Recurtion :-

* needs to stop somewhere.
* has a recurtion relation.

↳ fact(n) = n * fact(n-1)

$$fact(n) = \begin{cases} 1 \leftarrow \text{stop}, & n=0 \\ n * fact(n-1), & n>0 \end{cases}$$

↖ recurtion relation

```
long fact (int n)
{ if (n==0)
  return 1
  els
  return (n*fact(n-1)};
```

* Character takes

before [0-127 E 8 bits
[128-255 other

then → 2 bytes

*Review var.

---

x^y

$$p(x,y) = \begin{cases} 1, & y=0 \\ x * p^{(x,y-1)}, & y>0 \end{cases}$$

```
long p(int x, int y)
{
  if (y==0)
    return 1;
  else
  return pow(x,y-1);}
```

→ *In the main :-

x^y

```
if (y>=0)
  return (pow(x,y));
  else
  returne (1/pow(x,o));)
```

---

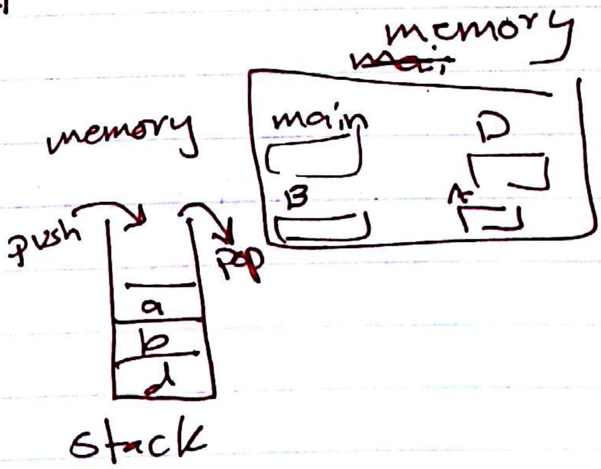
*As a programmer recurtion has no disadvantages :-

a → when it cals A()

func. $\overset{d}{D} \rightarrow \overset{b}{B} \rightarrow \overset{a}{A}$

calls ~~func A~~ ~~D~~ 3

main
↳ call D



*Therefore functions aren't good for the memory and compile.

*at Recurtion : .

→ fact (2) → each time builds anew space in stack so it can lead to stack over flow

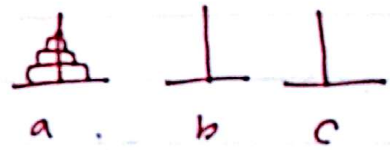
*TRy TO AVOID RecURTION! ← use loops (doesn't build stacks!) → St: doesn't call functions.

Using Recortion:

↳ main functions. (general case)

↳ Reduce a lot of code

Ex: Tower of Hanoi :



sol :

$a \xrightarrow{n-1} b$

$a \xrightarrow{1} c$

$b \xrightarrow{n-1} c$

move n from a to c using c

Hanoi (n, a, b, c) = { Hanoi(n-1, a, c, b) $\xrightarrow{n>0}$ *one at a time . *small ones above.

a → c

Hanoi (n-1, b, a, c)

Code:

```
void Hanoi(int n, int a, int b int c)
{ if(n>0)
  { Hanoi (n-1, a, c, b)
{
     sys.out.print (a + "→" + c)
     Hanoi (n-1 , b, a, c); }

}
```

when calling

++X ← skips 1st value

X++ ← never changes X

X+1 ← X is kept as a value and incremented

$\#$ of calls $= 2^{n+1} - 1$ → To improve the algorithim so that # of calls = # of moves.

$\#$ of moves $= 2^n - 1$

Ch. 2 Algorithm Analysis

CPU → 1 GHz → ساعة / نبضة / ثانية ↖ space ↘ time

* For huge Data Algorithm analysis is important.

Review sorting!

in Algorithm 10/10 $\boxed{n^2 > 1000n}$

- Running time calculation:

  - $T(n) = O(f(n))$ if there are constants c and n_0 such that $T(n) \leqslant c f(n)$ when $n \geqslant n_0$.

  - $T(n) = \Omega(g(n))$ if there are constants c and n such that $T(n) \leqslant (c(g(n)))$ when $n \leqslant n_0$.

  - $T(n) = \Theta(h(n))$ if and only if $T(n) = O(h(n))$ and $T(n) = \Omega(h(n))$

not actual time ع مدران تنفيذ الكل

* If $T_1(n) = O(f(n))$ and $T_2(n) = O(g(n))$, Then

  a) $T_1(n) + T_2(n) = \max(O(f(n)), O\,g(n))$ → 2 algorithm ورا بعض

  b) $T_1(n) * T_2(n) = O(f(n) * g(n))$. during each other

$T_n\, O(n^2)$ = [n [n^2 [c

T_n:

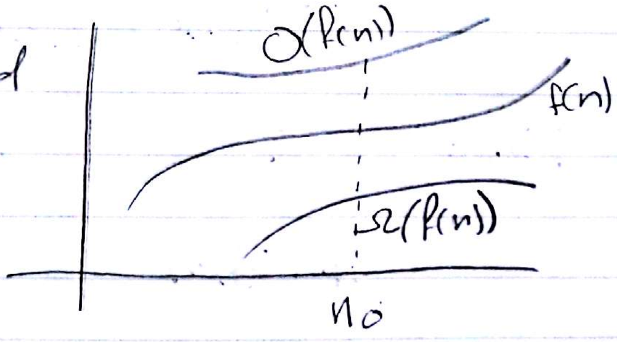
* If $O = \Omega$ → stable Algorithm we get Θ.

O → upper bound

Ω → lower bound.

*In huge n; any constant is neglected

* we study Algorithm when using loops while / for / recursion / do while



* $f(n) = 5n^3 + 17n^4 + 3n^2 + 2 < 5n^4 + 17n^4 + 3n^4 + 2n^4$

$< 27n^4 = O(n^4)$

```
* for (i=0; i<n ; i++)          n
    for (j=0 ; j<n*n ; j++)     n^2  } n^3
        ?

  for (k=0 ; k<1*n ; k++)       n^2
        ?
```

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

```
* For (i=0, i<n*n; i++)          n^2
    for (j=i ; j<n*n, j++)       n^4 - n^2
    for (k=0; k<n; k++)          n
```

$n^2(n^4 - n^2)n$

```
if      for(k=i ; k<n; k++)
if      j<n*n  -> n^2
         ↑ controls the loop
```

---

*

$$fact(n) = \begin{cases} 1 & n=0 \\ n * fact(n-1) & n>0 \end{cases}$$

← const

while → condition unknown

for → fixed times

```
long fact (int n)
{ if (n==0)
      return 1;
  else
  return (n* fact(n-1)); }
```

← n here is a value.

$$T(n) = \begin{cases} d & n=0 \\ C + T(n-1) & n \geq 1 \end{cases}$$

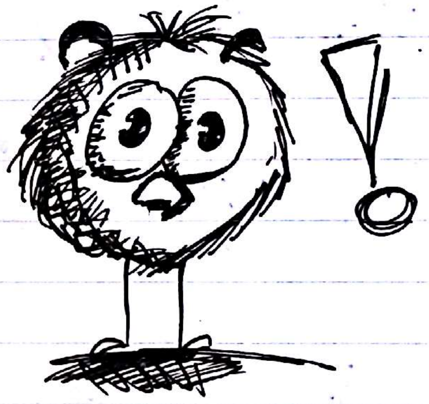
← n here

عدد المرات

$$T(n) = C + T(n-1)$$
$$T(n-1) = C + T(n-2)$$
$$T(\text{[illegible]}) = C + T(0)$$
$$= nC + d \rightarrow T(n) = O(n).$$

But
```
fact = 1;
for (i=0; i<=n ;i++)
    fact*=i.
```
$T(n) = O(n) \leftarrow$ loops in for are better though!

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



$$T(n) = 2T(n/2) + n$$
$$T(n/2) = 2T(n/4) + n/2$$

$$T(n) = 2\left[2T(n/2^2) + \frac{n}{2}\right] + n$$

$\vdots$

get the parameter to 1

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

let $\frac{n}{2^K} \rightarrow n = 2^k \rightarrow K = \log_2 n$

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

$$dn + n \log n$$

$$O(n \log n)$$

Insertion sort :

← algorithm
ال Code
$j=1$

times

times	cost		
n	1	←	for (j = 2 ; j <= n ; j++)
			Begin
$n-1$	2	←	Key = A[j] ;
$n-1$	3	←	i = j - 1 ;
$\sum_{j=2}^{n} T(j)$	4	←	While (i > 0 and A[i] > key) Do
$\sum_{j=2}^{n} T(j)-1$	5	←	A[i+1] = A[i] ;
$\sum_{j=2}^{n} T(j-1)$	6	←	i = i - 1 ;
			end while
$n-1$	7	←	A[i+1] = key ;
			end for.

last term ask for ? & don't enter

تابعة for N

$f(\text{loop}) = n$

$T_{\text{statements in loop}} = n-1$

$T = T_2 + T_3 + T_4$

each card has a time.

$= \sum_{j=2}^{n} T(j)$

A

A	5	7	2	4	10	8
	1	2	3	4	5	6

5	7

2	5	7

i	j	key
1	2	7
1 ~~2~~	3	2
⋮		
4 ~~5~~	6	8

2	4	5	7	8	10

Time Analysis

* Best Case / worst case
when [illegible] Data [illegible]

كأنو تالعة للعام

Best case $\rightarrow T(n) = c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1)$
$+ c_5(0) + c_6(0) + c_7(n-1) = O(n)$

worst case ~~c4~~ $4 \rightarrow$ as it for (i=0; i<n; i++) n
for (j=i; j>0, j++) n

n^2

Average case $\rightarrow$ Random

$c_1(n) + c_2(n-1) + c_3(n-1) + c_4 \sum_{j=2}^{n} t_j$
$+ c_5 \sum_{j=2}^{n} t_j - 1 + c_6 \sum_{j=2}^{n} t_j - 1 + c_7(n-1).$

EX: $T(n) = \begin{cases} d & n=1 \\ 2T(\frac{n}{2})+10 & n>1 \end{cases}$

طريقة بالتحليل

$\times \frac{2-1}{2-1} = \frac{2^k - 1}{2-1}$

$T(n) = 2T(\frac{n}{2}) + 10$

$\vdots$

$= 2^k T(\frac{n}{2^k}) + 10 \left[2^{k-1} + 2^{k-2} + \dots 1\right]$

$T(n) = 2^k T(\frac{n}{2^k}) + 10 \left[\frac{2^k - 1}{2-1}\right]$

$T(n) = nT(1) + 10(n-1)$

$T(n) = O(n)$

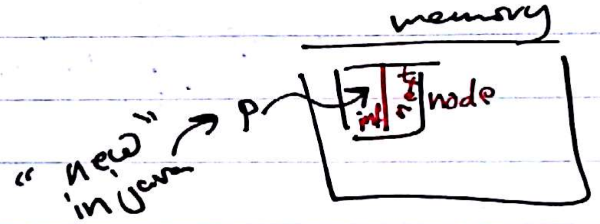
Linked List, Stack, Queue.

*floating point!

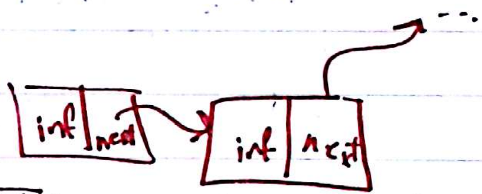
list $a_{i-1} \rightarrow a \rightarrow a+1$

- operations on lists:
  * print element
  * print list
  * Insert → Exp
  * Delete
  * Search
  * make null.

*Array:
→ fixed length (-)
→ stored in series
A[i]=Adress = A+(i-1) * # type



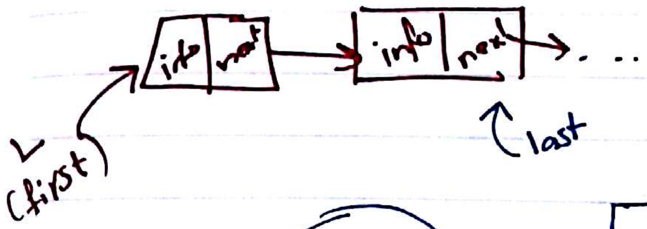
in C → p=(int)malloc(signature(int))



↳ This saves memory.

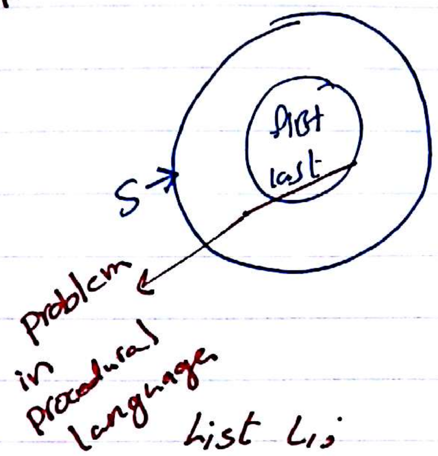
	Array	link list
print element (index)	constant	O(n)
print list.	O(n)	O(n)
search	O(n)	O(n)
Insert	O(n) → shift down	constant.
Delete	O(n) → shift up	constant.
Make null	Constant	O(n)

UML for linked list:



Node

Node
element: object next: Node
+ Node (object)



Linked List

Linked List
- first, last : Node - count : int
+ Linked List ( ) + getFirst ( ): object + getLast ( ): object + addFirst (object): void + addLast (object): void + add (objects, int): void + removeFirst ( ): boolean + removeLast ( ): boolean + remove (index): boolean + remove (object): boolean + printList ( ): void

List L;

∴ use last in java.

→ in procedural language use header node (so list is always there.

*header node is always used as a reference

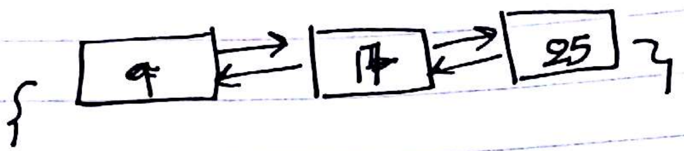
Code:

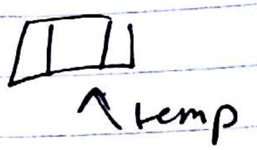
```
public class Node {
        Object elements;
        Node next;
    public Node (object x)
        { .element = x
            }
              }

public class LinkedList
    {   private Node first, last;
        privat int count;
      public LinkedList ()
         {
         }
```

constructor has no type.

Double LinkedList :





adding

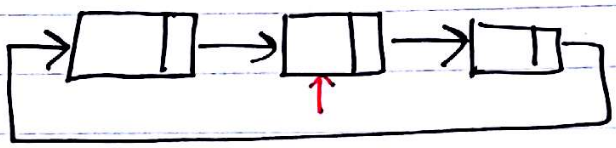
temp.next = p.next;

temp.previous = p;

p.next = temp;

temp.next.previous = temp;

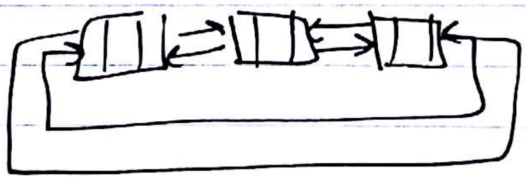
Circular Linked List :



← when printing use another pointer.

* has no header
* has no first nor last

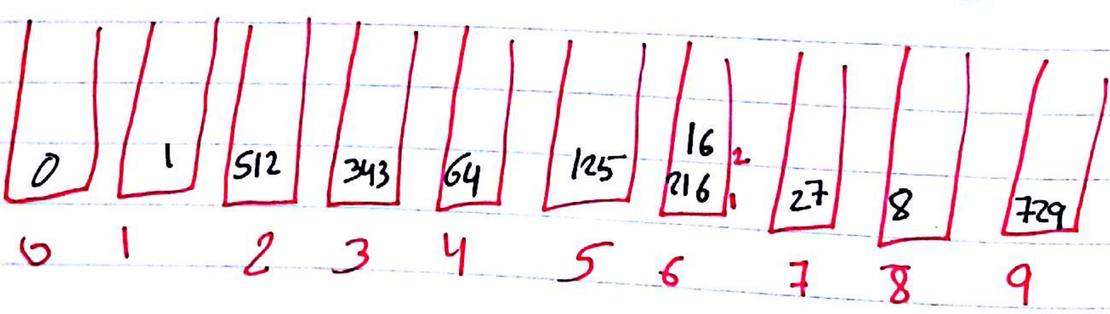
Circular double Linked List :



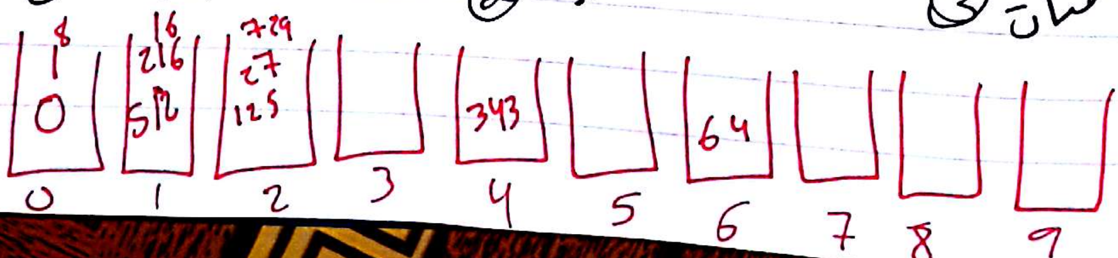
Radix Sort : 64, 8, 216, 512, 27, 729, 0, 1, 343, 125, 16

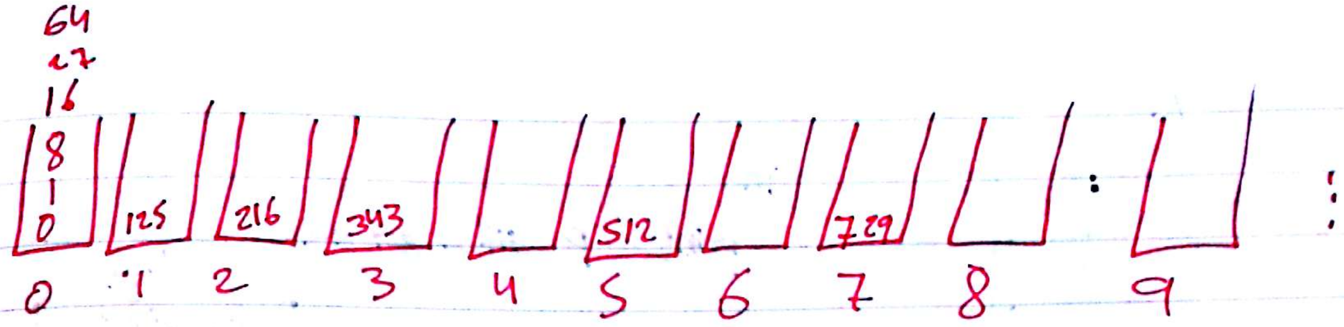
max = 729 O(n) → 3 digits

على حسب الاحاد



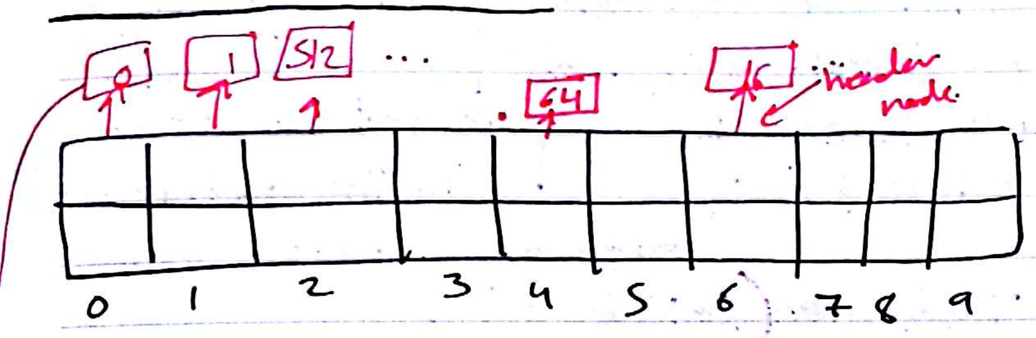
① احاد ② عشرات ③ مئات ④ —





time = $O(n)$ → n from max + (Digits)(n)

↑ Best time for sorting



if $n = 10^5$ ← 10 buckets

space = 10^6 unit X 2 ← using 2 Buckets sets at a time.

$= 8 \times 10^6$ bytes

= 8M bytes

↑ disaster!

so we use

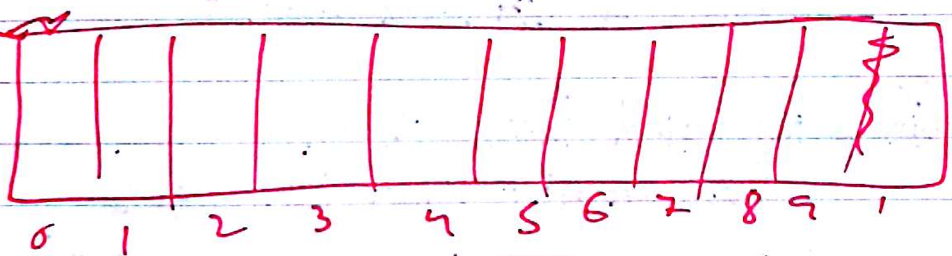
↑ array list

space = ~~10x10^5~~ n units

$= 4 \times 10^5$ int

400k byte

step ②



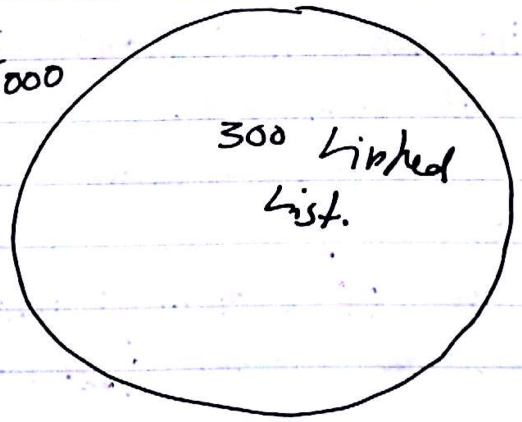
Curser Implementation:-

300 course ⇒ 45 students → 13500

3000 student → 6 hours → 6000

↓80

6000



A

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

:

in

print p till I get null

Node
info : Object next : int
+ Node(object)

Algorithm :-

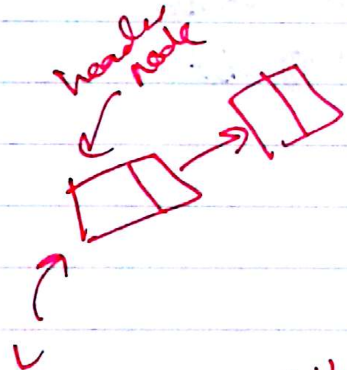
```
int newNode(){
int position;
position = Cursor[0].next;
Cursor[0].next = Cursor[p].next;
Cursor[p].next = 0
return p; }
```

Curser : Array [max size of Nodes].

we creat the array in the constructer.

empty when 0.next = 0

```
* Void ~~free~~ freeNode (int p){
   Cursor [p].next = Cursor [0].next
   Cursor [0].next = p
```




```
* boolean Empty (int L) {
   return(Cursor[L] .next==0);
```

```
~~find~~ * int find (int L, Object x){
   int Current = Cursor [L].next ;
while( cursor[current].info != X && (Current != 0)
   Current = Cursor[current]. next;
   return Current; }
```

[illegible]

~~[illegible]~~

```
* void insert (List L, ~~p~~int p, Object)

   int temp = new Node();
   if (temp==0)
      "out of memory"
  else {
  Cursor ~~of~~ [temp].info = X;
Cursor[temp].next = Cursor [p]. next.
```

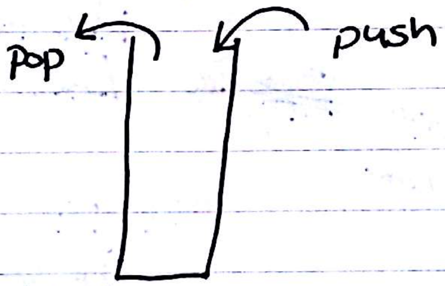
```
cursor[p].next = temp;

* Void delete (int L, inx x) {

    int p;
    p = find previous (L, x);
    if (curser(p].next != 0) {          ← مش آخرها
        ~~int t = cur~~
      cursor[p].next = cursor(cursor(p.next)).next;}

  * int findpreivos(int L, int x) {
      int p = L;
    While (cursor[p].next != 0 && cursor[p.next].info !=
        p = cursor[p].next;
        return p; }
```

Stack:



stack

* First In Last out.
  Last In First Out.

Functions:

- push
- pop
- empty
- top.

only !

Stack → Linked List
Stack → Array

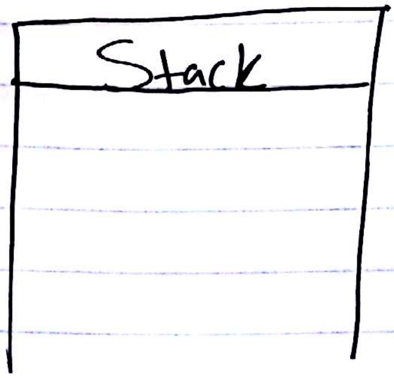
---

push (Lis

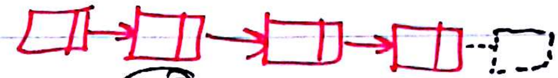
Stack is an object

s. push

→ class stack ←



Void



c push → add first	or	push → add last
c pop → remove first		pop → remove last

c O(n)

time better!

privat (utility functions)

private functions

```
void push(object x) {
    addFirst(x); }
void pop() {
    removeFirst(); }

Object top() {
    return getFirst(); }
```

If array
use a pointer

```
stackpointer = 0
void push (objectx)
   {   if (stack pointer +1 == size)      ← error
   else S[stack pointer++] = X; }
```

Array now

```
void pop () {
     if (stack == 0) error
  else   Stack pointer --; }

object top() {
     if (stackpoint == 0) ———
  else  return (S[stackpointer - 1]); }
```

* ~~Info~~ infix to postfix conversion :-

5 * 2 infix

* 5 2 prefix

5 2 * postfix

Ex: a+b*c+(d*c+f)*g → abc*+de*f+g*+

(a → operance; infix → a+b*c+(d*c+f)*g; postfix → abc*+de*f+g*+)

pred('*', '+') → T ← priority

pred('*', '/') → T

pred('*', '/') → T

pred('-', '/') → f

pred (op, op) → T

---

pred('(', op) → F   9*(7+5)/2

pred(op, '(') → F

pred(op, ')') → T   must be *

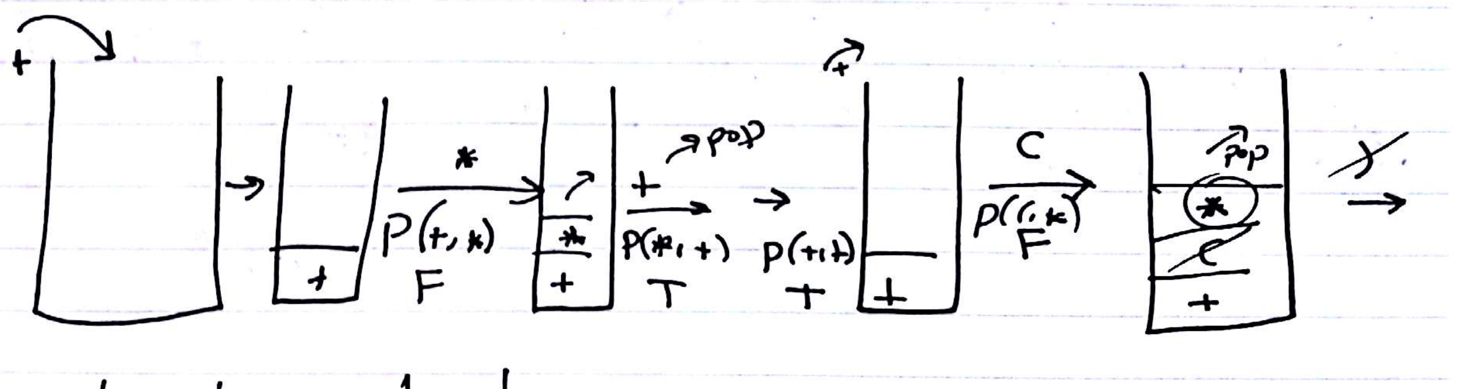
pred(')', '(') → Error (no operation) (5)(7)

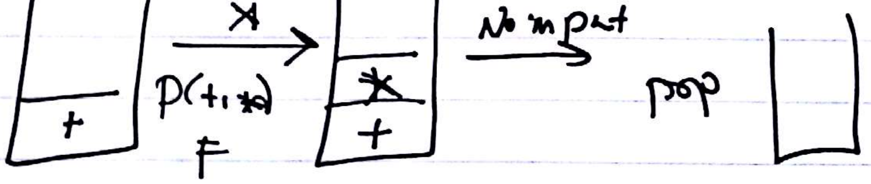
---

a+b*c+(d*c+f)*g

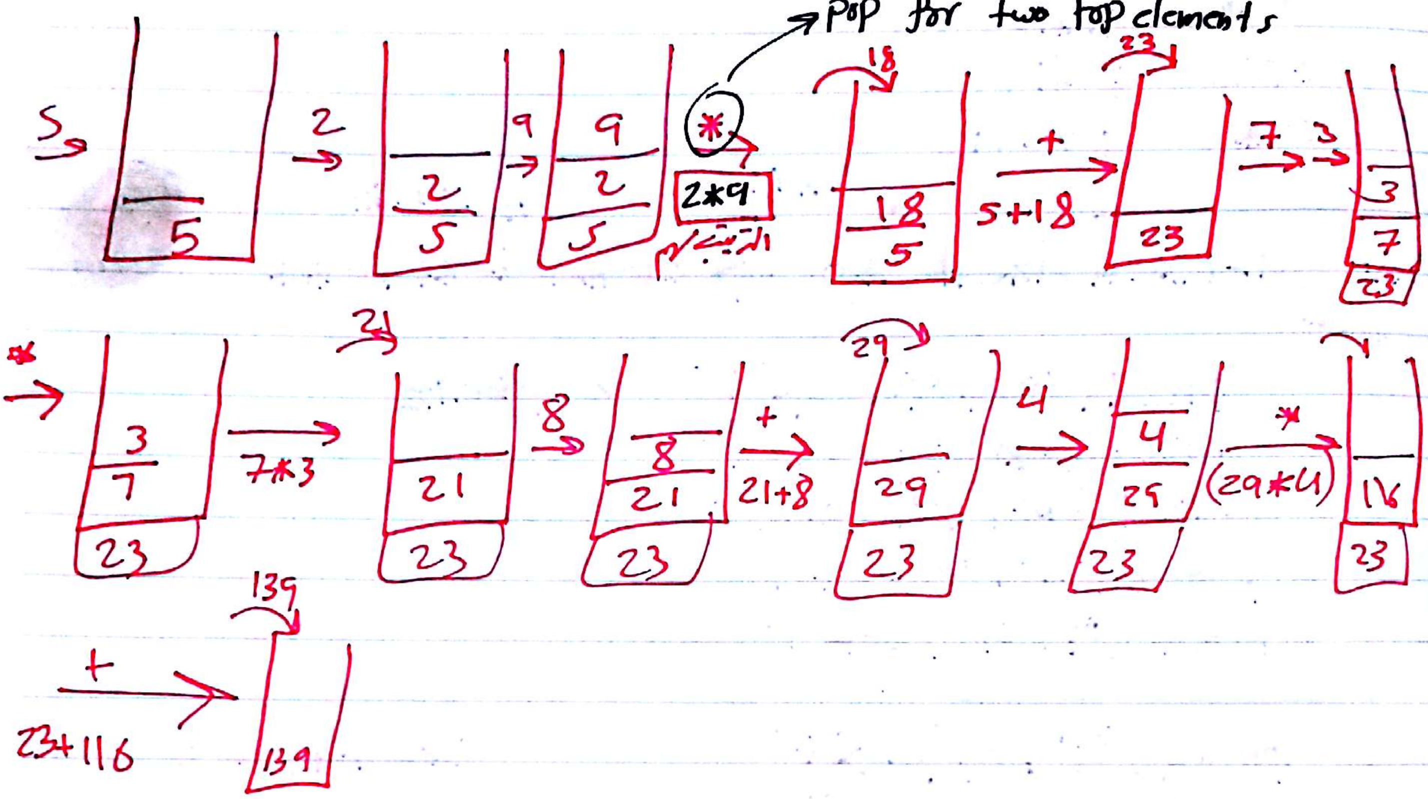
abc*+de*f+g*+

↑ output



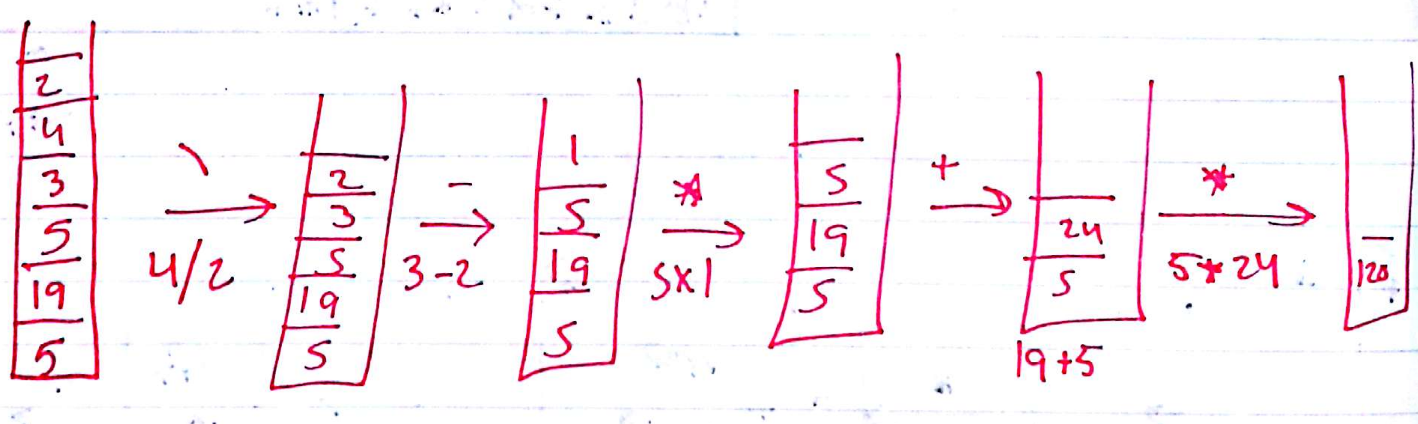


abc* + de *f + g*+



$5+(19+5*(3-4/2))$

$\boxed{5,19,5,3,4,2,\backslash,-,*,+,*}$



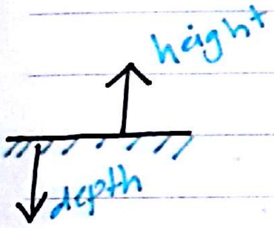
=120

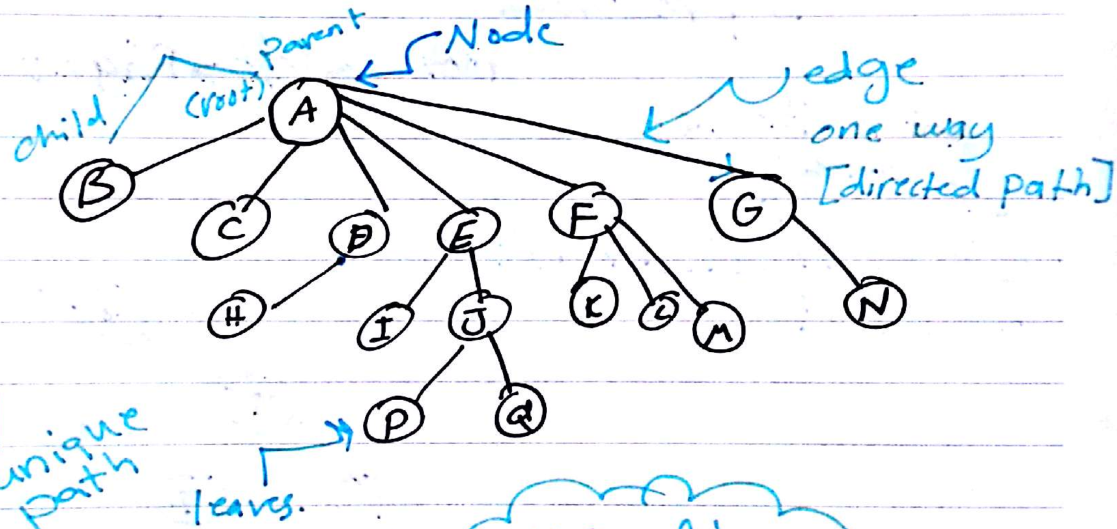
Tree

* A tree is a collection of nodes
* A tree consists of a distinguished node r, called the root, and zero or more sub trees $T_1, T_2, T_3, \dots T_K$ each of whoos roots are connected by directed edge to r.
* The root of each subtree is said to be a child of r, and r is the parent of each subtree root.

* # of nodes = n-1.

* depth: The depth of r, is the length of the unique path from the root to r.
* Height: The hieght of r; is the length path from r, to the leaf.





Depth

* depth (J) = 2
* depth (A) = 0
* depth (Q) = 3

unique path

H(leaf) = 0
depth (root) = 0

Height

* H(E) = 2 not 1
* H(A) = 3

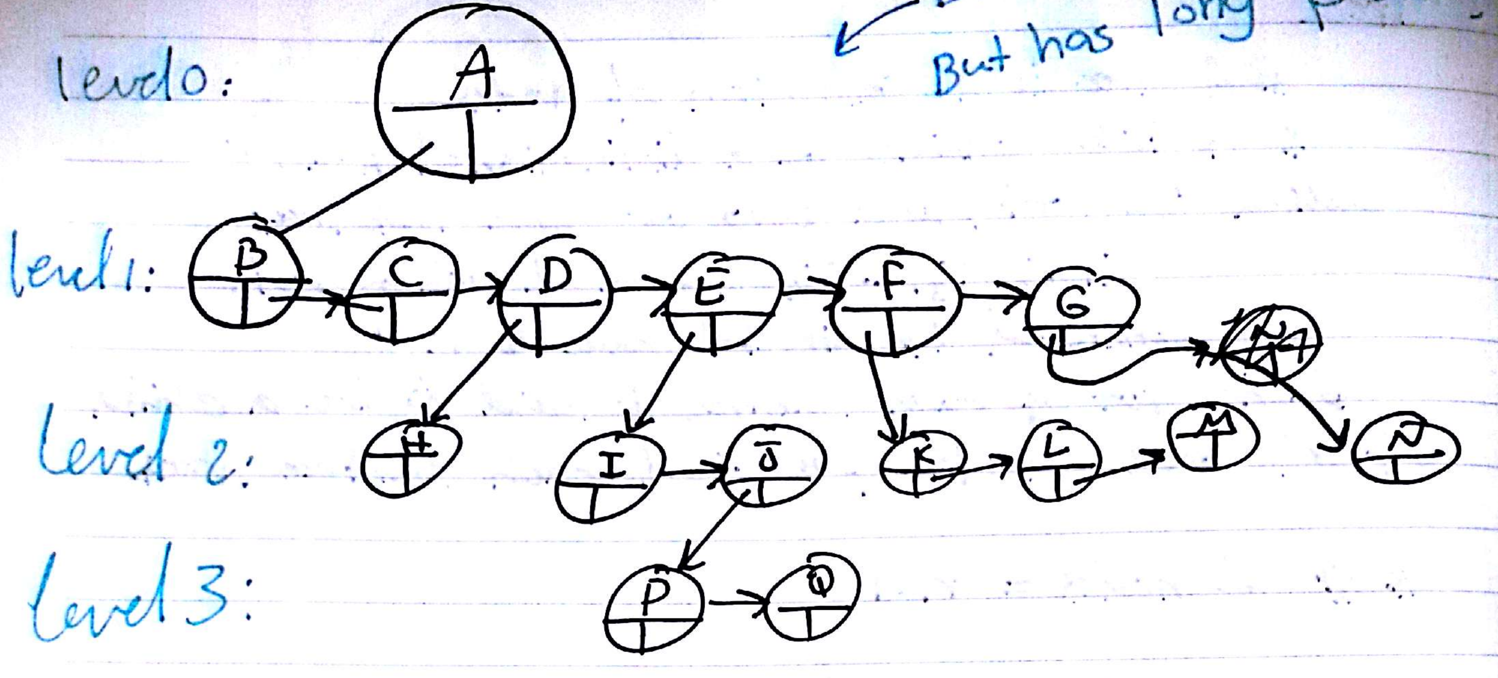
:

* H(B) = 0

leaf

longest path!

← Dynamic
But has long paths!



~~Tree Traversal~~ :-

reading a tree

root ←

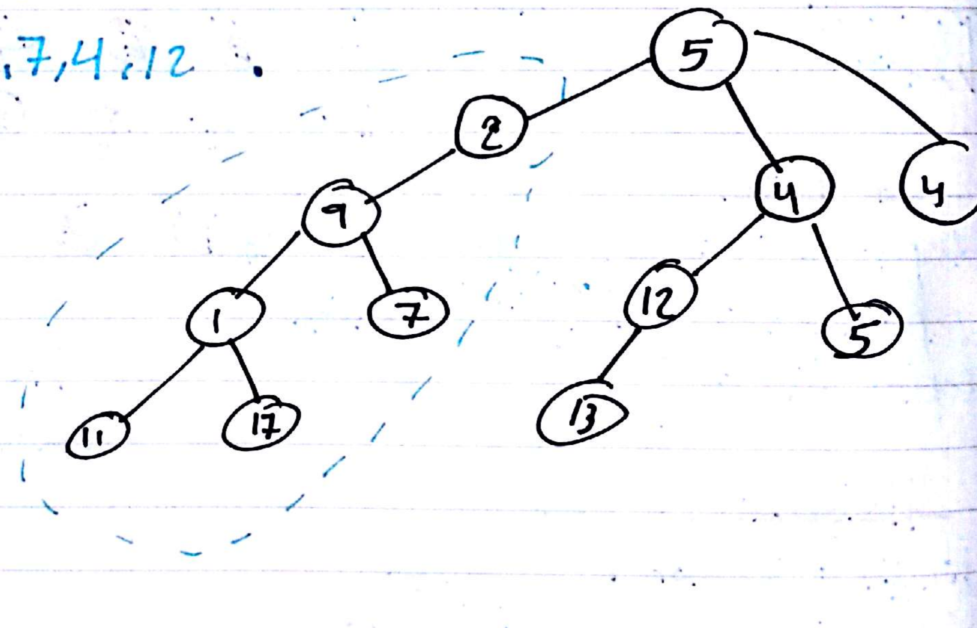
① pre order

always

root → left → Right

Root is the first value

① 5, 2, 9, 1, 11, 17, 7, 4, 12
13, 5, 4



* ② InOrder

left → Root → Right

11, 1, 17, 9, 7, 2, <u>5</u>, 13, 12, 4, 5, 4

(* If tree is balanced root in the middle)

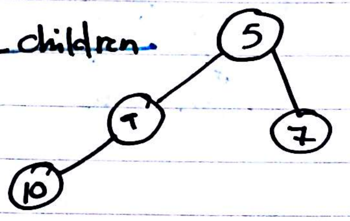
③ post order

left → Right → Root.

11, 17, 1, 7, 9 2, 13, 12, 5, 4, 4, 5 Root always

Binary Tree

* max = 2 children for a node.



Node
Object inf.
Node next.
Node right.

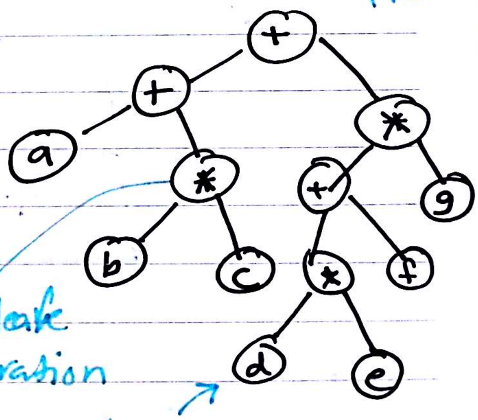
* Expression Tree:

- The length of an expression tree are operands, such as constants, variables names ....

any expression tree is a binary tree.

- The other Nodes contain operators.

- we use inorder traversal.



Ex: $(a+b*c)+((d*e+f)*g)$

→ In order

$a+b*c+d*e+f*g$

each none-leafe is an operation

→ pre order

$++a*bc*+*defg$

each leave is an operant

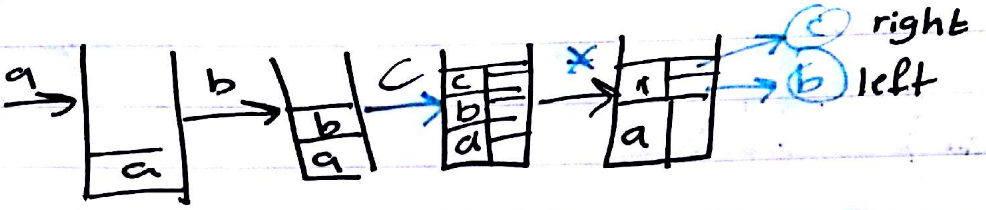
* (is read in-order!)

→ post order

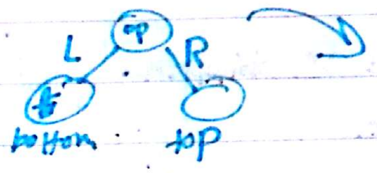
$abc*+de*f+g*+$

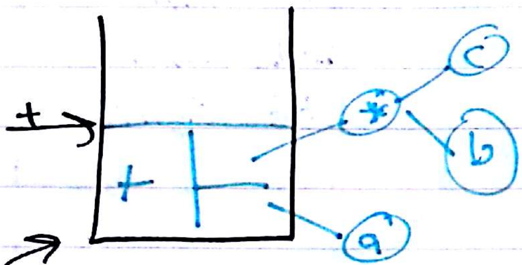
- How to build an Expression Tree

→ From postfix → Because we don't use Brackets!

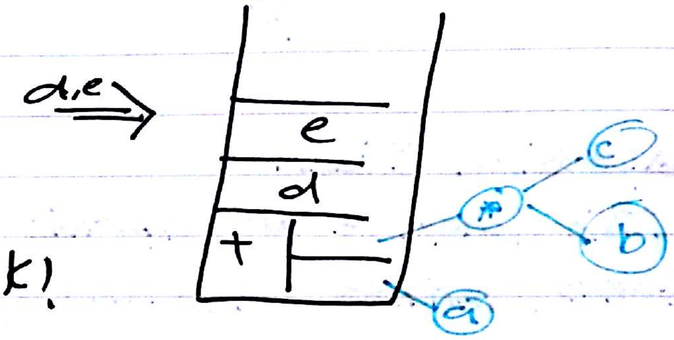


* when op → Pop 2 ele top → push

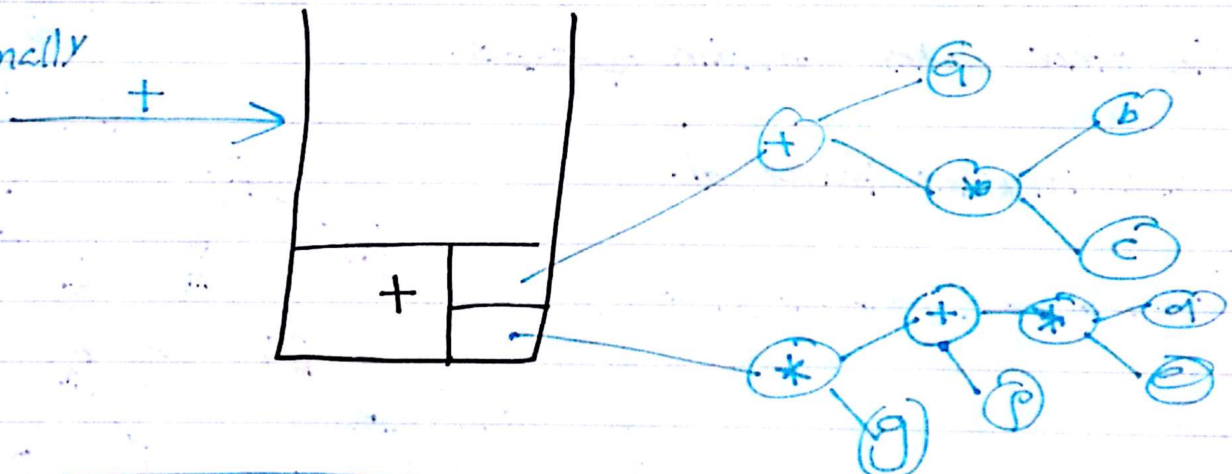


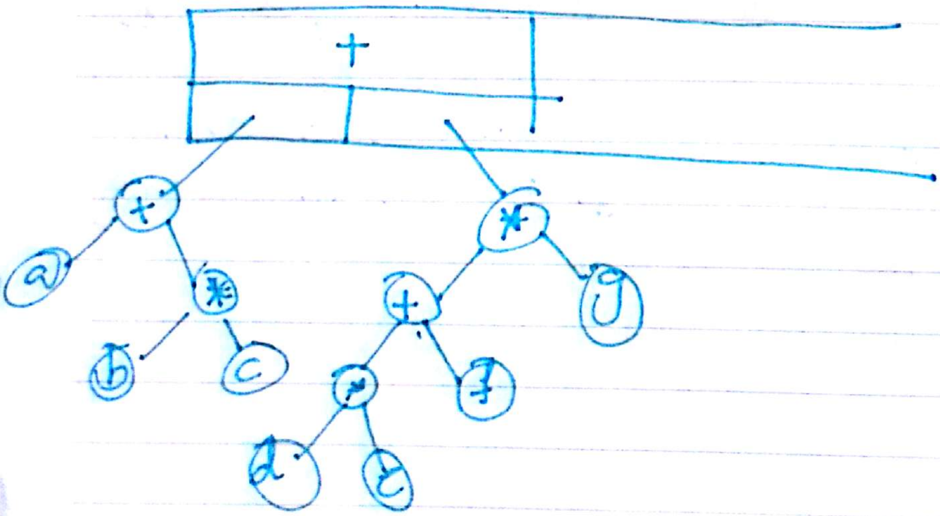


In this case one element is in the stack!



Finally





* Binary Tree

Node
info: Object left: Node right: Node
+Node(Object)

~~Binary Tree~~ BST
root : Node
+Binary Tree ( ) +find (object) :Node +Insert (Object) + Delete (object) + print inOrder () + preOrder () + postOrder () + IsEmpty ()

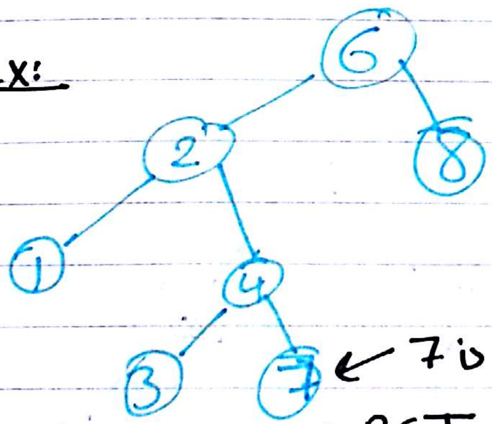
To insert elements :- we use

⇒ Binary Search Tree

Value (left) < Value (Root)

value (Right) > value (Root)

Ex:



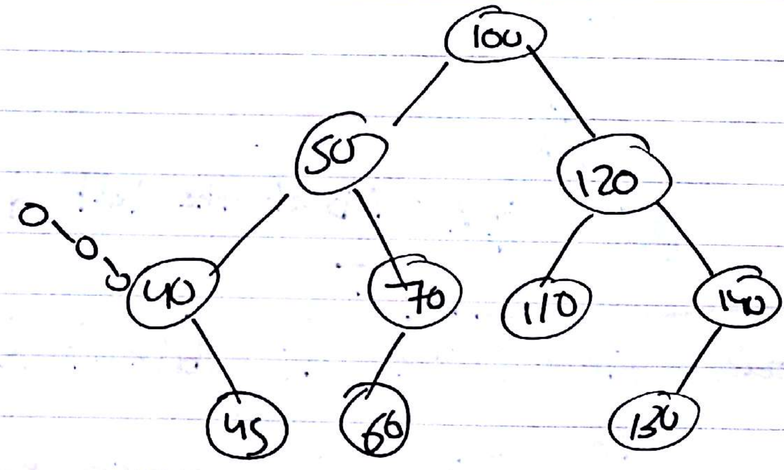
← 7 is at the left of 6

x BST

```
Node find (Node T, Object x) {

        if (T==null) return null;
    if (T. element
```

Time to find min
Best case const
Worst case O(n)
time O(n)



---

public Node findMin()
{ return findMin(root); }

private Node findMin (Node T)

if (T==null)
return null;

```
else  if (T.left == null)
          return ~~find~~ T;

    else  return ~~f~~ find Min(T);
```

Find

```
public    Node findMax(){

        Node T= root;
    if (T==null) return null;
    while(T.right!==null)
          T=T.right;
    return T;
          }
```

```
private Node insert (Node T, object x) {

        if(T==null)

        { ~~Node temp=new Node(x)~~
              T=new Node(x); }
        if (x<T.element){

            T.left = insert (T.left, x);
          else
            T.right= insert(T.right, x);

          return T;
```

doesn't work if Root =null

```
public    Node insert (Object x)
          root=insert (root,x); }
```

Delete:-

```
Node delete (object x, Node T) {

    Node child.
     if (T= = null)
          error;
    else
     if ( x <T.element )

          T.left= delete (x ,T.left);
    else if  (T>T.element)
          T.right= delete (x,T. right);


    else if (T.left && T.right) {                 or max(left)
            temp= find Min (T.right)
              T.element = temp.element;
               T.right = delete (T.element ,T.right);
         }
        else { if (T.left= =null)
                 Child= T.right;
            if Child= T.left;
            return Child;
                      }
          return T;
              }
```

delete(110, $\overset{100}{T}$)

↳ $\overset{100}{T}$.right = delete(110, $\overset{120}{T}$)

↳ $\overset{120}{T}$.left = delete(110, $\overset{110}{T}$)

{ temp = 113

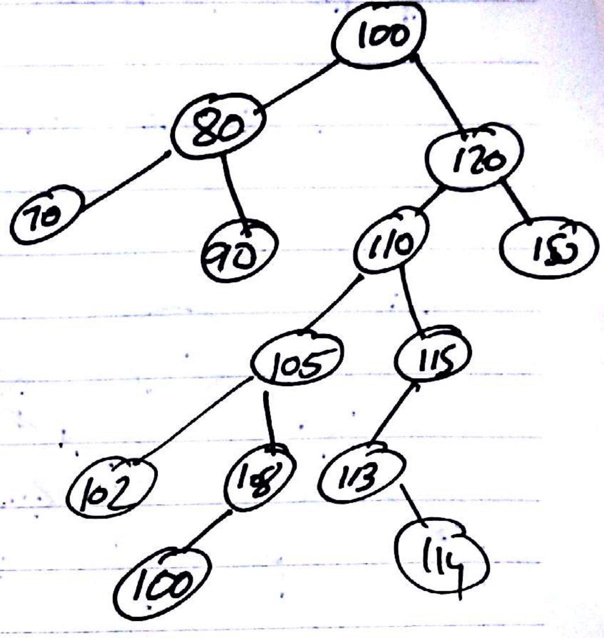
T.element = 113

T.right = delete (113, $\overset{115}{T}$)

↳ $\overset{115}{T}$.left = delete (113, $\overset{133}{T}$)

114

Child = 114



← in public

root = delete( )

---

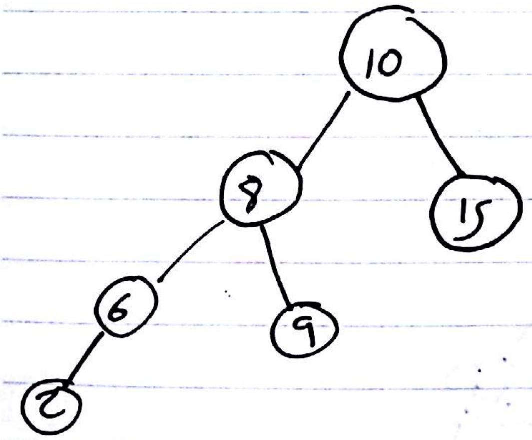
* AVL Tree :

BST

- AVL → Binary search tree with balance
- Balance ⇒ |height (left) − hight (right)| ≤ 1

↳ longest path from leave to Node)

tree
↓
Binary
↓
BST
↓
AVL



← Not AVL

→ Node 10 is a problem

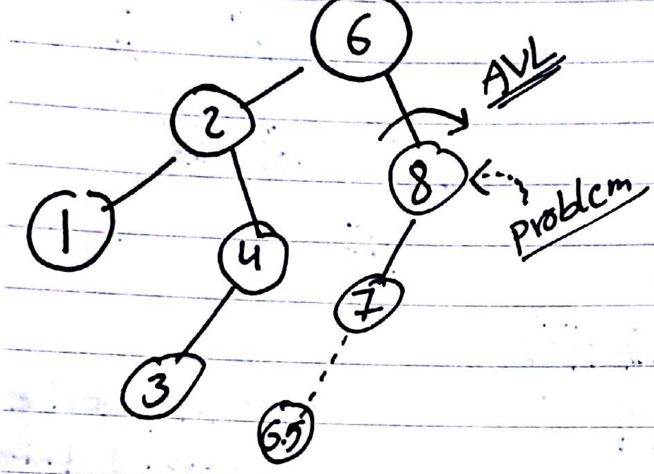
→ check each node

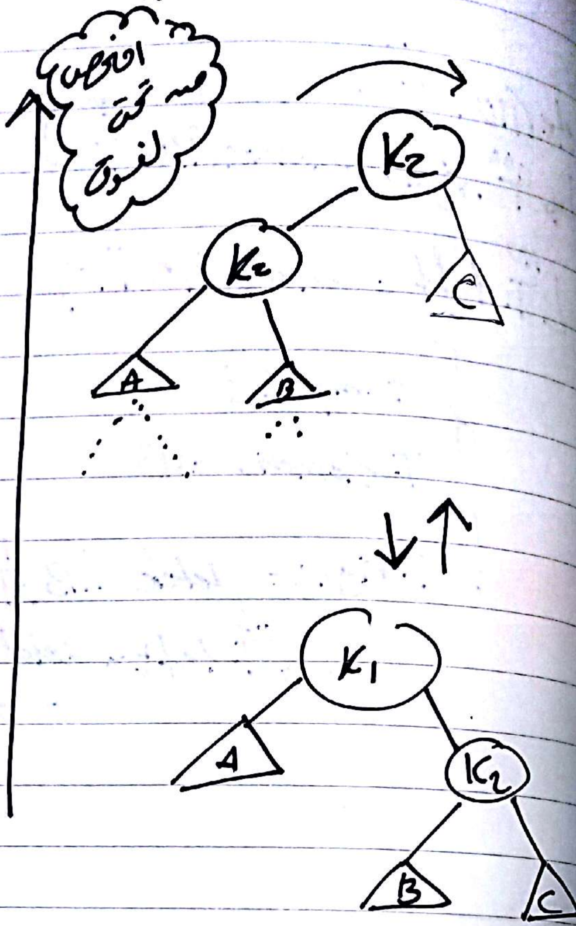
⇒ Search
Insert
Delete
} O (logn)

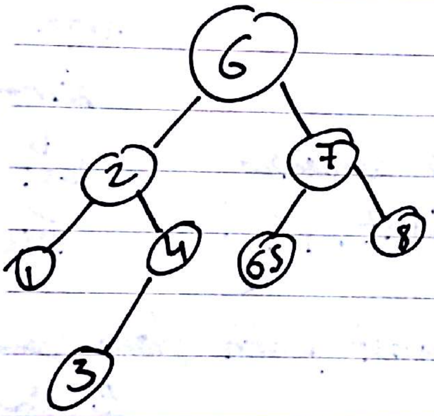
* How to make a tree balanced?

- Single rotate:-

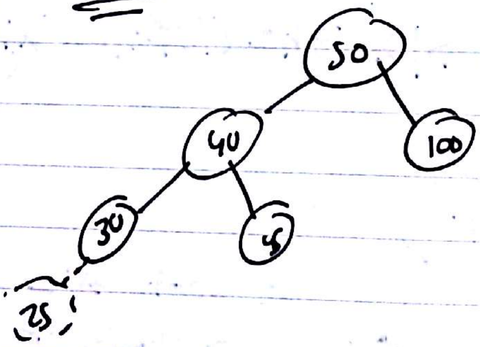
Ex:

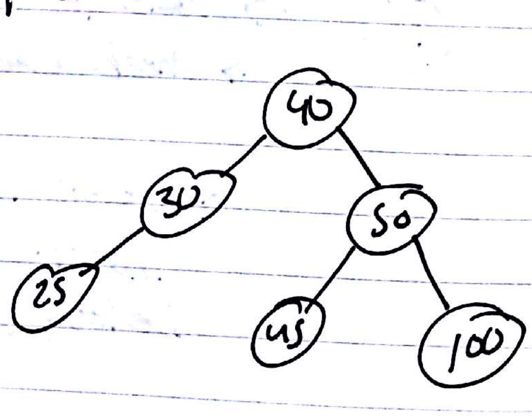


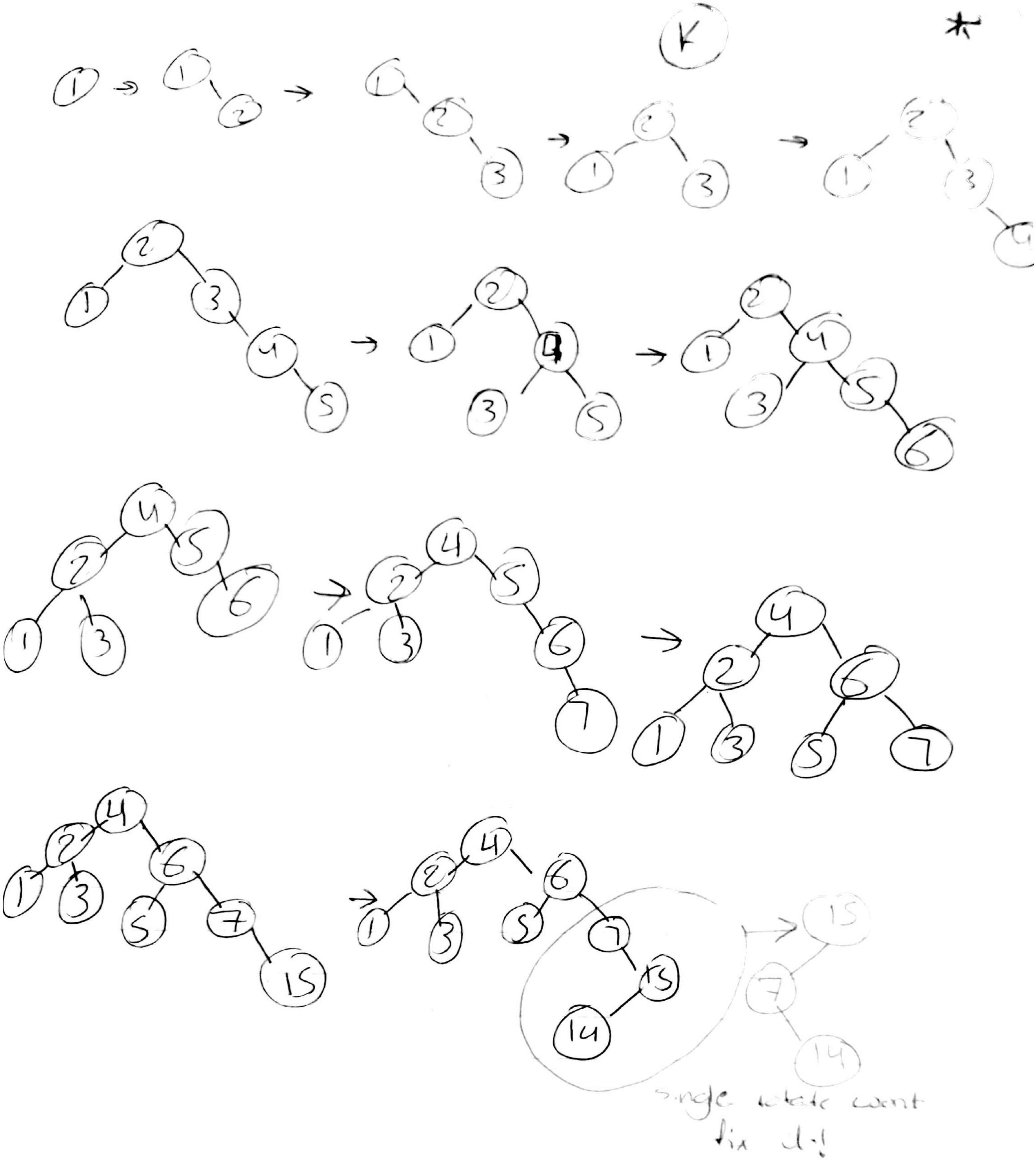




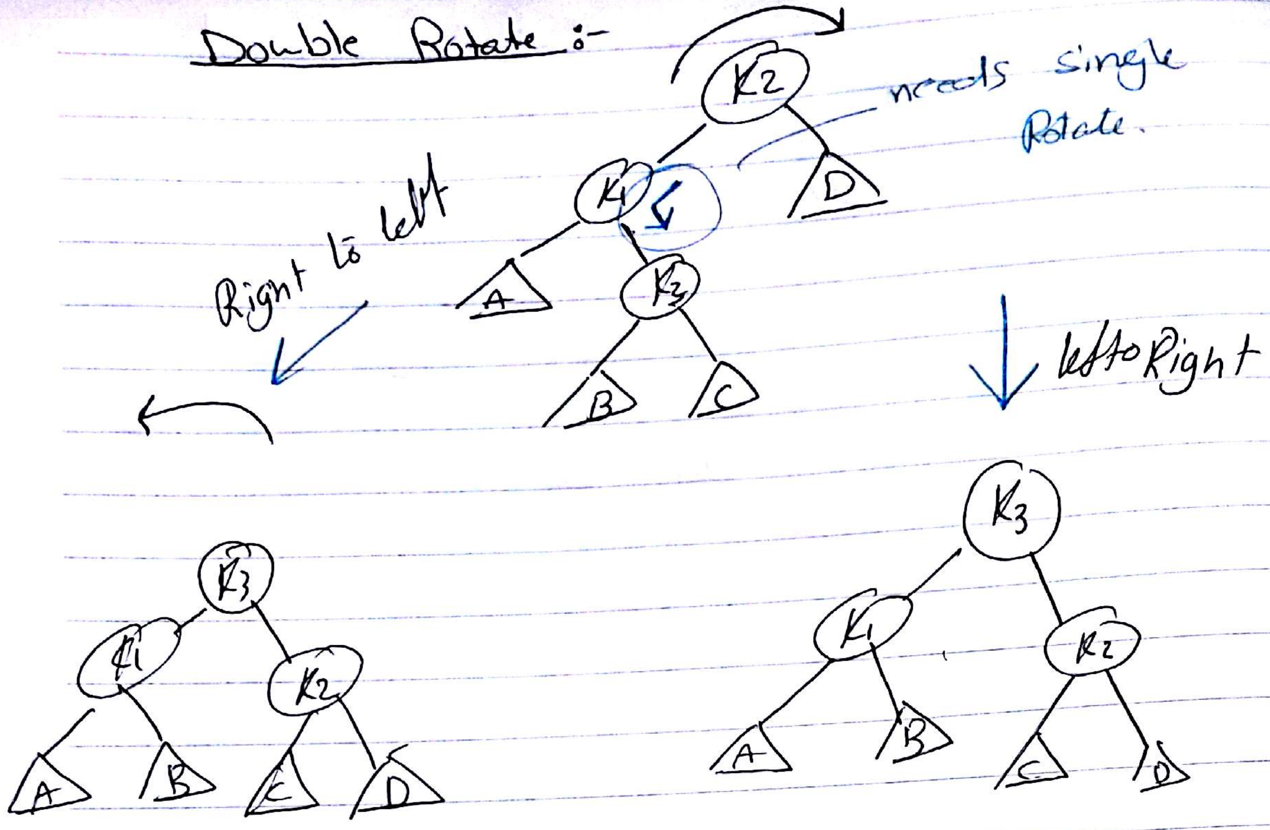
Ex



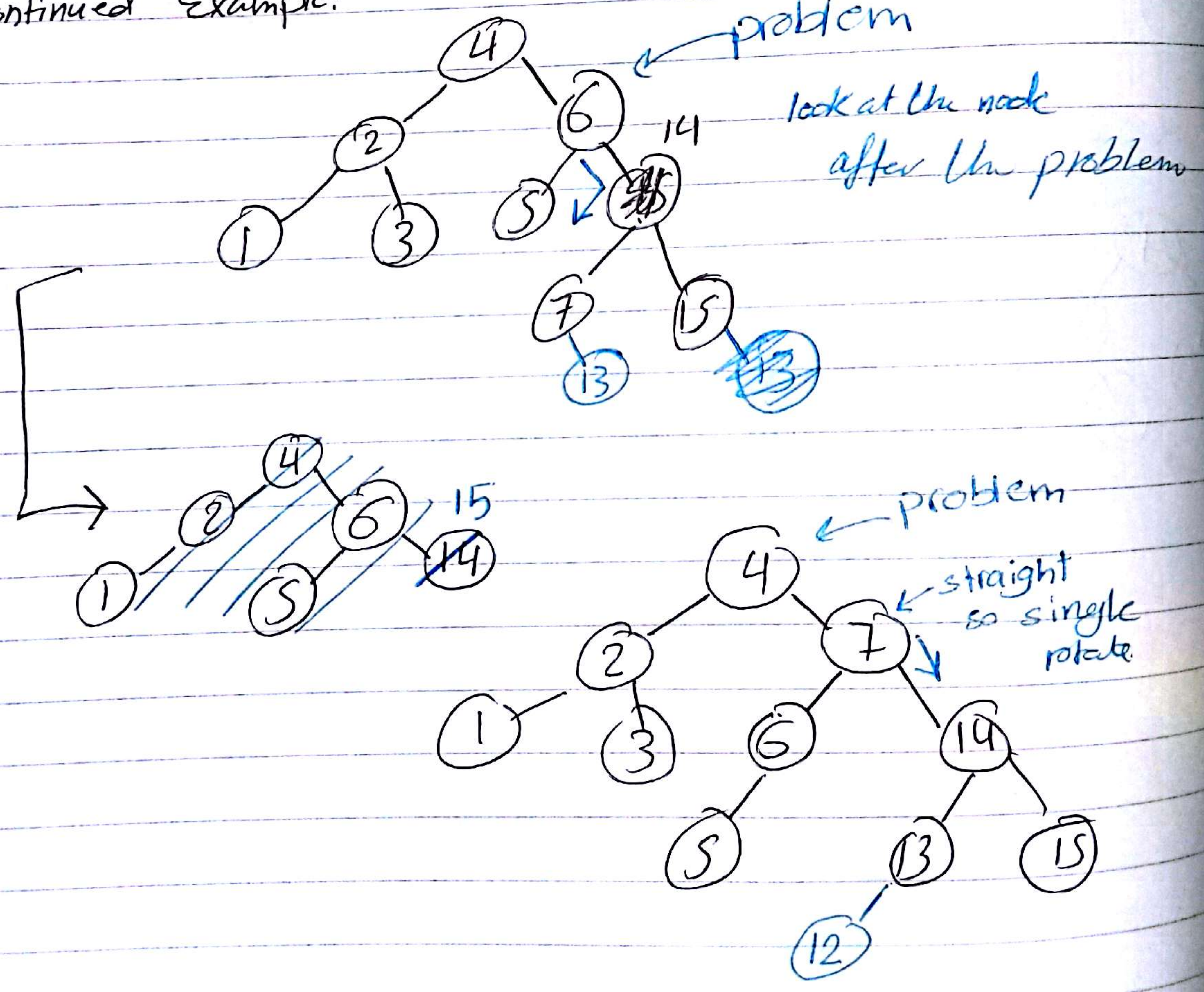


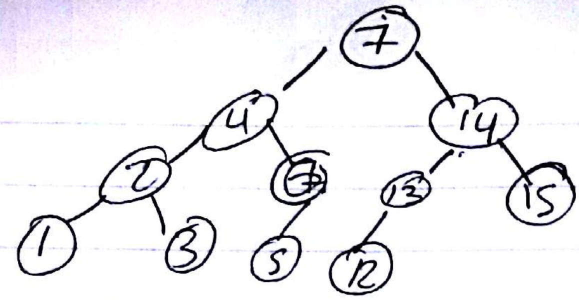

single rotate won't fix it!

Double Rotate :-



Continued Example:



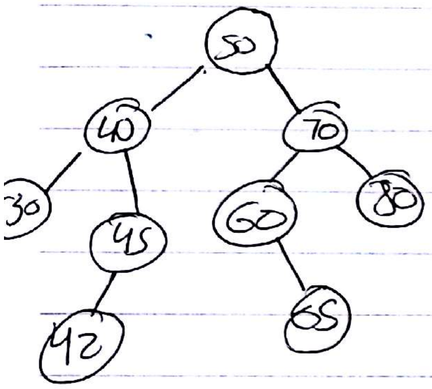


Best Case ⇒ Const
Worst Case ⇒ O(log n)
Average Case O(log n)

Node
info: Object left: Node Right: Node hight: int
+Node

hight for Node
max(hight left, hight right)

```
int getHeight (Node T) {
  if (T==null)
    return -1;
  else
    return (T.height)
}
```



```
int CalculateHeigh (Node T) {
  if (T==null)
    return -1;
  else {
    return (max(CalculateHeight(T.left), Calcula
      teHeight(T.Right)+1) ; }
```

CH (T) 50
↳ max(CH(T) 40, CH(T) 70)+1 → same thing we get 3
2 3

30 45
CH(T), CH(T)
0 1

-1 , -1   CH(T), CH(T)
0 -1

-1 -1

so CH(50) = 3+1 = 4

to improve the algorithm

```
{ if (T==null)
      return -1;
  else if (T.left, T.right)
    return max( ___
  else if (T.right)
      return Calcul(T.righ)+1
  else if (T.left) - - -
  else return 0;
```