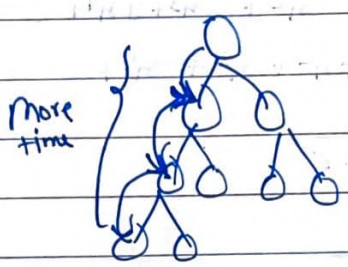
→ Avg. Case for deletion (percolate down)

Prob. of node = root → Prob. of leaf node ($\frac{1}{2}$)

$= \left(\frac{2^0}{n} * 1\right) + \left(\frac{2^1}{n} * 2\right) + \left(\frac{2^2}{n} * 3\right) + \cdots + \left(\frac{2^h}{n} * h\right)$

time @ root node ... time @ leaf node

$= \Theta(\log n)$



So, 50% chance it will come down to leaf so Avg. case is $\log n$.

Avg. $= \Theta(\log n)$

Worst $= \Theta(\log n)$

Best $= \Theta(1)$

So, Heap Time Complexity

	Best	Avg.	Worst.
Insertion	$\Theta(1)$	$\Theta(1)$	$\Theta(\log n)$
Find min	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Delete min	$\Theta(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Heapify @ root	$O(1)$	$\Theta(\log n)$	$\Theta(\log n)$
Searching	$O(1)$	$\Theta(n)$	$\Theta(n)$

Teacher's Signature _____

Notes

Date / /
Page No.

★ Heap Sort:

Step 1: Build Heap out of unsorted array (O(n)) → by heapify method

Step 2: Delete min n times (n * log n)

```
Heapsort(A)
{
    Buildheap(A)                → n
    for(i=1 to n).              → n
        A[1] → A[N]             → 1
        heapify(A[1])           → logn
}
```

T.C. = n + nlogn
= O(nlogn)

★ Heap v/s Balanced BST (AVL):

Heap is good at find min [O(1)] while BBST is good at all of them (O(logn)).

Heap is good bcz. avg. insertion time in it is O(1) & that in BBST is O(logn)

BBST is good bcz. arbitary search take O(log n) in BBST & that in heap is O(n) (Except for min & max).

Teacher's Signature _____

Notes

Date / /
Page No

⇒ Heap to BST can not be done in O(h).

⇒ Let us assume, can be done in O(n), then

Unsorted Array → Heap O(n) → BST O(n) → Inorder → Sorted Array

So, we can sort in O(n) time, which is not possible as min. time for sorting in O(nlogn) in worst case. That's why impossible in o(n) time.

It can be done in O(nlogn) time. Obviously by inserting repeatedly in BST which takes O(nlogn).

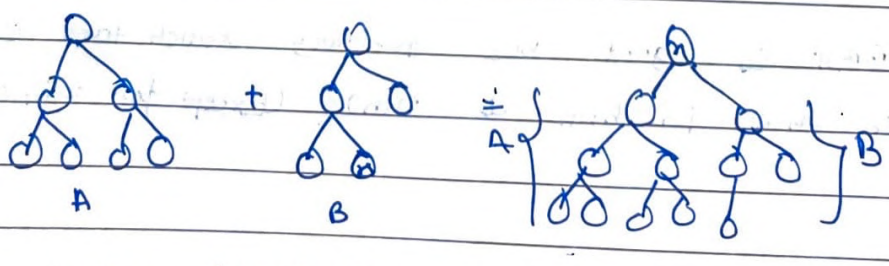
★ Merging two heaps: Generally it's simple, build heap with new array (merged).

↳ Build Heap on this

T.C = O(m+n) ✓

Special Case: If both the heaps are of same height & one of them is perfect then we can merge in O(logn) time.

⇒



& then heapify(n). As LIKE BST and heap.

Teacher's Signature _____

Notes

Date / /
Page No.

★ Priority Queues: (Nothing same as normal queues in implementation)

→ Normal queues are std. mechanism for ordering tasks on as FCFS mechanisms, but some tasks may have higher priority.

→ So, in priority queue, we store tasks using a partial ordering based on priority. & ensure that highest priority task is at the head of queue.

→ It basically has following operations:

- Enqueue (Insert new element).
- Dequeue (Delete min. element)
- Decrease key (Decrease value of element)
- Increase key (Increase value of element).

→ We can implement p.queue using various DS. Let's check their worst case T.C.

	Enque	Deque	Update Key (delete + insert) or (Search + Update)
Unsorted LL	O(1)	O(n)	O(n)
Sorted LL	O(n)	O(1)	O(n)
Unsorted Array	O(1)	O(n)	O(n)
Sorted Array	O(n)	O(1)	O(n)
Bal. BST (AVL)	O(log n)	O(log n)	O(log n)
Heap (min)	O(log n)	O(log n)	O(log n)

↓

do (delete + insert)

However worst case time are same for BBST & heap but avg. time is less for heap.

Teacher's Signature _____

Notes

So heap is considered as best DS for ~~for~~ And both are analogus that priority both used interchangebly.

Decrease key ⇒ bubble up the node
Increase key ⇒ bubble down " "

Notes

Date / /
Page No.

Module-6

Hashing: It is a super fast method for searching, inserting and deleting, O(1).

Hashing is preferred if:

- Order of data does not matter. (unlike Stack & queue)
- Relationship b/w data does not matter (like for BST, relationship matter)

⇒ Applications:

- Google Page Rank Algo.
- File System in Comp.
- Compiler
- ML (Local Sensitive Hashing)
- Digital Signatures
- Many more

⇒ Avg T.C. for hasing is O(1) for all three operation. Worst is O(n) but usually it does not occur.

Direct Addressing Table (Pre-Hash): It is just a fancy name of an array.

In this we insert data n at A[n]. So, if data ranges from 0-999, we can use an array of 1000 entries.

```
insert (key) :  a[key] = key ;
delete (key) :  a[key] = null ;
search (key) :  return a[key] ;
```

It's limitations are:

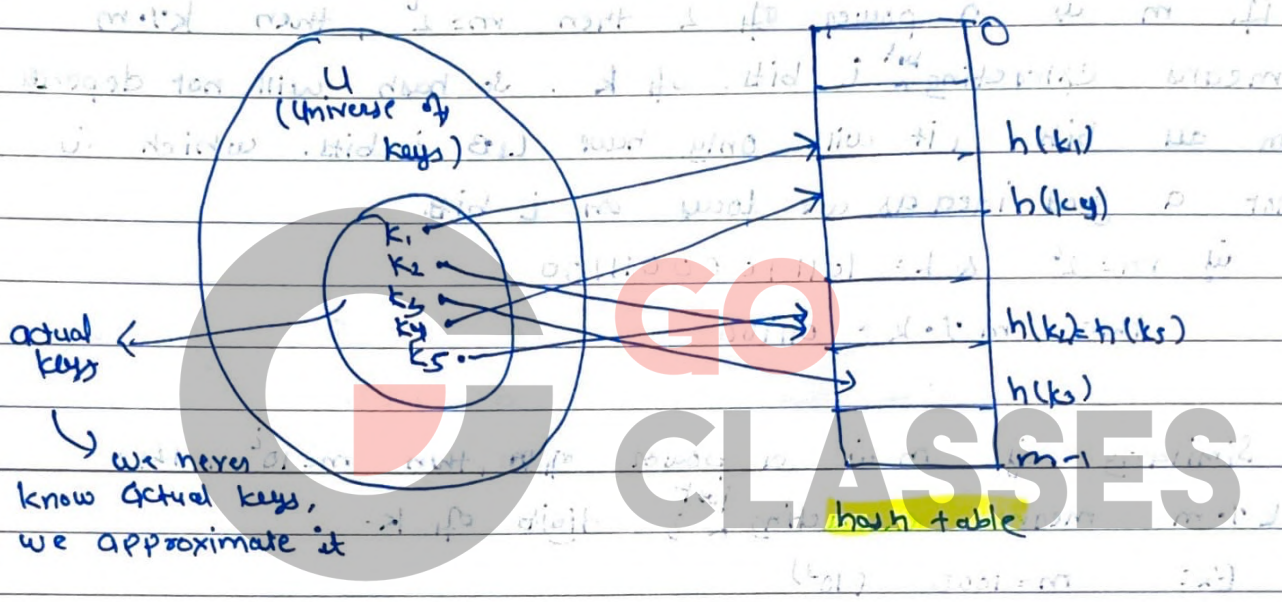
→ Cannot directly store non integer keys in DAT.

→ Key's range should be small. (like can't store phone no.), else require large array for a little data.

Notes

→ Keys should be dense / continuous i.e. not many gaps b/w keys.

Now, to overcome these limitations we can map non-int keys to integers and map larger integers to smaller integers (which is also known as hashing) by hash functions.



$h : U \longrightarrow \{0, 1, \ldots, m-1\}$

hash function

↳ $k_i \rightarrow h(k_i)$

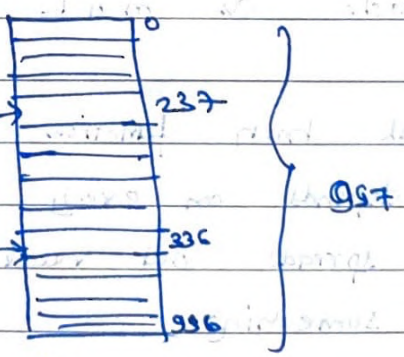
⇒ Example of hashing:

$x = 66752378$ → $h(x) \rightarrow 237$

$y = 68744483$ → $h(y) \rightarrow 336$

$h(k) = k \% 997$.

$m = 997$.



Teacher's Signature _____

Notes

There are various hash functions.

⇒ Division method (mod Operator): Map into a table of m slots

$$hash(k) = k \% m$$

→ If m is a power of 2 then $m = 2^i$, then k%m means extracting last i bits (LSB) of k. So hash will not depend on all bits, it will only have LSB i bits. which is not a good idea as we focus on i bits.

if $m = 2^6$ & k = 1011100001100

then m%k = 011100

→ Similarly if m is a power of 10, then $m = 10^i$, then k%m means extracting last i digits of k.

Ex: m = 1000 (10^3)

k = 123456

k%m = 456 (3 digits)

→ So, there is a rule of thumb of using div. method, that always pick prime number close to power of two, to be m. It works as m & k will not have anything common.

⇒ An ideal hash function should have following properties:

- h(k) depends on every bit of k.
- h(k) spreads out values, so that each slot of table gets something.
- h(k) should be efficient to compute. i.e O(1).

Teacher's Signature _____

Notes

Date / /
Page No.

★ Simple Uniform Hashing (SUH): Simple uniform hashing is when any given element is equally likely to hash into any of the m slots, independently of where any other element has hashed to:

$$P(h(x) = i) = \frac{1}{m}$$ for all x & all i.

where x is key,
i is slot
m is no. of slots.

★ Collisions: Since, size of Univers $|U| > m$ (size of hash table), there must be 2 keys that have the same hash value (PHP)

i.e. $h(k_i) = h(k_j)$ & $i \neq j$

A well designed hash function may minimise collisions, but we still need a mechanism for handling collisions.

There are two types of collision resolution techinques:

a) Closed Addressing / open hashing / Separate Chaining:
   a. → Separate Chaining.

b) Open addressing / closed hasing:
   b. → Linear Probing
   c. → Quadratic Probing
   d. → Double Hashing.

Teacher's Signature _____

Closed → guarantee

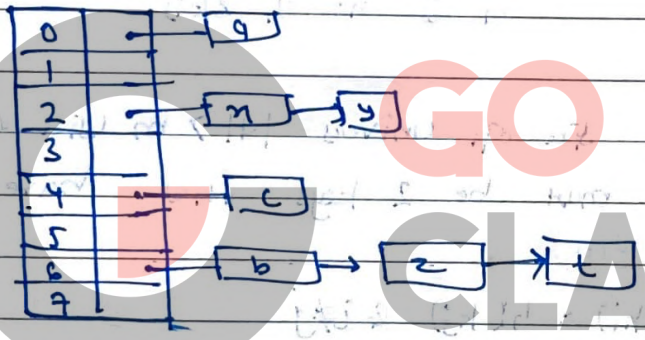
Notes

Date / /
Page No.

★ Closed hashing means we never leave the hash table. Open addressing means index is not completely determined by its hash value. So both are basically synonyms.

★ Closed addressing means guarantee that index is this only but may be out of table.

★ Chaining:



It's worst case performance i.e. when hashing turns into LL i.e. all objects are at same index. i.e. length of chain is 'n'.

Search = O(n)
Insert = O(n)
Delete = O(n)

However, in practice hash table work really well, that is because:

→ The worst case almost never happens.
→ Avg. case performance is really good, O(1).

Teacher's Signature _____

Notes

Date / /
Page No.

★ **Load factor :**

$\alpha = \frac{n}{m}$, $n \rightarrow$ no. of elements
$m \rightarrow$ size of table

It is avg. no. of elements per index. i.e avg. length of chain. It can be greater, less or equal to 1.

★ **Theorem:** In a hash table, in which collisions are resolved by chaining, an unsuccessful search takes avg. case time $\theta(1+\alpha)$, under the assumption of simple uniform hashing.

Proof: Avg. length of a chain is α. If we search for a key, that is not in the table, we need to search the whole list to determine the key is not in the table. Including cost of computing $h(k)$, the total work is $\theta(1+\alpha)$.

α is constant if $m = \theta(n)$ i.e $\alpha = 1$, and so $\theta(1+\alpha)$ is constant.

★ **Theorem:** For Successful Search, avg. case time is $\theta(1+\alpha)$.

⇒ The cost of a successful search is the same as an unsuccessful search at the time the key was added to the table. That is, when we first insert a key, the cost to insert it is the same as the cost of an unsuccefull search.

Teacher's Signature _____

Insertion: Search
↳ success
↳ unsuccess → Insert

Notes

Also intuitively, for successful search, on an avg. we will find in middle of chain
So $\Theta(1+\frac{\alpha}{2})$

Let's have n items, m slots and we are searching i^{th} item, then:

Time to successful search i^{th} item
|||
Time to insert i^{th} item, when there were $i-1$ items in table
|||
Time for unsuccessful search of i^{th} item.
|||
$\frac{i-1}{m}$

So $n_{total}^{avg.} = \dfrac{1+\sum_{i=1}^{n} \frac{i-1}{m}}{n}$

$= 1+\frac{1}{n}\sum_{i=1}^{n}\frac{i-1}{m}$

$= 1+\frac{1}{n}\sum_{i=0}^{n-1}\frac{i}{m} \Rightarrow 1+\frac{1}{n}*\frac{n(n-1)}{2m}$

$= 1+\frac{n}{m2} \Rightarrow \boxed{1+\frac{\alpha}{2}} \in \Theta$

Teacher's Signature _____

Notes

Date / /
Page No.

✱ Open Address / Closed Hashing: In this:

⇒ all elements are stored in the hash table, so $\alpha \leq 1$ i.e. $n \leq m$, that's why closed hashing

⇒ There is no chaining.

⇒ If collision, we move to some other index that's why open addressing.

⇒ To find other index, we probe. And there are three probing techniques in our syllabus.

⇒ How to probe?

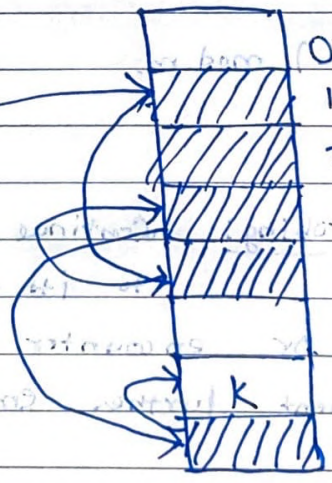
$h: U * \{0,1,\ldots,m-1\} \rightarrow \{0,1,\ldots,m-1\}$

- h: hash function
- U: Universe of keys
- $*$: Trial No.
- $\{0,1,\ldots,m-1\}$: Slot in table

So, $h(K,0), h(K,1), \ldots, h(K,m-1)$ is a permutation of $0, 1, \ldots, m-1$.

$h(K,n)$ means n^{th} trial on key K.

for Ex: Insert K; $h(K,0)=1$, $h(K,1)=4$, $h(K,2)=3$, $h(K,3)=7$, $h(K,4)=6$.



⇒ So basically, here we use empty space in table.

Teacher's Signature _____

Notes

So, in general.

$(h(key) + f(i)) \% m$

i. is trial no.

m is table size

i.e. $h(key) + f(0)$ initially, if full then,

$h(key) + f(1)$, this is full, then

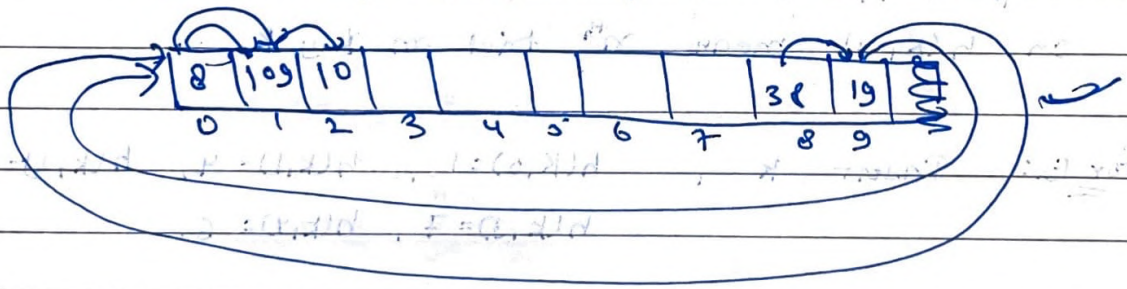
$h(key) + f(2)$ ---

Now, three probing strategies are:

a) Linear probing: $f(i) = i$, i.e. move to next slot if the previous one is full & keep moving to next, next, next.

Ex: $h'(k) = k \% 10$, probing: linear, m: 10,

insert 38, 19, 8, 109, 10



$h(k,i) = (h'(k) + i) \mod m.$

⇒ Searching in Linear Probing: Continue looking from hash to its successive location till we find it or encounter empty space as it will not be present further empty location.

Teacher's Signature _____

Notes

→ **Deletion in Linear Probing:** Search the element & then instead of removing it put some marker there so, that searching in future do not stop here, but new insertion can be done at this marker. If we simply remove it, it will not be able to search in future correctly.

→ **Problem with Linear Probing:** Primary cluster i.e. cluster at some continuous places.

So, to avoid, this issue we upgrade to quadratic probing.

b) **Quadratic Probing:** In this $f(i)$ changes

$$f(i) = c_1 i + c_2 i^2$$

where $c_2 \neq 0$,

else, quadratic = linear.

So $h(k,i) = (h'(k) + c_1 i + c_2 i^2) \% m$

c_1, c_2, are constant

i is trial no.

Ex: $h'(k) = k \% 7$, $m = 7$, probing = quadratic

$c_1 = 0$, $c_2 = 1$

Insert 76, 40, 48, 5, 55.

Teacher's Signature _____

Notes

0	1	2	3	4	5	6
48		5	55		40	76

a) $76 \% 7 = 6$ ✓

b) $40 \% 7 = 5$ ✓

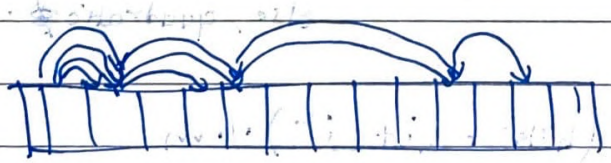
c) $48 \% 7 = 6$ ✗
→ $(6+1^2) \% 7 = 0$ ✓

d) $5 \% 7 = 5$ ✗
→ $(5+1) \% 7 = 6$ ✗
→ $(5+2^2) \% 7 = 2$ ✓
↳ Not 6

e) $55 \% 7 = 6$
→ $(6+1) \% 7 = 0$
→ $(6+2^2) \% 7 = 3$ ✓

Searching & Deleting approach is same as that of linear probing (i.e. marker and all)

→ Problem with Quadratic probing: Secondary clustering, unlike linear probing clustering is not continuous, but still their prob sequence is same.



$(h(k_1=0) = h(k_2=0)) \rightarrow (h(k_1=i) = h(k_2=i))$

So, if two keys have same probe position initially, then their probe sequences are the same.

Notes

Date / /
Page No.

→ **Cycles in quadratic probing:** (Not a problem BTW)

$h(k) + c_1 i + c_2 i^2$ may create cycle such that even when there are empty locations we may not be able to see them.

Fortunately, cycles can easily be eliminated with careful selection of c_1, c_2 to $h(k)$. (Not going in detail)

c) **Double Hashing:** In this we use two hash functions
- h_1 computes the hash code
- h_2 computes the increment for probing

$f(i) = i\,h_2(k)$

So

$$h(k,i) = (h_1(k) + i\,h_2(k)) \,\%\, m$$

Here probe sequence becomes:

$i=0$	$(h_1(k)) \,\%\, m$
$i=1$	$(h_1(k) + h_2(k)) \,\%\, m$
$i=2$	$(h_1(k) + 2h_2(k)) \,\%\, m$
⋮	⋮
$i=n$	$(h_1(k) + n\,h_2(k)) \,\%\, m$

In this, there is no problem such as clustering bcz even if $h_1(k_1) = h_1(k_2)$

then, $h_1(k_1) + h_2(k_1) \neq h_1(k_2) + h_2(k_2)$

So, even if initial probe is same, they don't follow the same path.

→ Searching & Deletion remains same. just probe next, next till you find or get empty slot.

Teacher's Signature _____

Notes

Sequences

☆ No. of possible probes ^ for any given key:

a) In linear probing:

$h(k), h(k)+1, h(k)+2 \ldots$

now $h(k)$ has m choices (say i) (m → size of table)
then all subsequent have only 1 choice ($i+1, i+2 \ldots$)

So, no. of possible probes in LP: m i.e. size of table

b) In quadratic probing:

$h(k), h(k)+a, h(k)+b \ldots$

no of possible probes = m (by same logic)

c) Double hashing:

$h_1(k), h_1(k)+h_2(k), h_1(k)+2h_2(k), \ldots$

↓ m choice ↓ 1 ↓ m choices ↓ 1 only now

So total choices = $m * m$

$= m^2$

because at trial 1 = m
trial 2 = m
then trial 3 = 1
4 = 1
5 = 1
So on.

Teacher's Signature _____

Notes

Date / /
Page No.

d) Uniform Hashing: As each slot have equal chance in this

$h(k), h(k)+f(1), h(k)+f(2), \ldots$

$m \quad m-1 \quad m-2$

Total Choice = $m!$

↳ bcz each slot have equal probability

for slide

* Analysis of Open Addressing (Unsuccesful & Succesful Search):

We are going to do this analysis with respect to uniform hashing instead of linear, quadratic & double.
↳ here $m!$ permutations of probes are possible.

In open addressing α (load factor) can't exceed one.

Theorem: Given an open address hash table with load factor $\alpha = n/m < 1$, the expected no. of probes in an unsuccesful search is atmost $1/(1-\alpha)$, assuming uniform hashing.

⇒ Unsuccesful means we end up hitting an empty slot.

Prob(filled slot hitting) = $\frac{\text{no. of filled slots}}{\text{no. of total slots}} = \frac{n}{m} = \alpha$

P(hitting unfilled slots) $= 1-\alpha$

Teacher's Signature _____

Notes

As we know from prob.

if P(success) = p

then, expected no. of trials for success = $1/p$

So,

expected no. of probes for unsuccessful hits = $\frac{1}{1-\alpha}$

Hence Proved

This can be proved in many way like

$$\sum_{k=1}^{m} k\alpha^{k-1}(1-\alpha) = \frac{1}{1-\alpha}$$

(k: trial, α^{k-1}: success)

OR by recursive method

$$E = (1-\alpha)\cdot 1 + \alpha(1+E)$$

$$E - \alpha - \alpha E = (1-\alpha)$$

$$E\cdot(1-\alpha) - \alpha = (1-\alpha)$$

$$E - \frac{\alpha}{1-\alpha} = 1$$ } Galat hogaya Recursive

$$E = \frac{1}{1-\alpha}.$$

∴ So, if we have n items in hash table, expected no. of attempts for unsuccessful search = $\frac{1}{1-\alpha}$

⇒ Successful search: Its cost is same as an unsuccessful search at the time key was added in table. That is, when we first insert a key, its cost is equal to unsuccessful search.

$$\Theta\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$$

Teacher's Signature _____

Notes

Date / /
Page No.

$$\frac{1}{n}\sum_{i=1}^{n}\frac{1}{1-\frac{i-1}{m}} = \Theta\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$$

Summary:

	Successfull	Unsuccefull
Chaining	$O(1+\alpha)$	$O(1+\alpha)$
Open Addressing	$\Theta\left(\frac{1}{1-\alpha}\right)$	$O\left(\frac{1}{\alpha}\ln\frac{1}{1-\alpha}\right)$

∴ After this Discrete & Continuous Random Variable from probability section.

⇒ Expected items per slot:

Let x_i: no. of items in slot i, n = no. of items, m = no. of slots

$E[x] = x_1 + x_2 + \dots + x_n$

where $x_i = \begin{cases} 1 & \text{if } i^{th} \text{ item maps to slot } 1 \\ 0 & \text{o.w.} \end{cases}$

$P(x_i) = \frac{1}{m}(1) + \left(1-\frac{1}{m}\right)(0)$

↳ Prob. of no. mapping to slot 1

So $E[x_i] = \frac{1}{m}$

$E[x] = E[x_1] + E[x_2] + \dots + E[x_m]$

$E[x] = \frac{1}{m} + \frac{1}{m} + \dots + \frac{1}{m}$

$E[x] = n/m = \alpha$

Teacher's Signature

Notes

Method 2:

$$E[X] = 0\cdot\left(1-\frac{1}{m}\right)^n + 1\cdot {}^nC_1 \left(\frac{1}{m}\right)\left(1-\frac{1}{m}\right)^{n-1} + \dots = n\cdot\frac{1}{m} = \frac{n}{m}$$

⇒ **Expected number of empty Location:**

X: no. of empty locations

$E[X] = x_1 + x_2 + x_3 + \dots + x_m$

$$x_i = \begin{cases} 1 & \text{if slot } i \text{ is empty} \\ 0 & \text{o.w.} \end{cases}$$

$x_i = \left(1-\frac{1}{m}\right)^n$ (No slot here, i.e. in slot not here)

$E[X] = \left(1-\frac{1}{m}\right)^n + \left(1-\frac{1}{m}\right)^n + \dots + \left(1-\frac{1}{m}\right)^n$ (m times)

$E[X] = m*\left(1-\frac{1}{m}\right)^n$

⇒ **Expected number of collisions:**

X: no. of collisions

$E[X]: x_1 + x_2 + \dots + x_n$ → n insertions

$$x_i = \begin{cases} 1 & \text{collision at } i^{th} \text{ insertion} \\ 0 & \text{o.w.} \end{cases}$$

$x_1 = 0$ (obviously)

$x_2 = \frac{1}{m}$

$x_3 = \frac{2}{m}$

Teacher's Signature _____

Notes

Date / /
Page No.

$$E(X) = 0 + \frac{1}{m} + \frac{2}{m} - - + \frac{n-1}{m}$$

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

Teacher's Signature _____

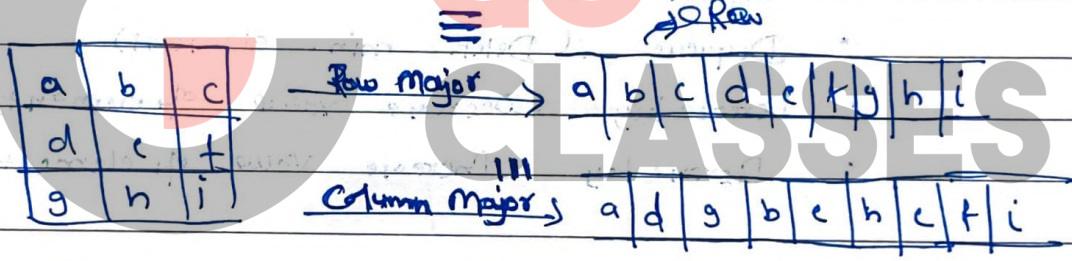
Notes

So heap is considered as best DS for priority queue. And both are analogus that priority queue & heap are both used interchangebly.

Decrease key ⇒ bubble up the node in Heap.
Increase key ⇒ bubble down " " " " (Heapify)

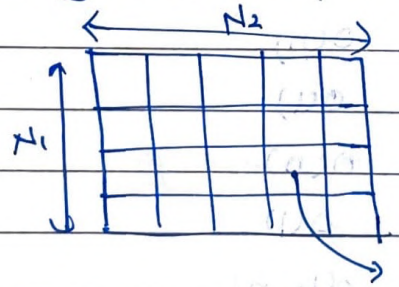
Module-7

★ **Row major and Column Major in Array:** In memory 2-D, 3-D n-D array are stored linearly only. Now there are two approaches for this



2D Array ≡ Memory

⇒ Row major: 2D



a[i][j] = i * N2 + j

↳ As i rows are crossed. * j elements

⇒ Column major: a[i][j] = j * N1 + i

↳ As j cols are crossed * i elements.

Teacher's Signature _____

Notes

→ Beyond 3D, we lose visual intution, so we will have to stick to fixed mathematical notation.

~~Row Major:~~

→ Let N_i = Size of i^{th} dim.
d = Total Dimensions
n_i = index of an element in i^{th} dimension.

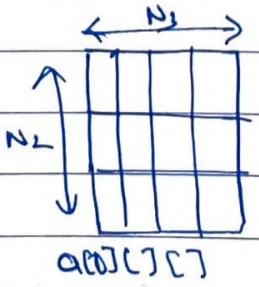
→ W.K.T in row major index of last dim is changing frequently (00, 01, 02, 10, 11, 12, ...)
and in col. major, index of first dim (first index) is changing frequently (00, 10, 20, 01, 21, 22, ...).

→ a[2][3][4] : Row Major : 2 arrays of 3*4 (=) (Traverse like this)
Col. major : 4 array of 2*3. (||)

D	Row Major	Col major
2D	$n_1N_2 + n_2$	$n_1 + n_2N_1$
3D	$n_1N_2N_3 + n_2N_3 + n_3$	$n_1 + n_2N_1 + n_3N_1N_2$
4D	$n_1N_2N_3N_4 + n_2N_3N_4 + n_3N_4 + n_4$	$n_1 + n_2N_1 + n_3N_1N_2 + n_4N_1N_2N_3$
⋮		

→ In 3D

$a[\overset{N_1}{2}][\overset{N_2}{3}][\overset{N_3}{4}]$ Row Major:



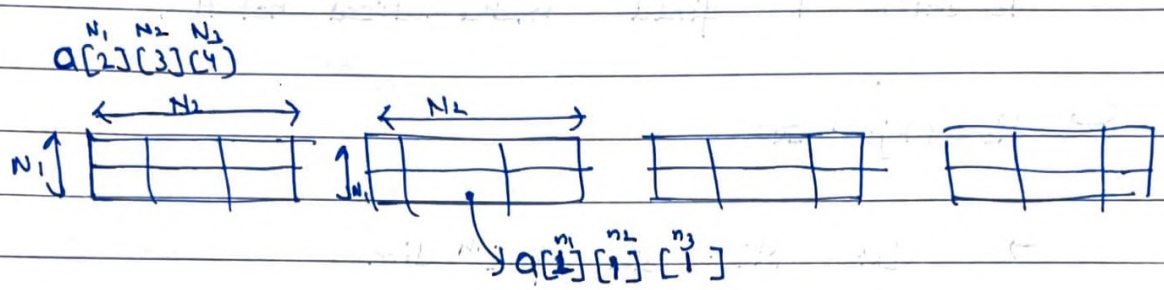
$a[\overset{n_1}{}][\overset{n_2}{}][\overset{n_3}{}]$ = ?

$n_1 * N_2 * N_3 + n_2 * N_3 + n_3$

Teacher's Signature _____

Notes

Column Major:



$n_3 * (N_2 * N_1) + n_2 * (N_1) + n_1$

Que: Let m be 2D array:

m: array [5...15][5...20] of integer.

Assume each int takes 1 memory location & first element is at 200. Then address of element m[i][j] if elements are stored in:

a) Row Major:

$N_1 = 15 - 5 + 1 = 16$

$N_2 = 20 - 5 + 1 = 16$

$n_1 = i - 5$

$n_2 = j - 5$

$= n_2 + n_1 N_2$

$= (j-5) + 16(i-5)$

$= j - 5 + 16i - 80$

$= 16i + j - 85$

Total: $200 + 16i + j - 85$

$= 16i + j + 115$

Ans.

b) Col Major:

$= n_1 + n_2 N_1$

$= (i-5) + (j-5)16$

$= i - 5 + 16j - 80$

$= 16j + i + 115.$

Teacher's Signature _____

Notes

CH: 94

Que: In a compact 1D array rep. for
a) Lower b) Upper (for practice)
triangular matrices (i.e. all the elements above/below diagonal are zero). of size $n \times n$, non-zero elements of each row are stored one after the other. Index of $(i,j)^{th}$ element in this repr. is:

⇒ a) Lower

*	0	0
*	*	0
*	*	*

i=0	i=1		i=2		
*	*	*	*	*	*
j=0	j=0	j=1	j=0	j=1	j=2

$$(i,j) = \underbrace{1+2+3+\cdots+i}_{\text{All rows except last (we want)}} + j-1$$

$$= \frac{i(i+1)}{2} + j - 1$$

b) Upper.

*	*	*	—	*
	*	*	—	0
		*	0	0

$$(i,j) = \underbrace{n + (n-1) + (n-2) + \cdots + n-(i+1)}_{\text{i.e. } (i-1) \text{ times}} + (j-i+1)$$

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

Teacher's Signature _____

These notes are the property of GOClasses. Please do not share them without their consent.

For any questions or issues regarding my handwritten notes, feel free to reach out via email at karan757527@gmail.com or DM me on telegram at @karan757527 (or by clicking here).

I would love to know if you liked my notes. Your feedback and appreciation are invaluable, so don't hesitate to share your thoughts. Click here to connect with me on LinkedIn or you can find me as @agrawalkaran.

Wishing you success on your journey ahead!

- Karan Agrawal