Recurtion :-

* needs to stop somewhere.
* has a recurtion relation.
↳ fact(n) = n * fact(n-1)

$$fact(n) = \begin{cases} 1 & n=0 \\ n * fact(n-1) & n>0 \end{cases}$$
(1 ← stop; n * fact(n-1) ← recurtion relation)

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

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

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

---

* Character takes
before [0-127 E 8 bits
128-255 other
↳ then 2 bytes

*Review var.

→ *In the main :-

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

---

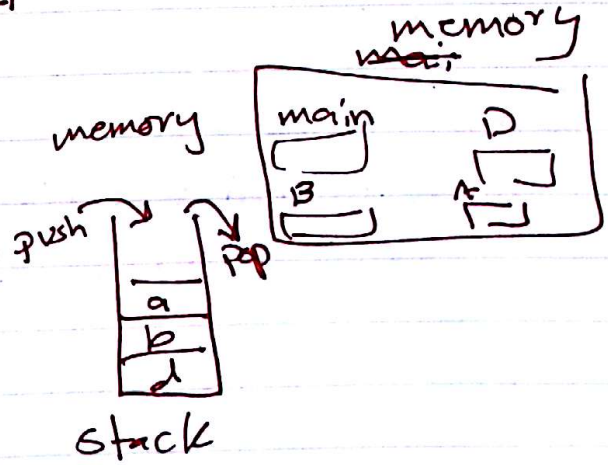
*As a programmer recurtion has no disadvantages.

call ~~fun~~ 3

main
↳ call D

func. D → B → A (d, b, a)

a → when it calls A()



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

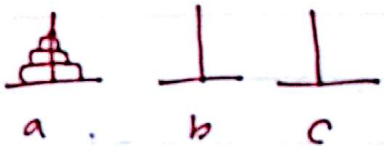
* at Recurtion :

→ 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!) ⇒ it 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

$$Hanoi(n, \overset{s}{a}, b, \overset{p}{c}) = \begin{cases} Hanoi(n-1, a, c, b) & n>0 \rightarrow \text{*one at a time.} \\ a \rightarrow c & \text{* small ones above.} \\ Hanoi(n-1, b, a, c) \end{cases}$$

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 it value
x++ ← never changes x
x+1 ← x is kept as a value and incremented

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

# of moves = $2^n - 1$

Ch. 2 Algorithm Analysis

CPU → 1 GHz → ↖ space ↘ time

* For huge Data Algorithm analysis is important.

*Reveiw sorting!

-in Algorithm 10/10

$n^2 > 1000n$

• Running time calculation:

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

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

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

not accurate time

* 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 working during each other

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

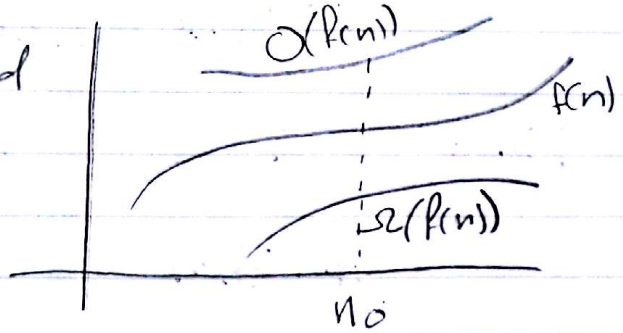
* If $O = \Omega$ → stable Algorithm we get Θ.

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<i*i, 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 → a 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
متغير الدخل

$T(n) = c + T(n-1)$

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

$T(2) = c + T(1)$

$= nc + d \rightarrow T(n) = O(n).$

But

```
fact = 1;
for (i=0; i<=n ; i++)
    fact *= i.
```

$T(n) = O(n)$ ← loops in for one better though n!

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

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

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

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

⋮

get the parameter to 1

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

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

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

$d n + n \log n$

$O(n \log n)$

Insertion Sort :

times		Algorithm / if Code
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] ;
		i = i - 1 ;
$\sum_{j=2}^{n} T(j-1)$	6	end while
$n-1$	7	A[i+1] = key ;
		end for.

last turn ask for ≠ don't enter

for N

if(loop) = n
T_5 statements in loop = n-1

$T = T_2 + T_3 + T_4$ ← each card has a time.

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

i	j	key
1	2	7
1	2	2
⋮		
5	6	8

A

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

5	7				
2	5	7			
2	4	5	7		
2	4	5	7	8	10

Time Analysis

* Best Case / Worst Case when data is sorted / reversed

كأنو تالتة للحلقة

Best case → $T(n) = C_1 n + C_2(n-1) + C_3(n-1) + C_4(n-1) + C_5(0) + C_6(0) + C_7(n-1) = O(n)$

worst case

~~C4~~ 4 → as if `for (i=0; i<n; i++)` n

`for (j=i; j>0; j--)` n

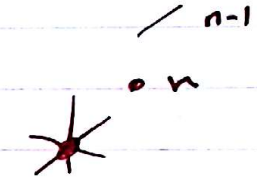
n^2

Average case → Random

$$C_1 A(n) + C_2(n-1) + C_3(n-1) + C_4 \sum_{j=2}^{n} t_j + C_5 \sum_{j=2}^{n} t_j - 1 + C_6 \sum_{j=2}^{n} t_j - 1 + C_7(n-1).$$

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

طريقة التحليل



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

$$\vdots$$

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

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

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

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

$$T(n) = O(n)$$

Linked List, Stack, Queue.

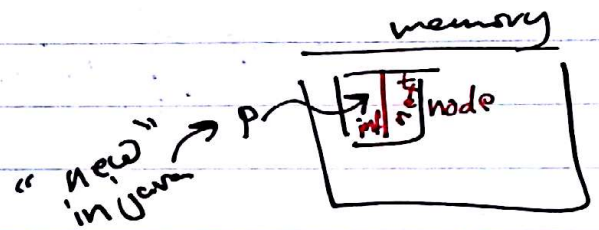
*Reading point

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

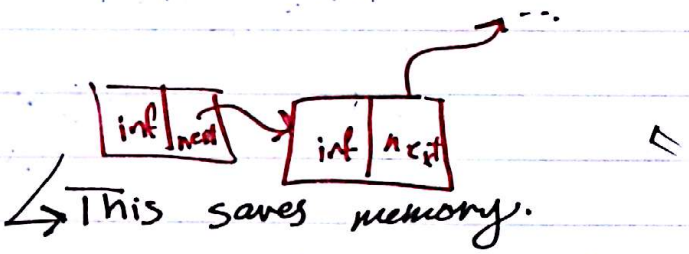
- operations on lists:
  - * print element
  - * print list
  - * Insert → Exp
  - * Delete
  - * Search
  - * make null.

* Array:
  - → fixed length (-)
  - → stored in series

A[i] = Address = A + (i-1) * # type

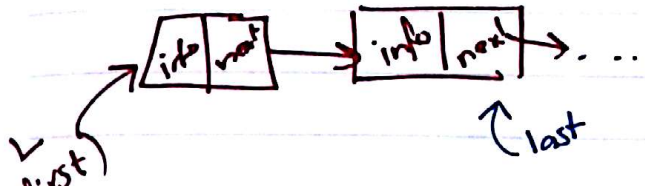


in C → p=(int)malloc(signature(int



	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
element: object next: Node
+ Node (object)

Problem in procedural Language

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

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

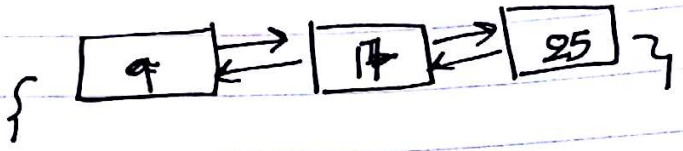
Code:

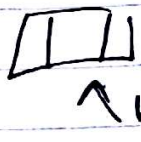
```
public class Node {
    Object element;
    Node next;
    public Node (object x)
    { .element =X
    }
}

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

Constructor has no type.

Double LinkedList :

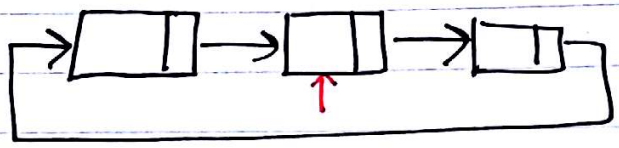




adding

```
temp.next = p.next;
temp.previous = p;
p.next = temp;
temp.next.previous = temp;
```

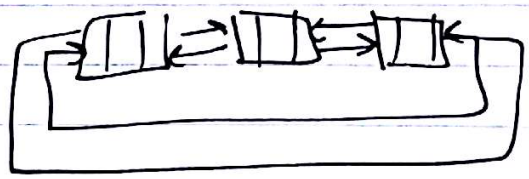
Circular Linked List :



← when printing use another pointer.

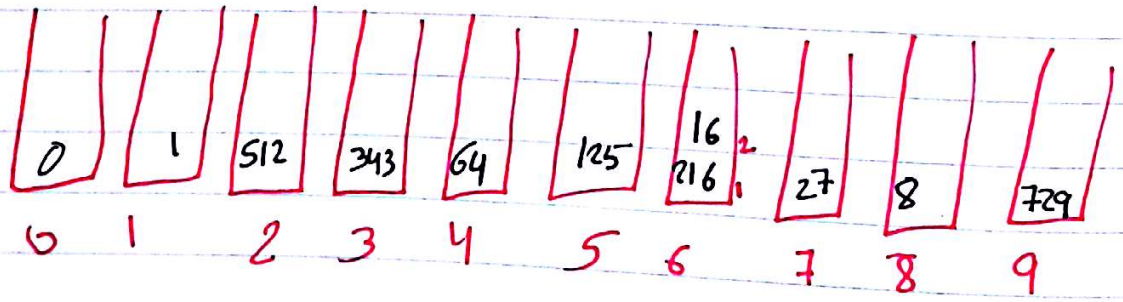
* has no header
* has no first nor last

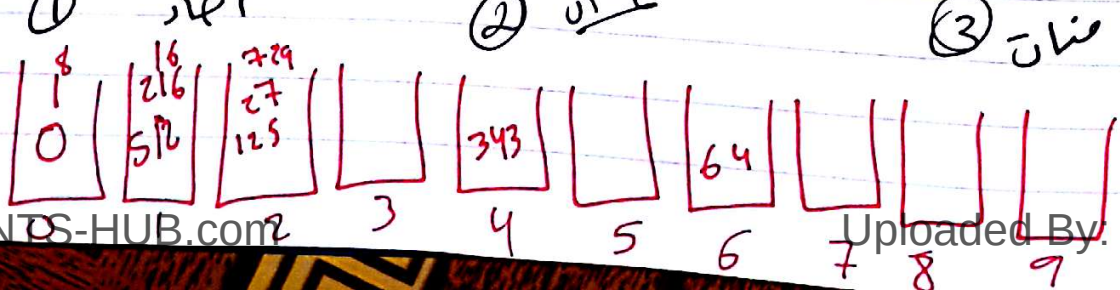
Circular double Linked list :

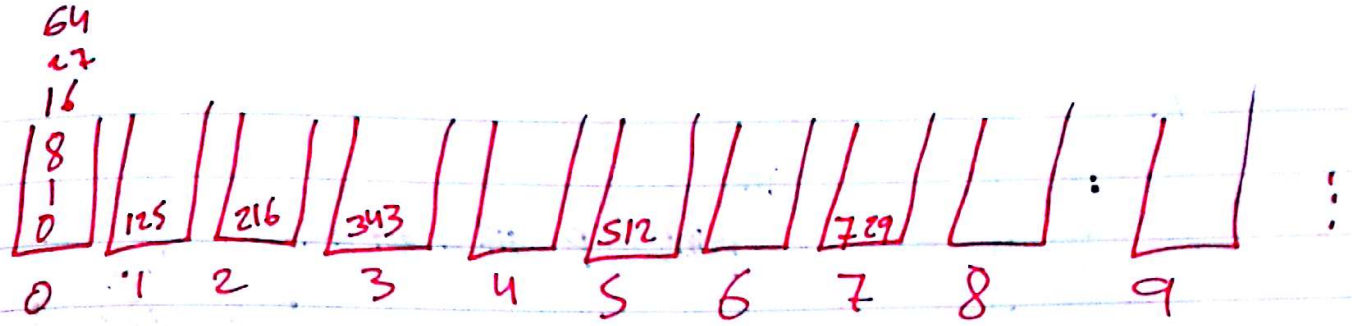


Radix Sort : 64, 8, 216, 512, 27, 729, 0, 1, 343, 125, 16

max = 729 O(n) ⇒ 3 digits

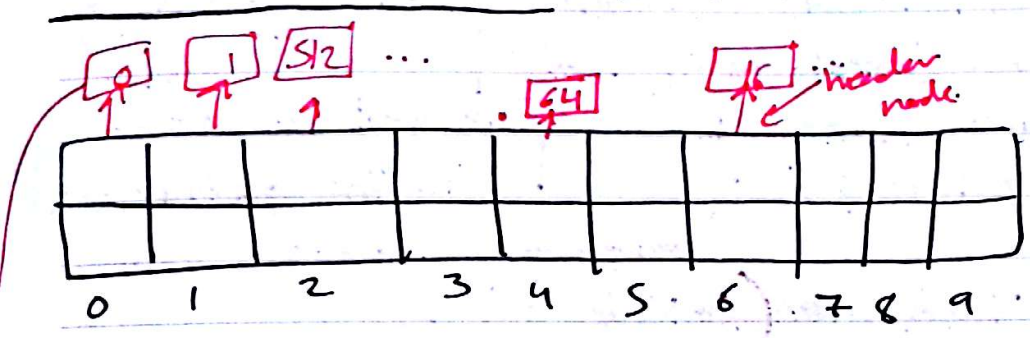






time = O(n) → n from max + (Digits use)(n)

↑ Best time for sorting



if $n = 10^5$ ← 10 buckets

space = 10^6 unit X 2 ↳ using

$= 8 \times 10^6$ bytes

= 8M bytes ↳ 2 Buckets sets at a time.

disaster!

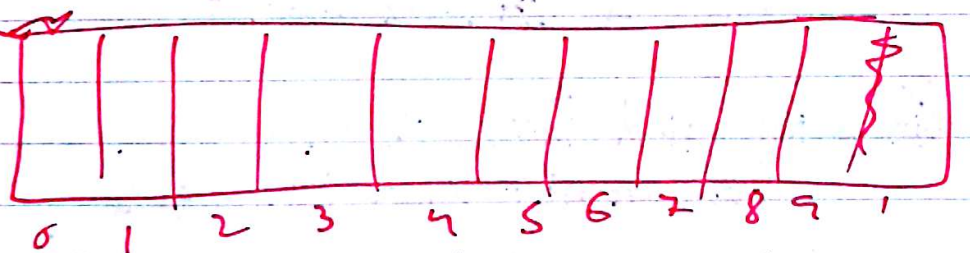
so we use array list

Space = ~~10×10~~ 5 n units

$= 4 \times 10^5$ * int

400k byte

step ②

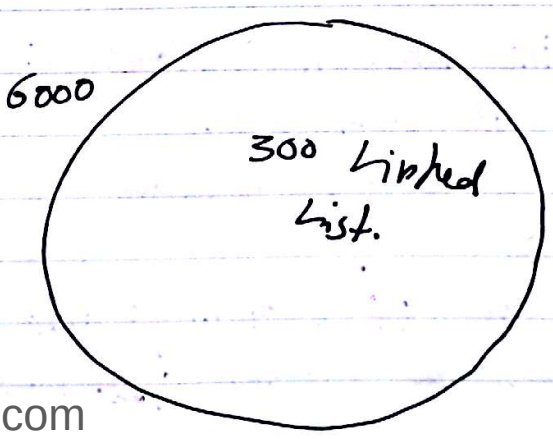


Curser Implementation :-

300 course ⇒ 45 students → 13500

3000 students → 6 hours → 6000

60



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

Algorithm :-

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

empty when 0.next=0

Node
info : Object
nat : int
+ Node(Object)

Curser : Array[max size of Nodes].

we creat the array in the constructer.

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

header node

```
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; }
```

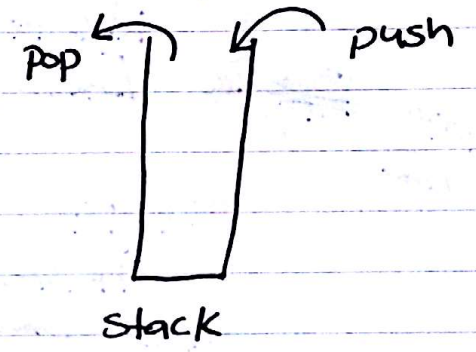
```
void insert (List L, int p , Object)

   int temp=new Node( );
    if (temp==0)
       "out of memory"
  else {
  Cursor [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(cursor[p].next != 0){
        int t = cur
        cursor[p].next = cursor[cursor[p.next]].next;}
int findpreivous(int L, int x){
    int p = L;
    while(cursor[p].next != 0 && cursor[p.next].info!=
        p = cursor[p].next;
    return p;}
```

Stack:



*First In Last out.
Last In First Out.

Functions:

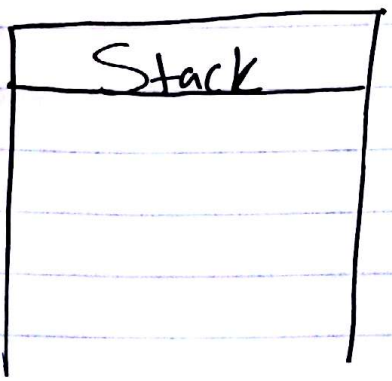
- push
- pop
- empty
- top.

only !

stack → Linked List
stack → Array

Stack is an object

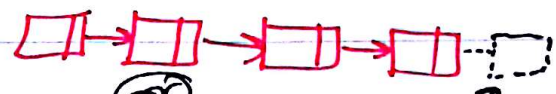
push (Lis



void

s. push

→ class stack ←



c push. → add first
c pop → remove first

push → add last
pop → remove last

C O(n)

for me better!

```
void push(object x) {
    addFirst(x); }
void pop() {
    removeFirst(); }
Object top() {
    return getFirst(); }
```

private (utility functions)

private functions

If array
use a pointer

```
stackpointer = 0
void push(object x)
    { if (stackpointer++ == size)
        S[stackpointer++] = x; }
```

← error

Array now

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

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

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

5*2 infix

*52 prefix

52* postfix

Ex: a+b*c+(d*c+f)*g → abc*+de*f+g*+

(infix ↙ ; postfix ↙ ; operance ↗)

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 *

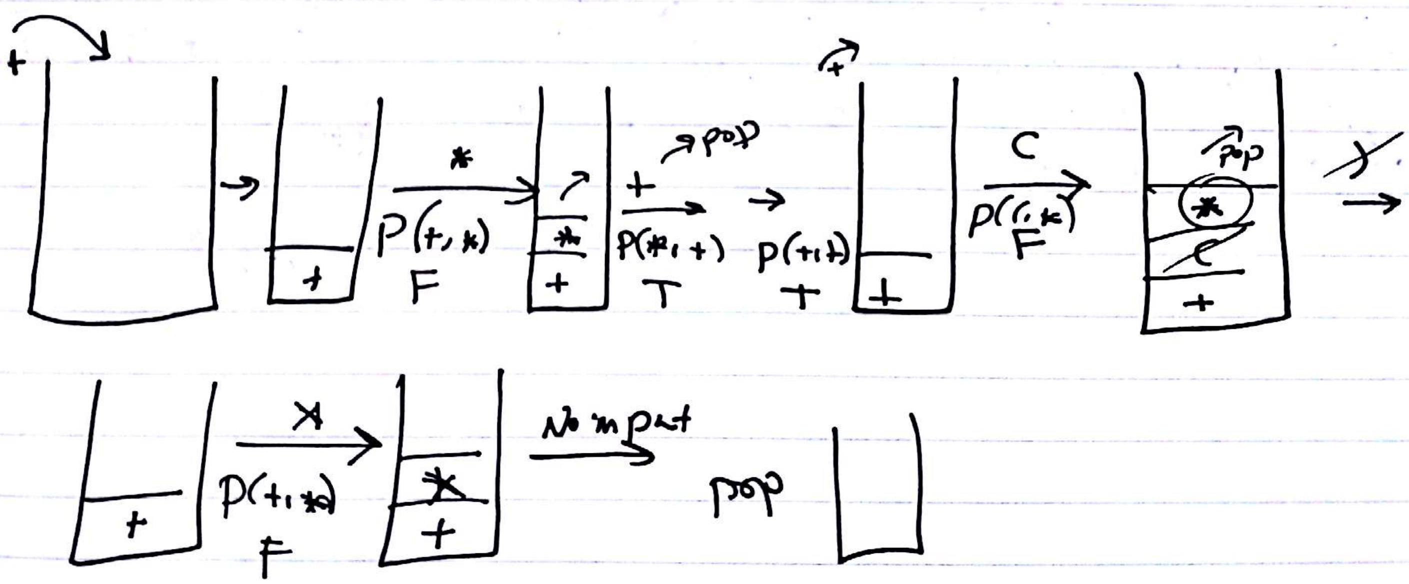
pred(')', '(') → Error (no operation)   (5)(7)

---

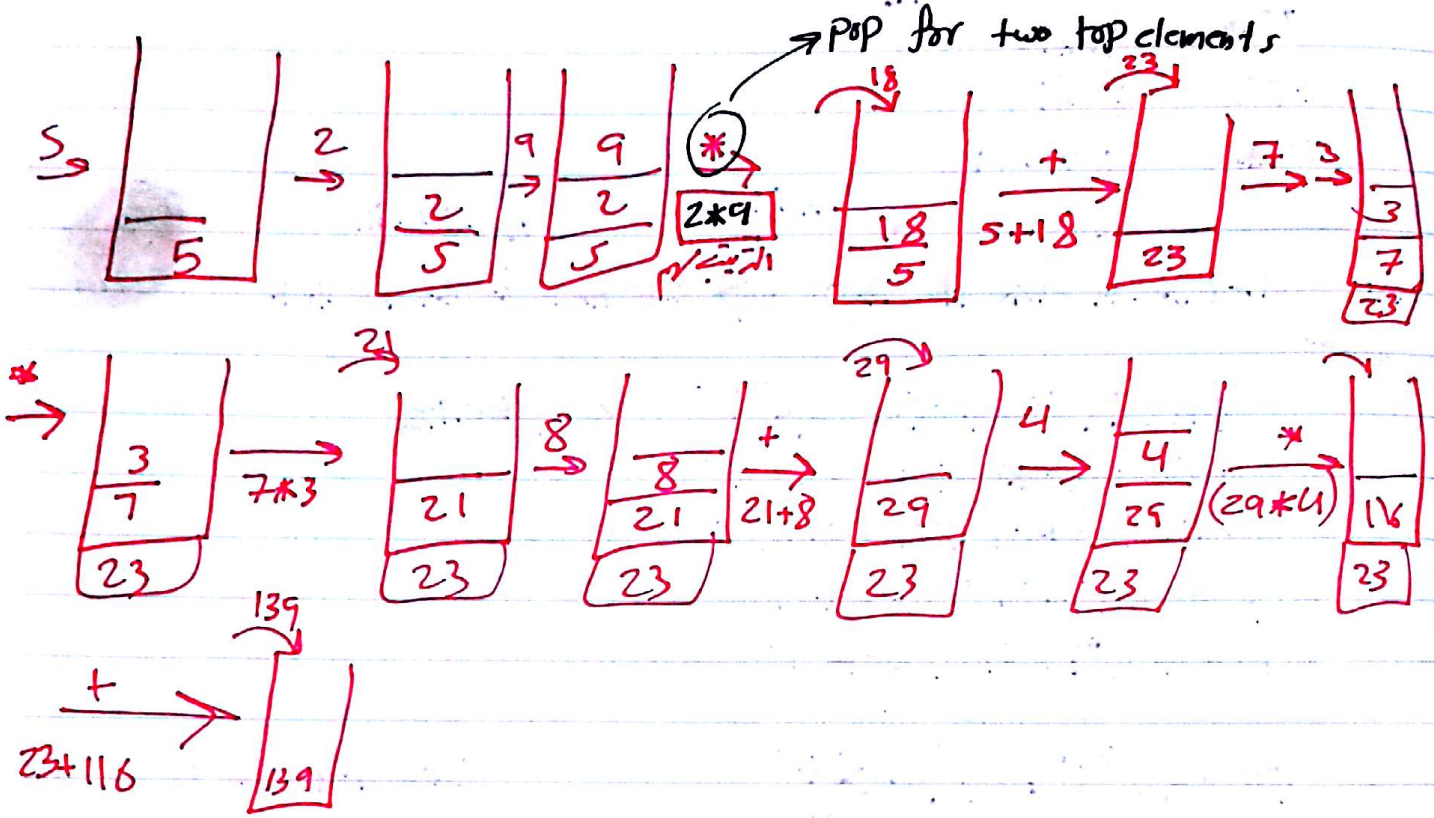
a+b*c+(d*c+f)*g

abc*+de*f+g*+

↑ output



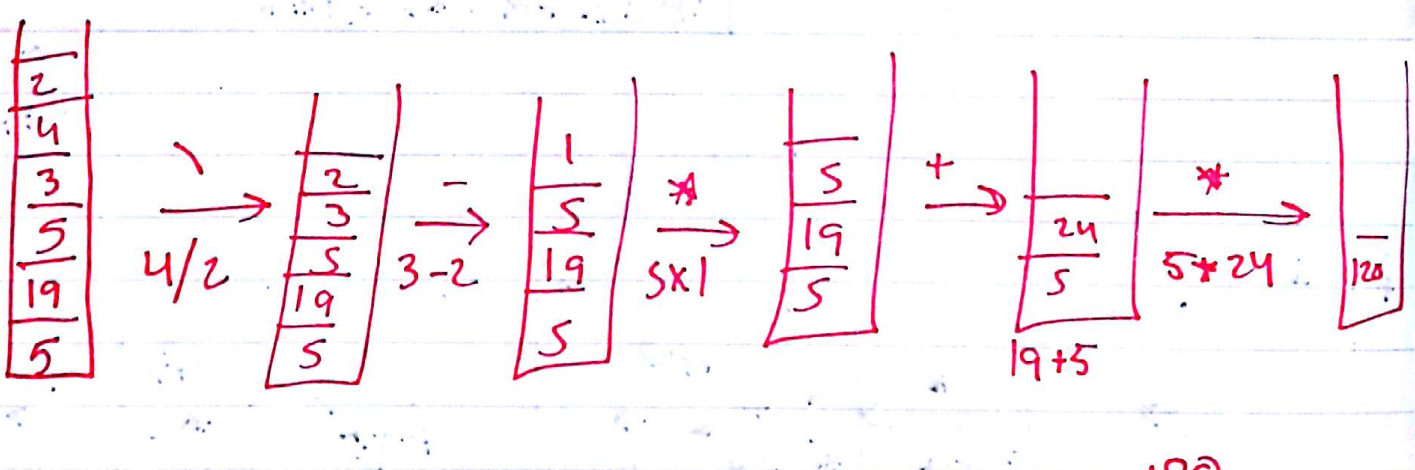
abc * + de * f + g * +



---

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

5, 19, 5, 3, 4, 2, /, -, *, +, *



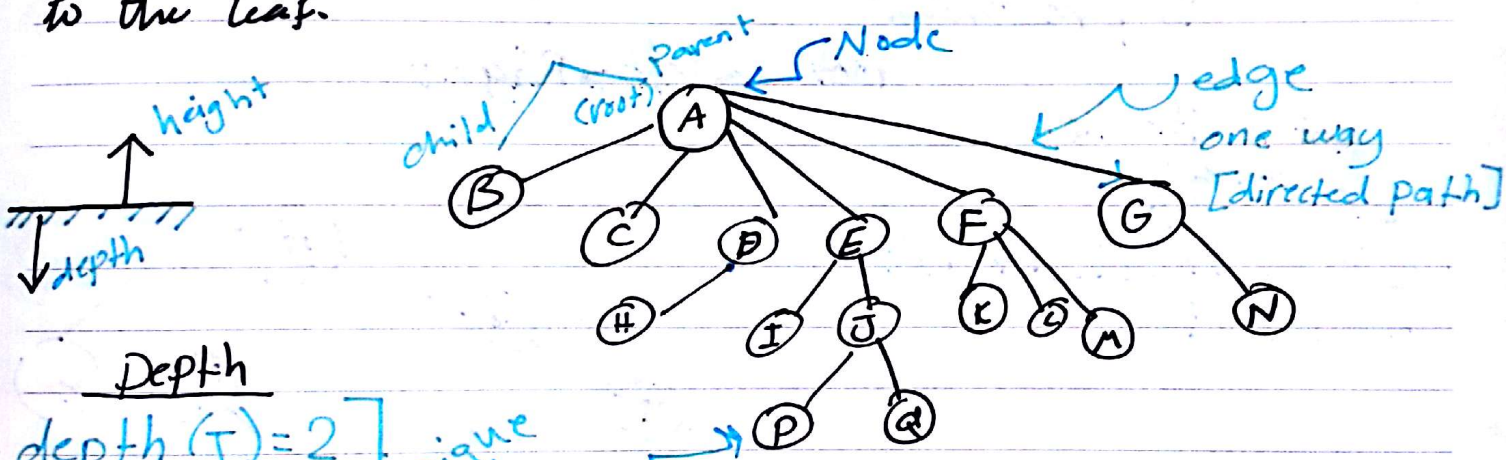
= 120

Tree

* A tree is a collection of nodes
* A tree consists of a distinguished node r, called the root, and zero or more subtrees $T_1, T_2, T_3, \ldots T_k$ each of whose 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 height of r; is the length path from r, to the leaf.



Depth

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

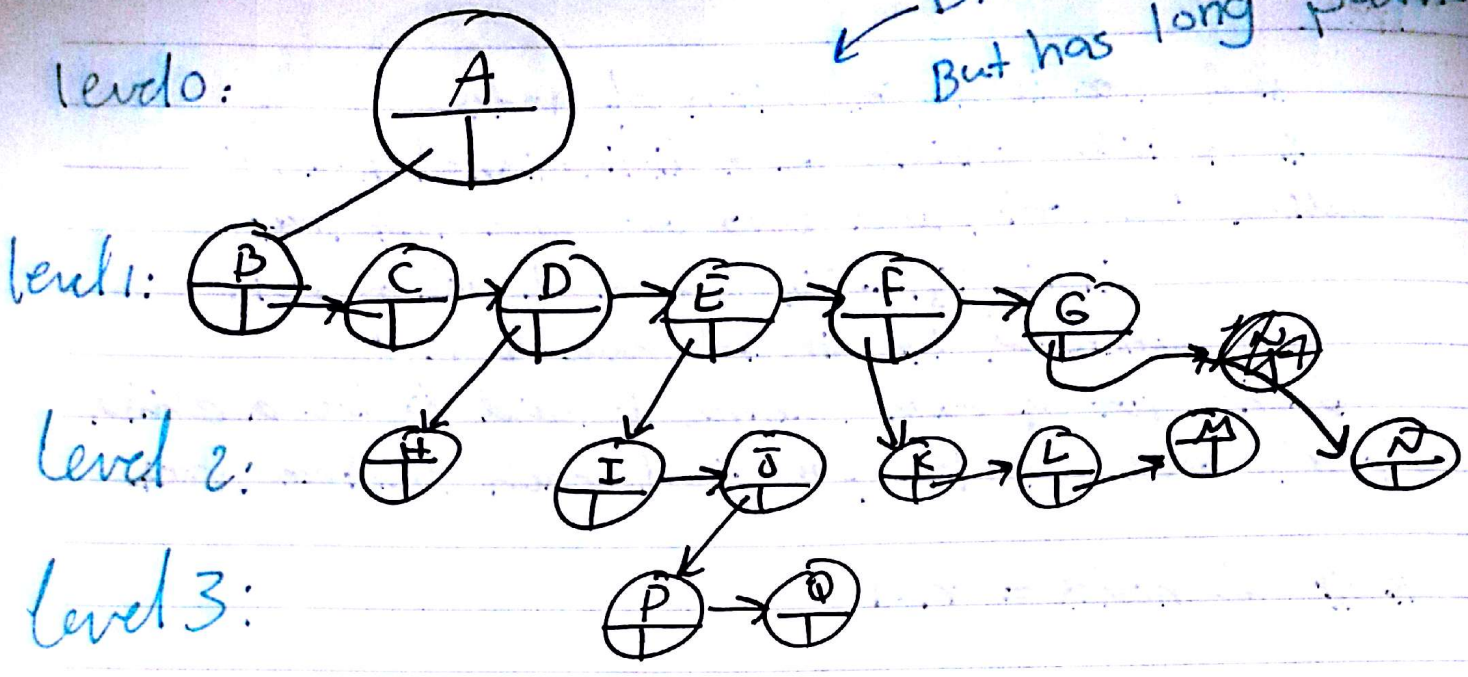
unique path

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!

---

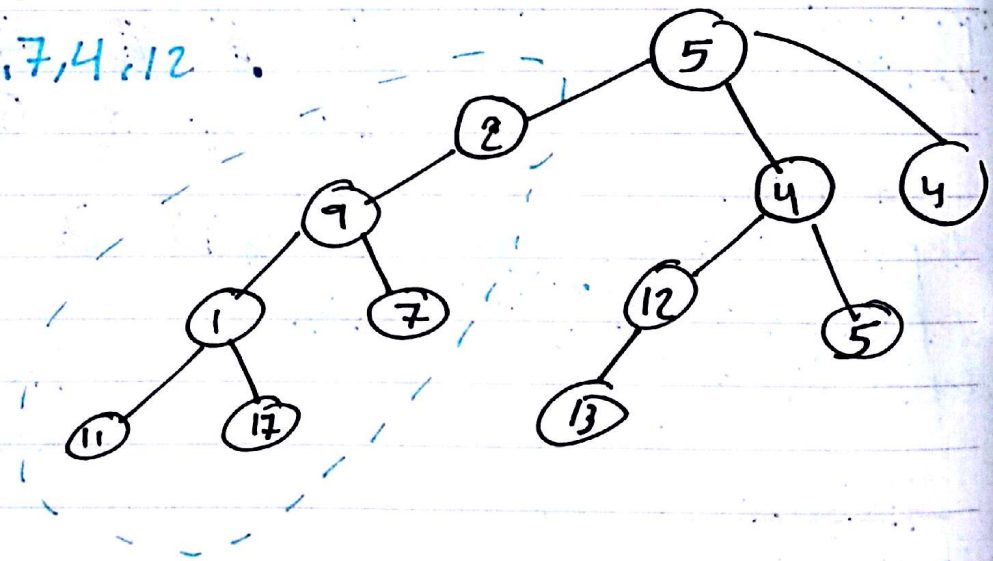
* Tree Traversal :- reading a tree

① pre order (root) 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, 5, 13, 12, 4, 5, 4

(* If tree is balanced root in the middle)

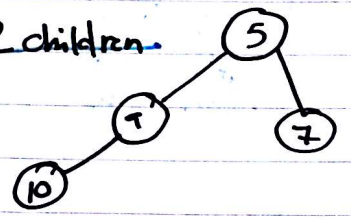
③ post Order

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
Objet inf.
Node next.
Node right.

* Expression Tree :

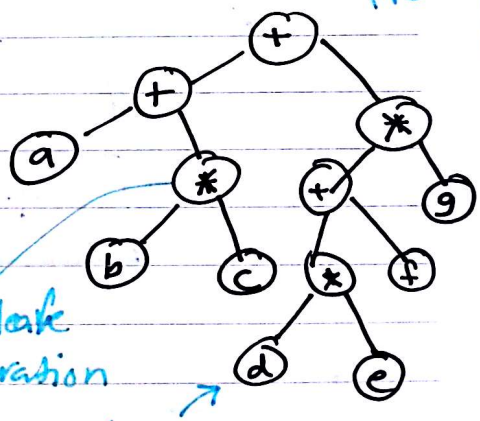
- The length of an expression tree are operands, such as constants, variables names ....

- The other Nodes contain operators.

- we use inorder traversal.

any expression tree is a binary tree.

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



each none-leafe is an operation

each leave is an operant

* (is read in-order!)

→ In order

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

→ pre order

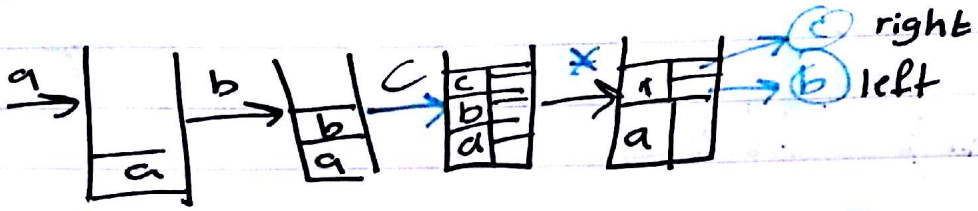
+ + a * bc * + * def g

→ post order

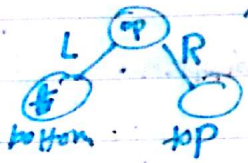
abc * + de * f + g * +

- How to build an Expression Tree

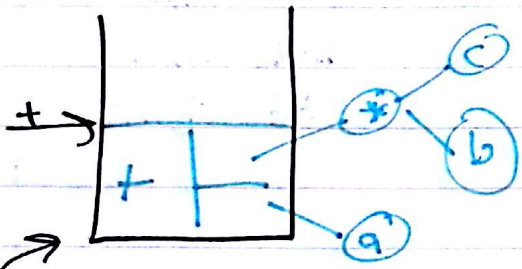
→ From postfix → Because we don't use Brackets!



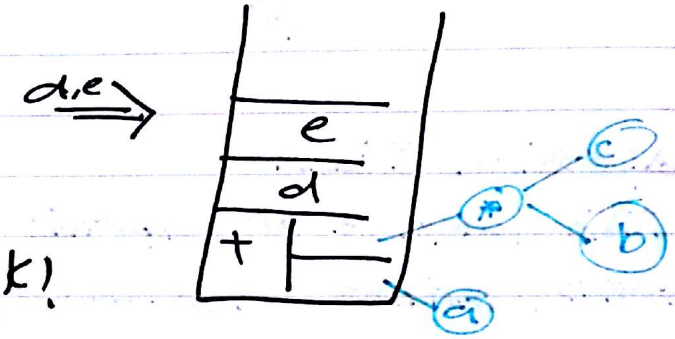
* when op → pop 2 ele → top



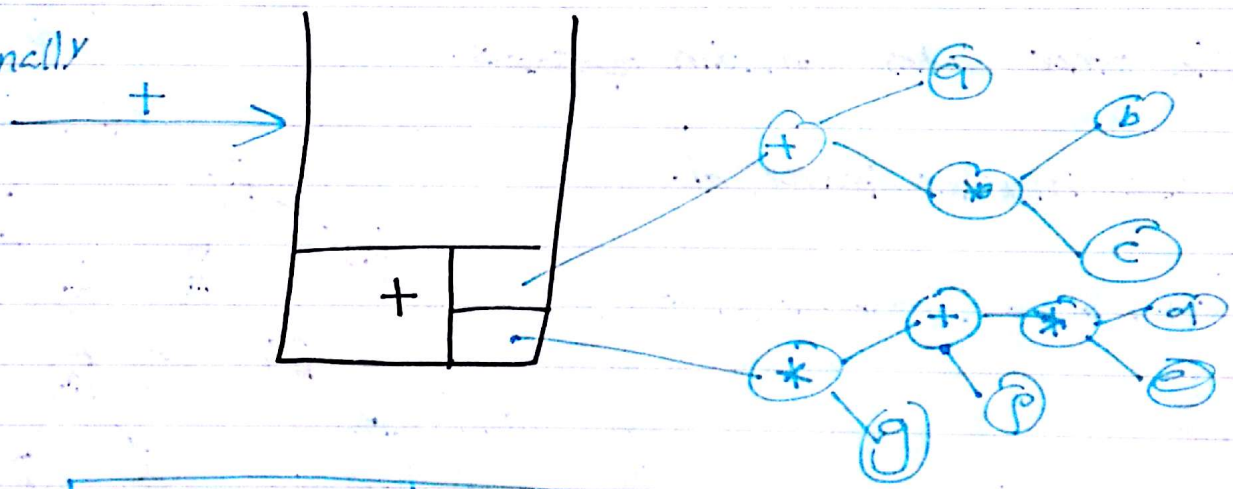
→ push.



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 ()

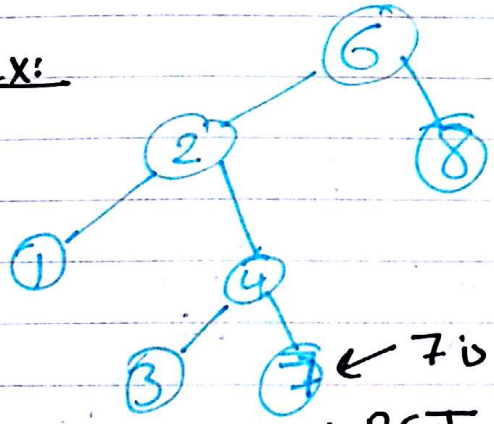
To insert elements :- we use

⇒ Binary Search Tree

value (left) < value (Root)

value (Right) ≥ value (Root)

Ex:

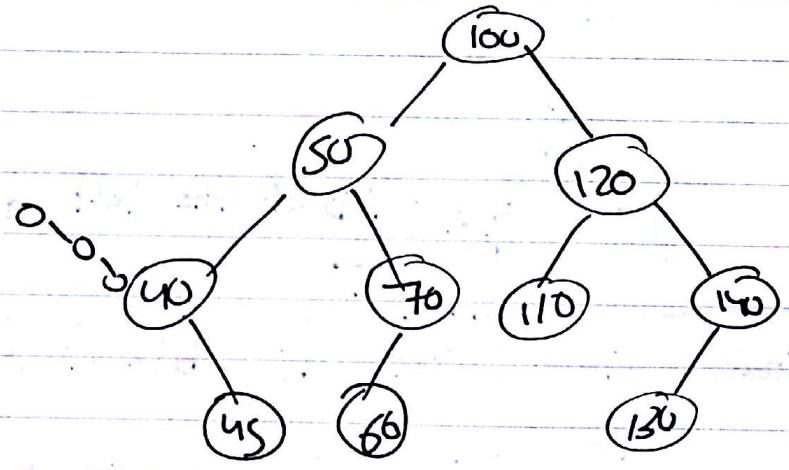


← 7 is at the left of 6

xBST

```
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)
↑ max. v



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

private Node findMin (Node T)
    if (T==null)
        return null;
```

```
else if (T.left == null)
        return T;
else return findMin(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)
    {
        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, T^{100})

↳ T^{100}.right = delete(114, T^{120})

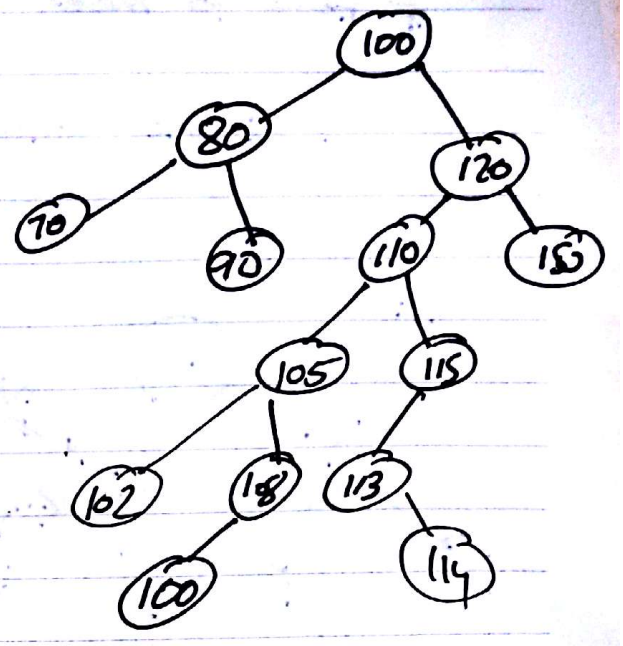
T^{120}.left = delete(110, T^{110})

- temp = 113
- T.element = 113
- T.right = delete(113, T^{115})

↳ T^{115}.left = delete(113, T^{133})

114

Child ∈ 114



∈ in public
root = delete( )

---

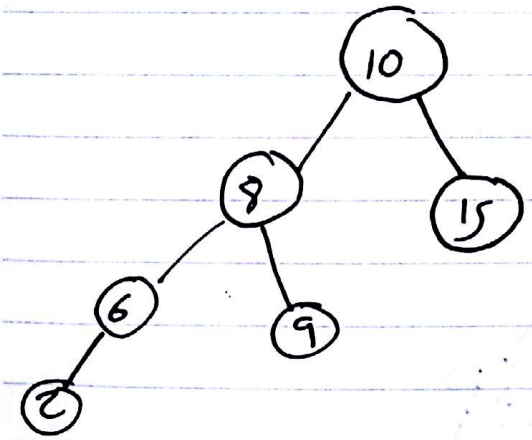
* AVL Tree :

BST

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

↳ longest path from (leaf to Node)

tree
↓
Binary
↓
BST
↓
AVL

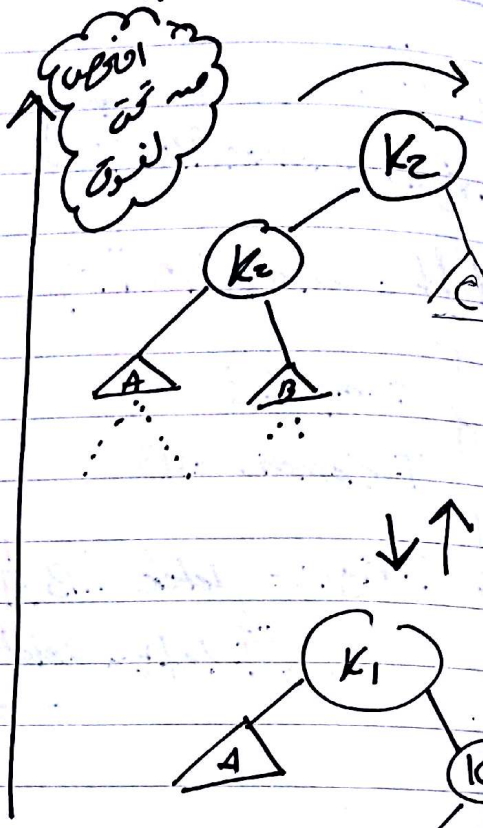


∈ Not AVL
→ Node 10 is a problem
→ check each node

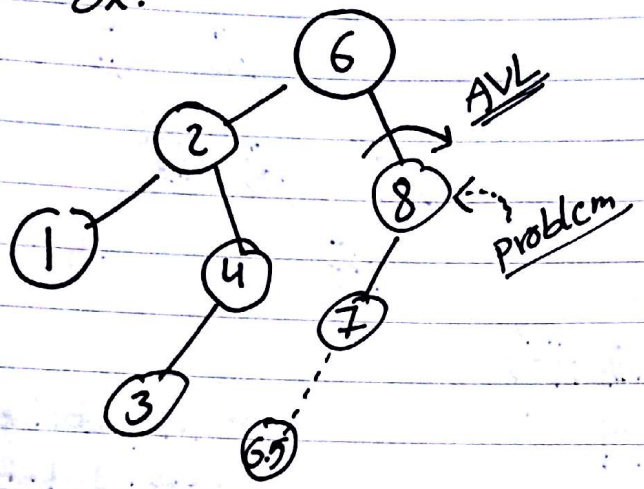
⇒ search, insert, Delete } O(log n)

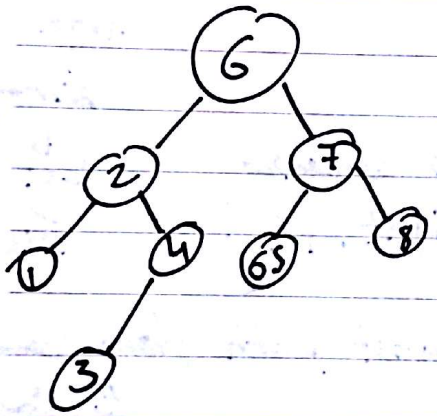
* How to make a tree balanced?

- Single rotate:-

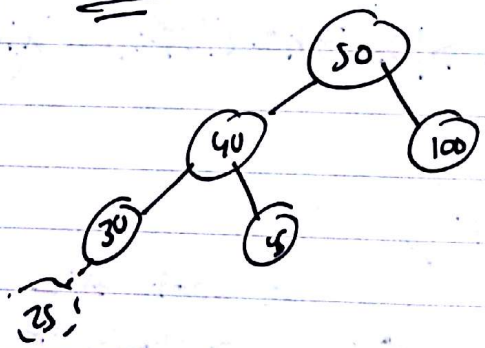


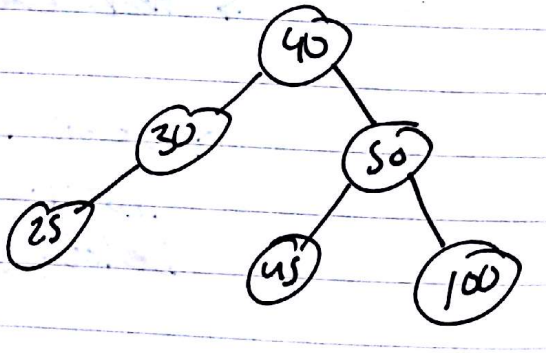
Ex:

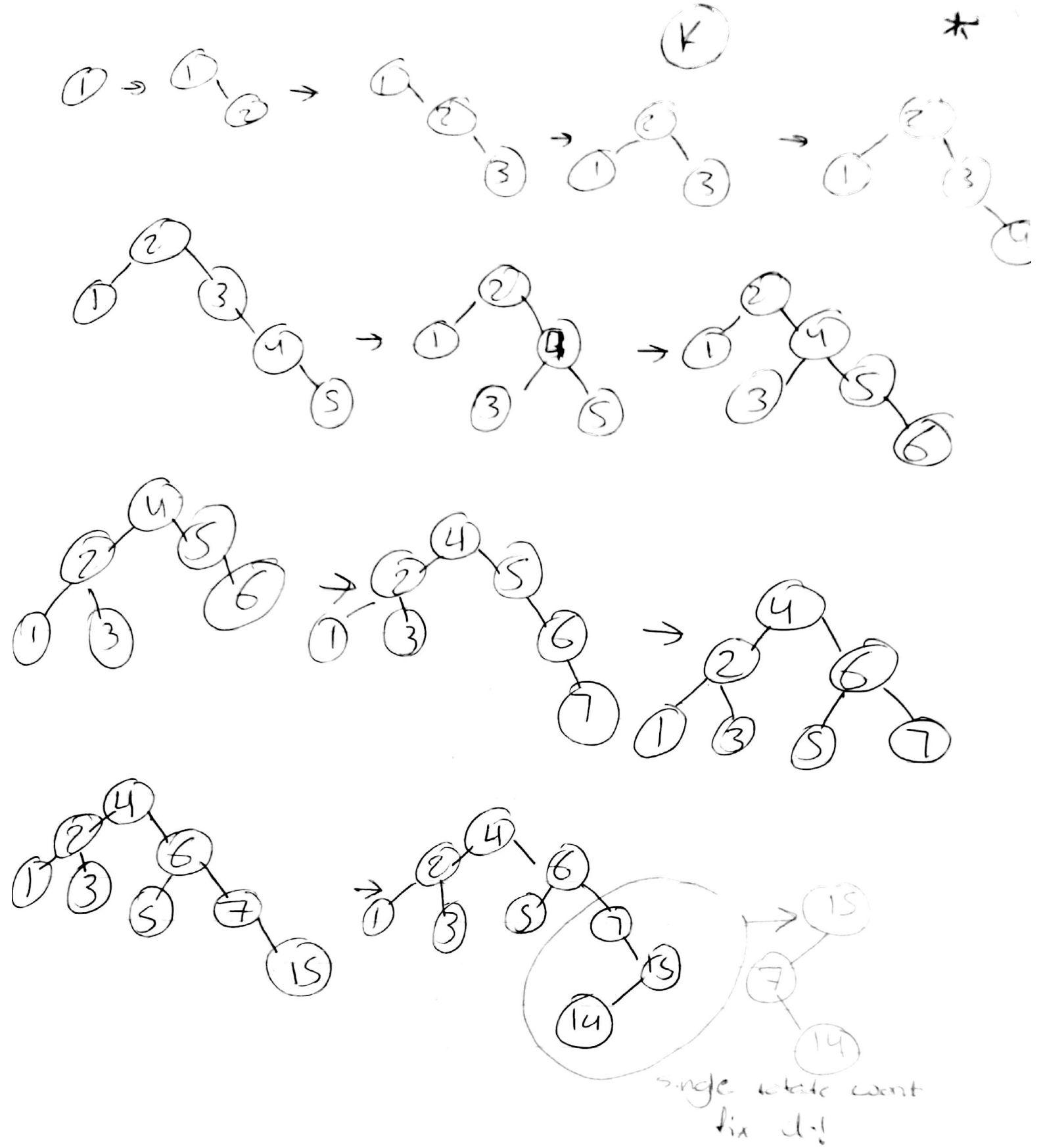




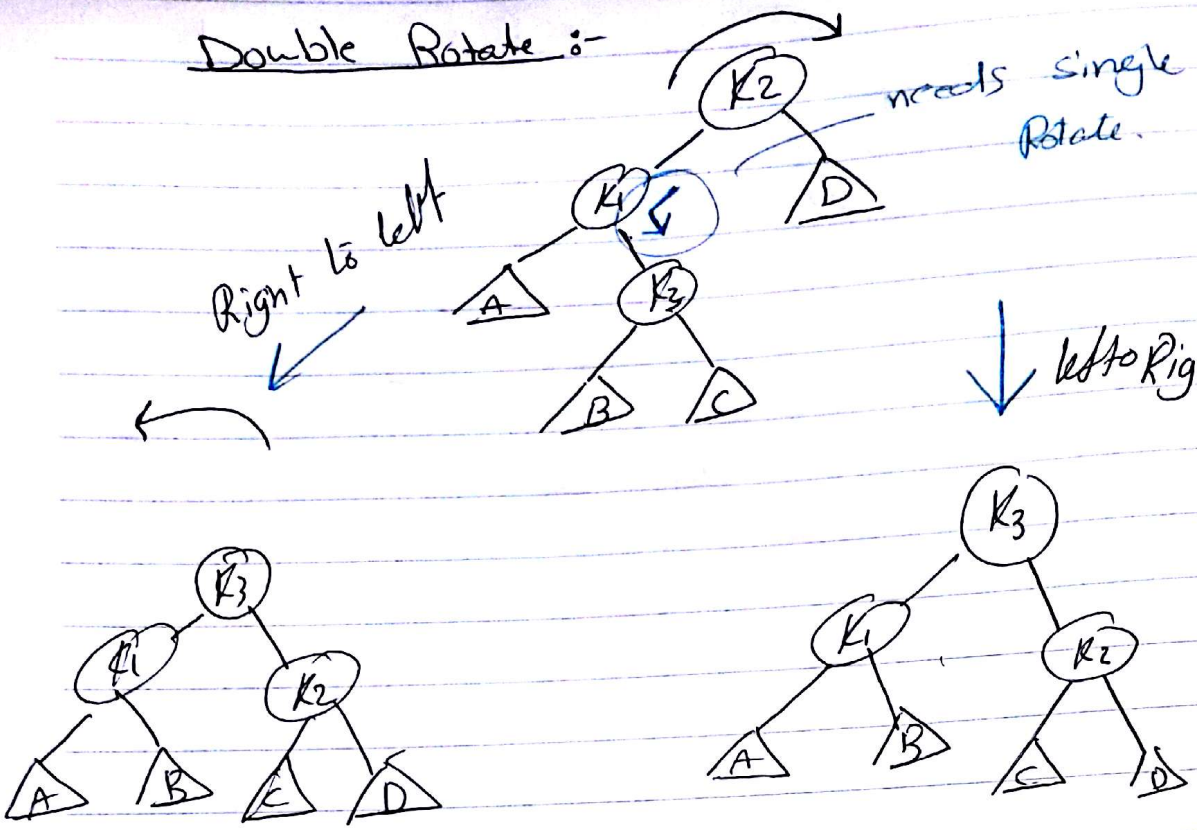
Ex



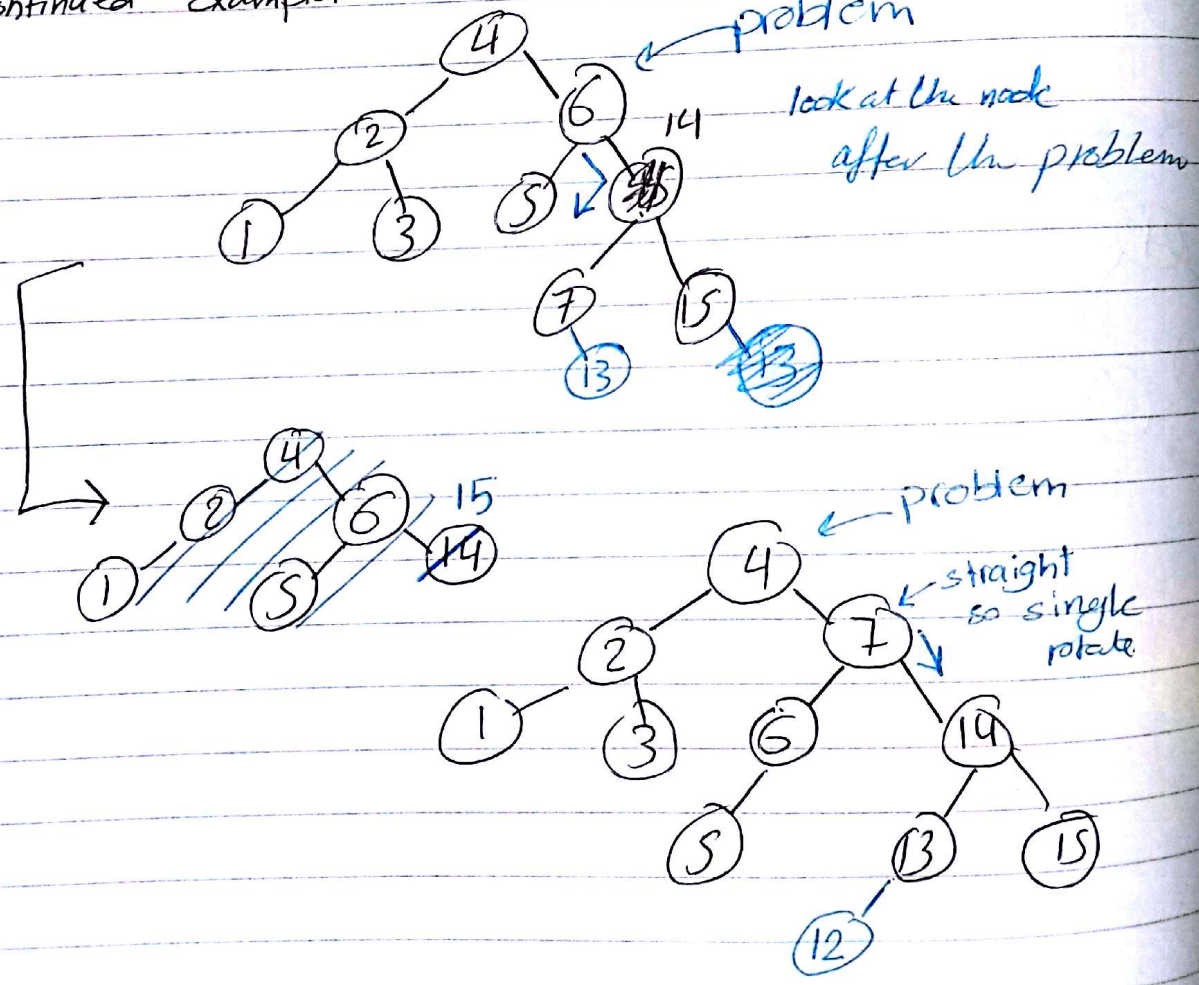


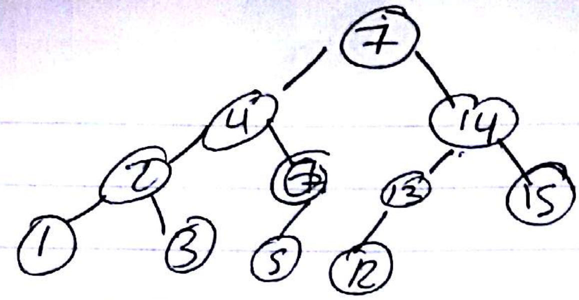


Double Rotate :-



Continued Example:



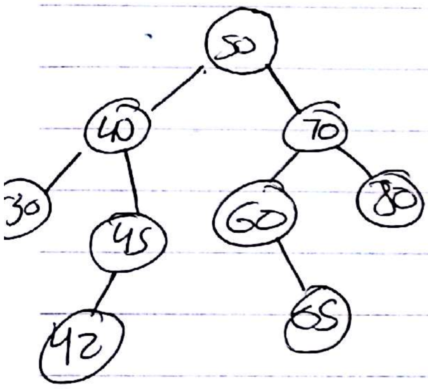


Best Case ⇒ Const
Worst Case O(log n)
Average Case O(log n)

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), CH(T))+1 [40 → 2, 70 → 3]
→ same thing we get 3

CH(T), CH(T) [30 → 0, 45 → 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;
```