Chapter 1 :-

- Data Structure :- an orgaization or structing for acollection of data items in the main memory (RAM).

- a solution is said to be efficient if it solves the problem with it's constraints (space and time).

- abstract Data Type (ADT) and Data Structures :-
  - * Type :- is a collection of values.
    1) Single types :- integer / boolean.
    2) Composite / aggregate :- circle object.

Ex :- boolean → true
                → false

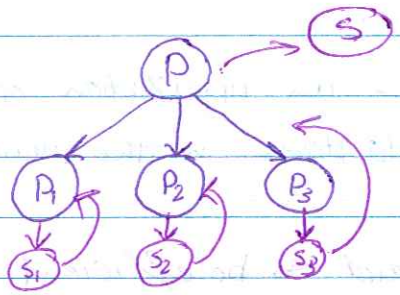
- byte → -128 : 127

- a data type :- is a type together with a collection of operations to maimpulate the type.
  (integer / Double)

- an abstract Data type (ADT) :- is the realization of a data type as a software component.

- A Data Structure :- is the implementation of the ADT.
  Ex :- ADT : list, stack / Tree, ...
  add(), remove(), clear(), ...

* Recursion :- method that directly or indirectly called itself



- Factorial problem :-

Find 5! ?

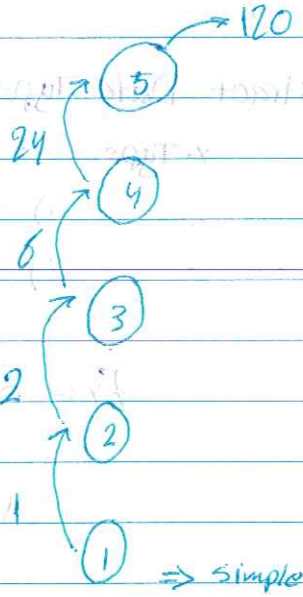
$P_1 \Rightarrow 5 * 4!$

$P_2 \Rightarrow 4 * 3!$

$P_3 \Rightarrow 3 * 2!$

$P_4 \Rightarrow 2 * 1!$

1



⇒ simple case / base case

```
int fact (int n){
  int val;
  If (n==1)
    val = 1;
  else
  val = n*fact(n-1);
  return val;
}
```

* Fibonacci number *

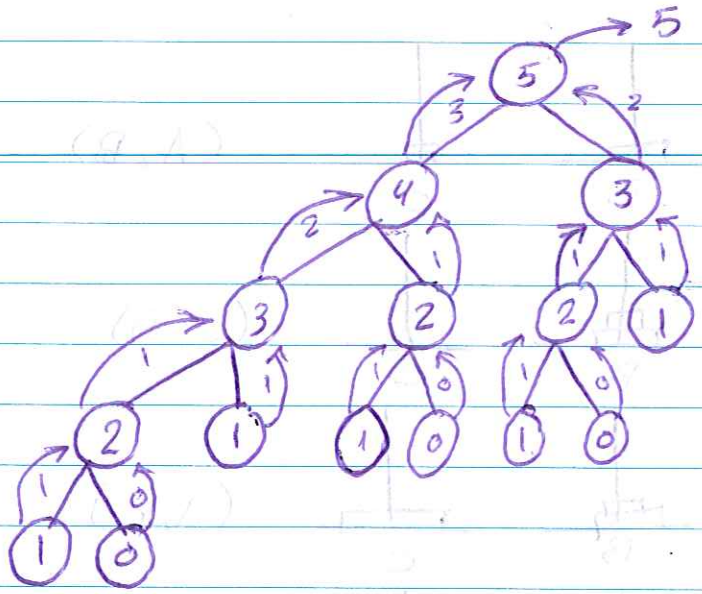
- Fibonacci Numbers :-

0, 1, 1, 2, 3, 5, 8, 13, 21, ...

Simple case :- $n == 0$, $Val = 0$

$n == 1$, $Val = 1$

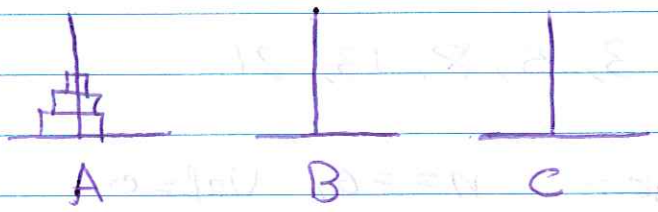
$\Rightarrow (n > 1)$ $val = fib(n-1) + fib(n-2)$ $\Rightarrow$ recursive case
inductive case

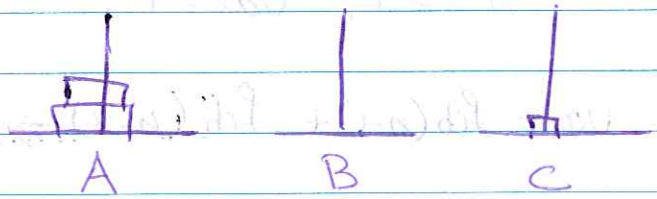


Fibonacci in Java programming :-

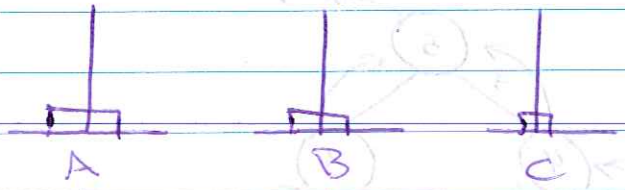
```
int fib (int n) {
If (n == 0 || n == 1)
    return n;
else
    return ( fib(n-1) + fib(n-2));
}
```

- Towers of Hanoi :-

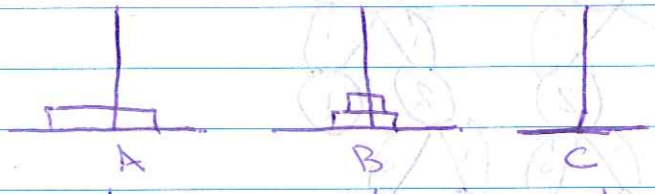




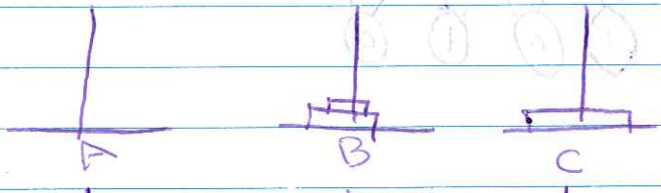
(A, C)



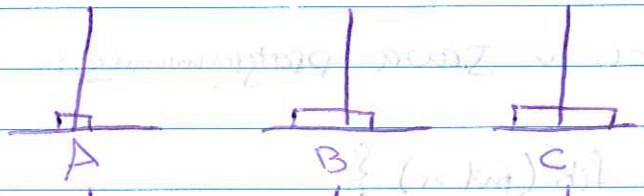
(A, B)



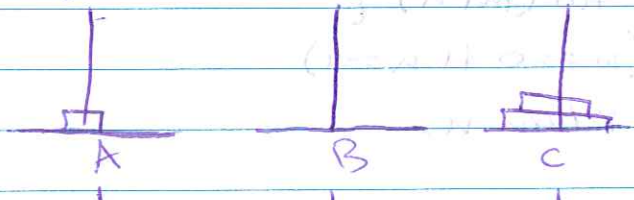
(C, B)



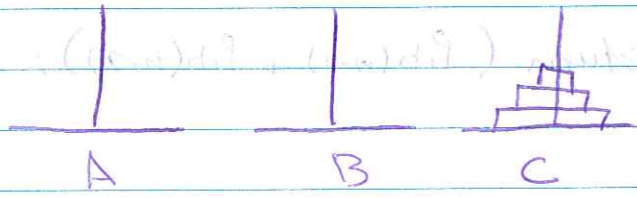
(A, C)



(B, A)



(B, C)

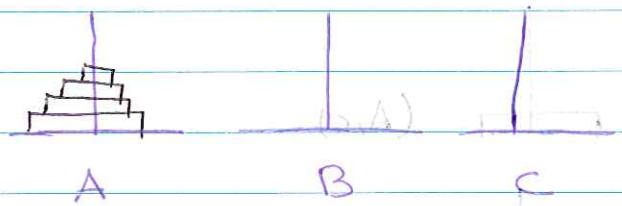


(A, C)

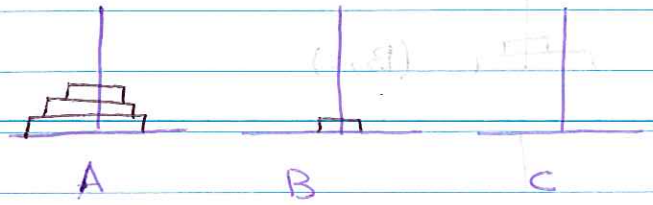
* # of moves = $2^n - 1$ "n = عدد الأقراص"

Ex :- $2^4 - 1 = 15$ move "4 أقراص"

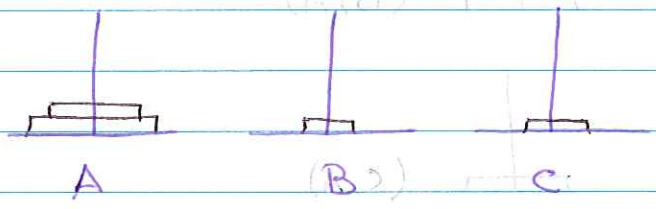
* problem :- moving 4 disks from A to C via B :-



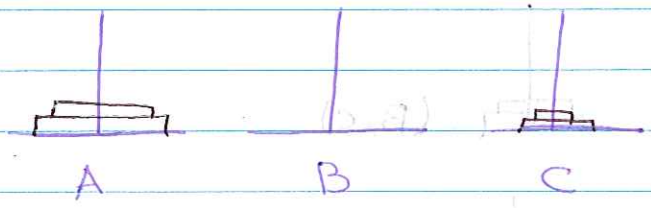
(problem)



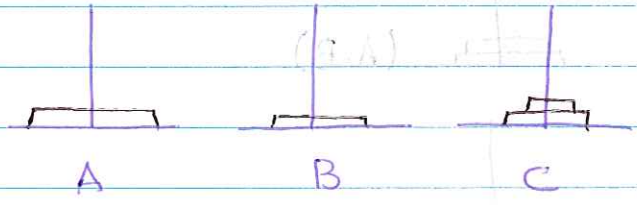
(A, B)



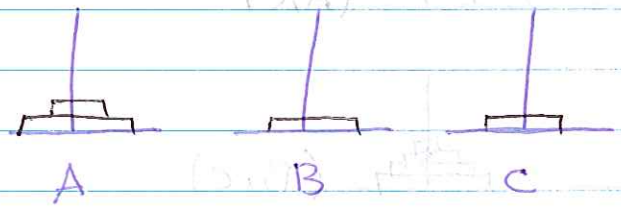
(A, C)



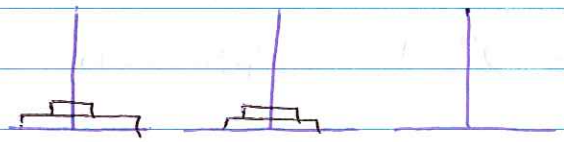
(B, C)



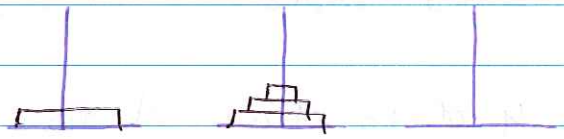
(A, B)



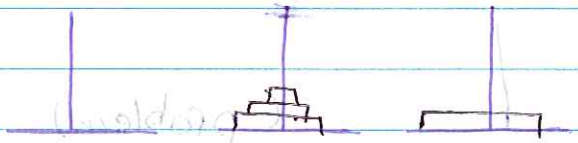
(C, A)



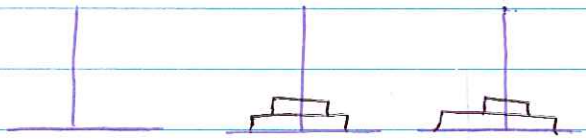
(C, B)



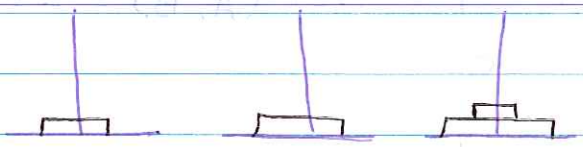
(A, B)



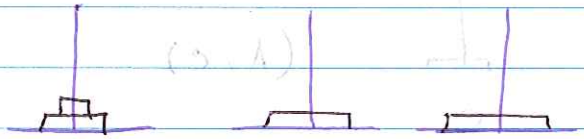
(A, C)



(B, C)

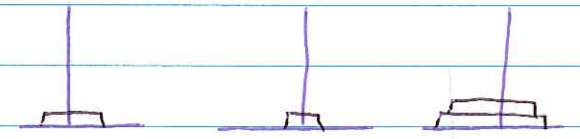


(B, A)

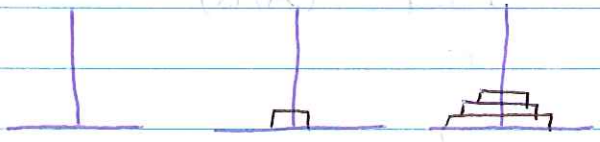


(C, A)

(B, C)



(A, B)



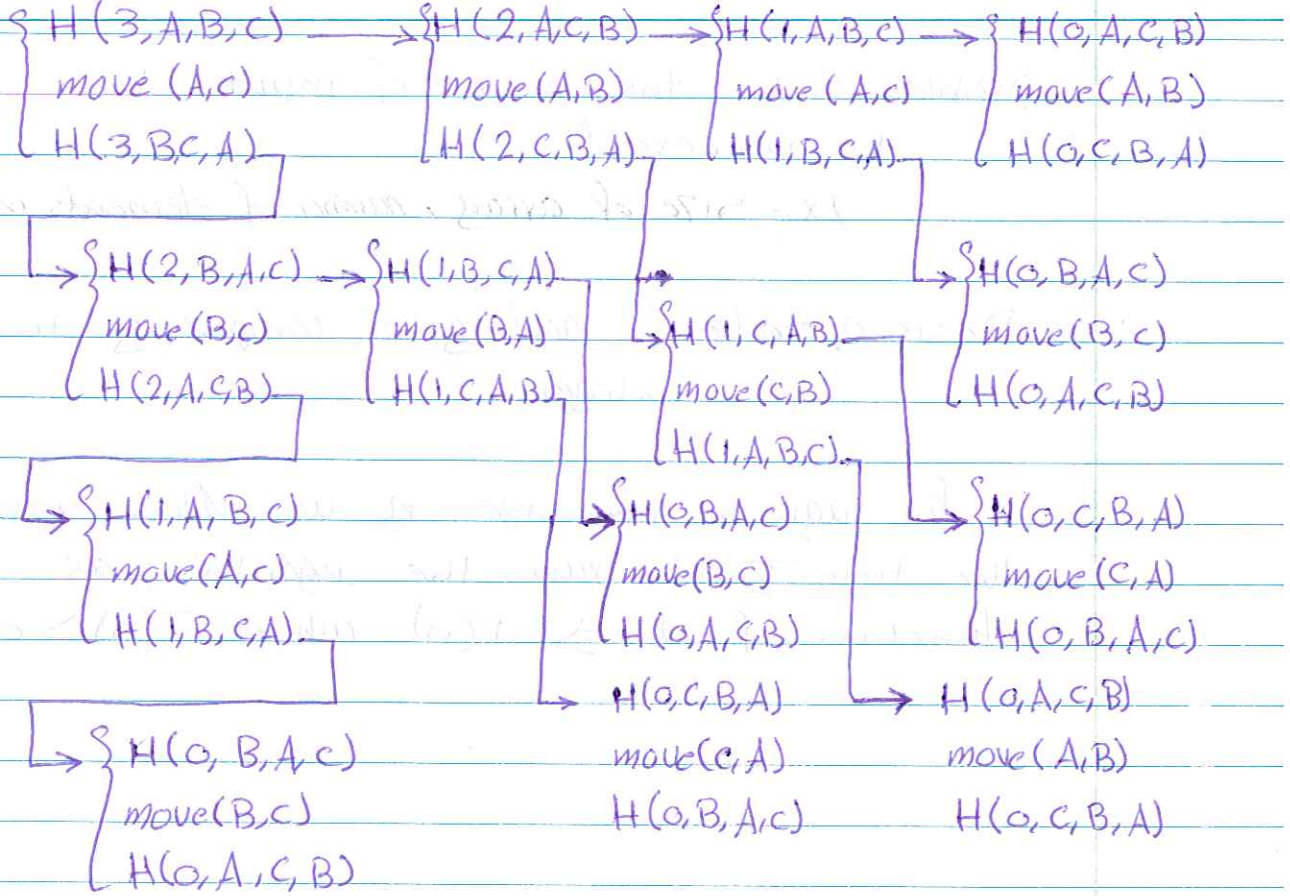
(A, C)

(B, C)

* Code in Java :-

```
Static Void H(int n, char start, char goal, char temp) {
    if (n == 0)
        return;
    else {
        H(n-1, start, temp, goal);
        move(start, goal);
        H(n-1, temp, goal, start); } }
```

* Example ∞ H(4, A, C, B)



* Analysis of algorithms

Critical resources → time (*)
                  → space

- The theoritical study of a computer program performance and resource usage.

- Asymptotic analysis mesures the efficiency of an algorithm, its OR its implementation as aprogram, as the input size become Large.

- problem size:- the number of inputs to be processeed
  EX:- size of array, number of elements in a sequenc

- Basic operation:- adding or comparing two integers.

* for agiven input size n we often express the time T to run the algorithm as a function of $n \Rightarrow T(n)$ where $T(n) > 0$.

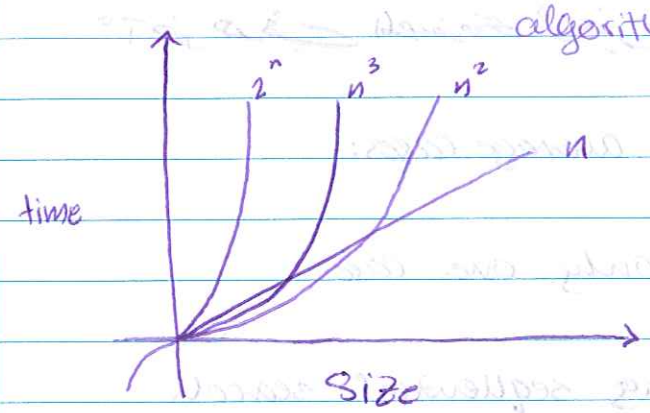
Example:-

```
for (int i=0; i<=n; i++)
    for (int j=0; j<=n; j++)
        sum ++;
```

$$T(n) = cn^2$$

Where c is the amount of time required to complete one basic operation.

~~X~~ Groth Rate:- ~~is the rate at which the cost of the~~ algorithm grows as the size of its inputs grows



$cn \Rightarrow$ linear groth rate
$n^2 \Rightarrow$ quadratic
$n^3 \Rightarrow$ cubic
$2^n \Rightarrow$ exponential groth rate
$n! \Rightarrow$ exponential groth rate

$$\log n \rightarrow n \rightarrow n\log n \rightarrow n^2 \rightarrow n^3 \rightarrow 2^n \rightarrow n!$$

* Sequential search:-

```
for( int i=0 ; i<n ; i++)
    if( key == A[i])
        return i;
```

7	8	15	9	2	0	9	3

- search for 7 ⇒ Best Case

- search for 9 ⇒ Average case ⇒ n/2

- search for 3 or 13 ⇒ Worst case ⇒ n

* Best, worst, and average cases:-

- factorial (n) ⇒ only one case

- Largest value using sequential search
  ⇒ one case

* upper Bound : O (big-oh)

* Asymptotic Analysis :-

it refers to the study of an algorithm as the input size gets big or reaches a limit

* Upper_Bound :- "Big_Oh" :-

$T(n) = c_1 n^2 + c_2 n$

$c_1 n^2 + c_2 n \leq c_1 n^2 + c_2 n^2$

$\leq (c_1 + c_2) n^2$

$c_1 n^2 + c_2 n \leq c_0 n^2$ where $c_0 = c_1 + c_2$

$\Rightarrow T(n)$ is in $O(n^2)$

"it indicates the upper or heighest groth rate that the algoritum can have."

* Lower Bound :- (Ω) "Omega" :-

$T(n) = c_1 n^2 + c_2 n$

$c_1 n^2 + c_2 n \geq c_1 n^2$

$\Rightarrow T(n)$ is in $\Omega(n^2)$

* Θ notation :-

When the upper-Bound and the lower-Bound are the same with a constant factor, we indicate this by Θ (Big-Theta) notation.

* an algorith is said to be $\Theta(h(n))$ if it is in $O(h(n))$ and is in $\Omega(h(n))$

* Simplifying Rules :-

1. if $f(n)$ is in $O(g(n))$ and $g(n)$ is in $O(h(n))$, then $f(n)$ is in $O(h(n))$.

2. if $f(n)$ is in $O(kg(n))$ for any constant $k > 0$, then $f(n)$ is in $O(g(n))$.
   'ignore constants'

3. if $f_1(n)$ is in $O(g_1(n))$ and $f_2(n)$ is in $O(g_2(n))$ then $f_1(n) + f_2(n)$ is in $O(\max(g_1(n), g_2(n)))$

4. if $f_1(n)$ is in $O(g_1(n))$ and f_2 is in $O(g_2(n))$, then $f_1(n) f_2(n)$ is in $O(g_1(n) * g_2(n))$

* Classifying Functions :-

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} \begin{cases} = \infty, & f(n) \text{ is in } \Omega(g(n)) \text{ // } f(n) \text{ faster} \\ = 0, & f(n) \text{ is in } O(g(n)) \text{ // } g(n) \text{ is faster} \\ \neq 0, & f(n) \text{ is } \Theta(g(n)) \text{ // Both grows at the same rate} \end{cases}$$

* Calculating the Running time for a program :-

The core of algorithm analysis is to find out the number of Basic operation ~~dep~~ depends the size of the input.

* Rules to count the operations :-

- Rule 1 :- for loop :-

```
for(i=0; i<n; i++)
    sum += i;
```

$i=0$	executed once	$O(1)$
$i<n$	$n+1$ times	$O(n)$
$i++$	n times	$O(n)$

heading + loop Body = $O(n)$ + $O(n)$ sum += i

$= O(n)$

for(0 → n) { n
  for(0 → 2*n) { 2n } } $O(2*n^2) \Rightarrow O(n^2)$

Rule 2 :- Nested Loops :-

1)
```
Sum = 0;
for(i=0; i<n; i++) { n
  for(j=0; j<n; j++) { n } } O(n*n) = O(n^2)
    Sum += i;
```

2)
```
Sum = 0;
for(i=0; i<n; i++)
  for(j=i; j<n; j++)
    Sum++;
```

i=0 the inner loop will execute	n times
i=1	n-1 times
i=2	n-2 times
i=n-1	once

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

3) sum = 0;
for(i=0; i<n; i++)
for(j=0; j<n; j++)
sum = sum + function(sum);

$(n^2 \log n)$

* function(sum) is in $O(\log n)$

(4) * Note: not all nested loops run in $O(n^2)$

Ex:- sum = 0;
for(k=1; k<=n; k*=2) { $\log n \rightarrow 1, 2, 4, = 2^k$
for(j=1; j<=n; j++) { n
sum++;

$\Rightarrow O(n \log_2 n)$

$n = 2^i \Rightarrow \log_2 n$

4) sum2 = 0;
for(k=1; k<=n; k*=2) $\Rightarrow \log n$
for(j=1; j<=k; j++)
sum2++;

$$\sum_{i=0}^{\log n} 2^i = 2^{\log n + 1} - 1 = 2n - 1 = O(n)$$

Rule 3 :- If statement

```
if (condition)
    S1;
else
    S2;
```

The Running time is the maximum of the running times of S1 and S2

Example :-

```
{int maxSum = 0;
  for(i=0; i<n; i++)
    for(j=i; j<n; j++)
      {  int sum=0;
         for(k=i; k<=j; k++)
             sum += a[k];
         if(sum> maxSum)
             maxSum = sum;
      }
    return maxSum;
    }
```

$$\sum_{k=i}^{j} j-i+1 \quad \rightarrow \quad \sum_{j=1}^{n-1} (j-i+1) = \frac{(n-i+1)(n-1)}{2}$$

$$\sum_{i=0}^{n-1} \frac{(n-i+1)(n-1)}{2} = \frac{n^3+3n^2+2n}{6} = O(n^3)$$

* Big-oh for Recursive function:-

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

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

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

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

$$T(n-2) = T(n-3) + 3$$

$$T(n-3) = T(n-4) + 4$$

$$T(n) = T(n-k) + k$$

$\Rightarrow$ let $k = n-1 = T(1) + n - 1$

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

Example :-

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

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

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

$$T(n/4) = 8T(n/8) + 3n$$

$$T(n/8) = 16T(n/16) + 4n$$

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

$T(1) = 1, \quad n/2^k = 1$ OR

$$n = 2^k \Rightarrow k = \log n$$

$$T(n) = 2^{\log n} T(1) + (\log n) n$$

$$= n + n \log n$$

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

$T(n) = T(n/2) + O(1)$ Binary search $O(\log n)$

$T(n) = T(n-1) + O(1)$ sequential sort $O(n)$

$T(n) = T(n-1) + O(n)$ selection Sort $O(n^2)$

$T(n) = 2T(n/2) + O(n)$ merge Sort $O(n \log n)$

$\frac{n(n+1)}{2}$

Example:-

$$T(n) = \begin{cases} 1 & , n = 1 \\ 2T(n/2) + 10 & , n > 1 \end{cases}$$

$= 2[2T(n/4) + 10] + 10$

$= 4T(n/4) + 20 + 10$

$= 4[2T(n/8) + 10] + 20 + 10$

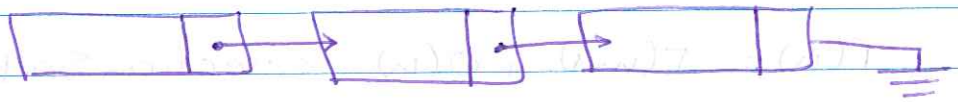
$= 8T(n/8) + 40 + 20 + 10$

$= 16T(n/16) + 80 + 40 + 20 + 10$

$= 2^4 T(n/2^4) + 2^{k-1} * 10 + 2^{k-2} * 10 + 2^{k-}$

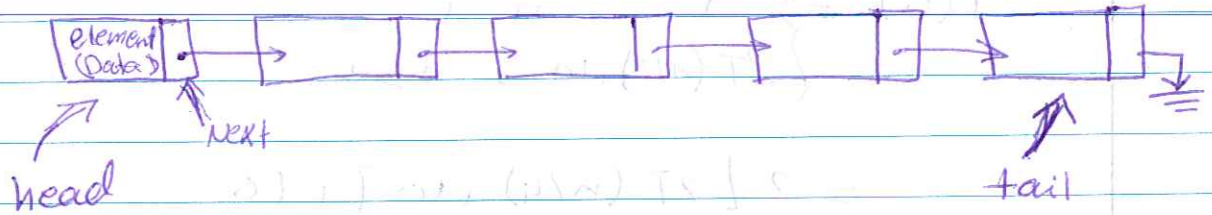
chapter 3 :- Lists , Stacks and queues :-

Linked list :-



* List operations :-

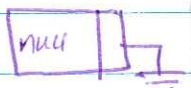
insert , remove, read , creat new list
clear , getNext



* List implementation (Arrays vs linked lists)

	operation	Array	Linked list
1)	Search "for an element"	O(n)	O(n)
2)	find kth element "index"	constant	O(n)
3)	print list	O(n)	O(n)
4)	~~Insert~~		
4)	insert the end	O(n)	constant
5)	"at the Beginning"	O(n)	constant
6)	insert to the end	constant	constant

```
public class Node {
    Node next;
    object data;
    public Node (object _data)
    {   next = null;
        data = _data;
    }

    public Node (object _data , Node _next)
    {
        next = _next;
        data = _data;
    }

    getData ()
    setData ()
    getNext ()
    setNext ()
}

public class LinkedList {
    private Node head;
    private int listCounter;
    public LinkedList ()
    {   head = new Node (null);
        listCounter = 0;
    }
```



```
// add to the end "tail"
public void add(object data) {

    Node temp = new Node(data);
    Node current = head;

    while(current.getNext() != null) {

        current = current.getNext();
    } // end loop
    current.setNext(temp);
    listCount ++;
}
```

"يمشي من head إلى أن يصل إلى null"
"إضافة Node جديدة إلى آخر الـ list"

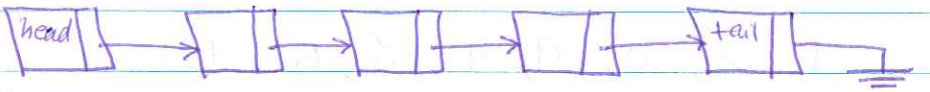
```
public void add (object data, int index) {
    Node temp = new Node (data);
    Node current = head;

    for(int i=1; i< index && current.getNext() != null; i++)
    { current = current.getNext(); }

    temp.setNext (current.getNext() );
    current.setNext(temp);
    listCount ++;
}
```

"يضيف Node في أي مكان داخل list"

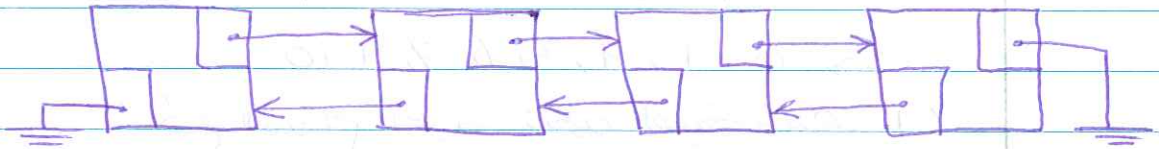
```
public object get(int index) {

  if (index = 0)
     return null;

  Node current = head.getNext();

  for(int i = 1; i< index; i++)
   {
      if(current.getNext() == null)
          return null;
      current = current.getNext();
   }
      return current.getData();
  }

  public boolean remove (int index)
  {
      if( index <1  ||  index > size() )
         return false;

     Node current = head;

  for(int i=1; i<index ; i++) {
    if (current.getNext() == null)
         return false;  // وصل آخر list ولم يجد العنصر
    current = current.getNext(); }
  current.setNext(current.getNext().getNext());
     listCounter --;
     return true; }
```

```
Public boolean remove(Object data){

    Node current= head;
    While(current.getNext() != null)
    {

    if((( current.getNext().getData()).equals(data))
    {
      current.setNext(Current.getNext().getNext());
       listCount --;
       return true;
    }
      Current = Current.getNext();
    }
  return false;
}

  public int size()
  {
    return listcount;
  }

  Public String toString() { ... }

 Public class Test{
  LinkedList List = new LinkedList();
     List.addel("comp232");
          :
          :

}
```

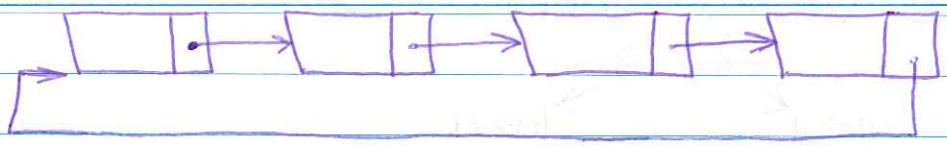
1) Single linked list :-



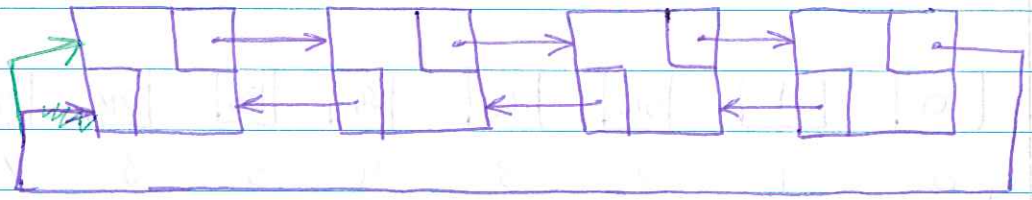
2) Doubly linked lists :-



3) circular linked lists :-

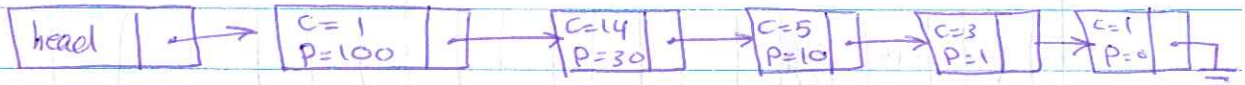


4) Doubly circular linked lists :-



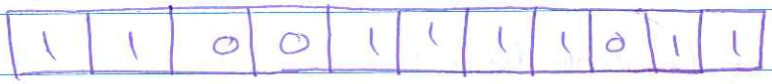
* application :- ⇒ (Polynomial)

$P(x) = x^{100} + 14x^{30} + 5x^{10} + 3x + 1x^0$



⇒ (Bucket Sort)

1, 7, 0, 9, 4, 6, 5, 10



↳ 0, 1, 4, 5, 6, 7, 9, 10

"يطبع الأرقام التي تكون القيمة عندها 1"

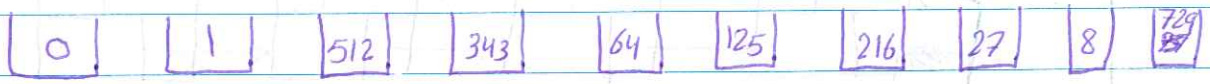
⇒ O(n)

⇒ (Radix Sort)

723

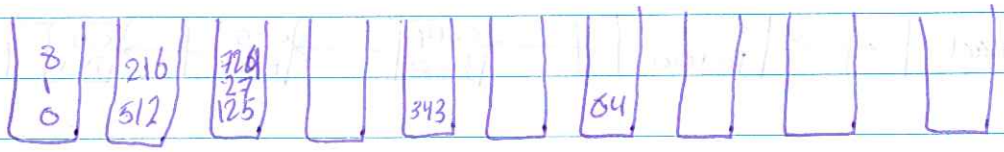
most ← → least

Example:- 64, 8, 216, 512, 27, 729, 0, 1, 343, 125



↳ 0, 1, 512, 343, 64, 125, 216, 27, 8, 729

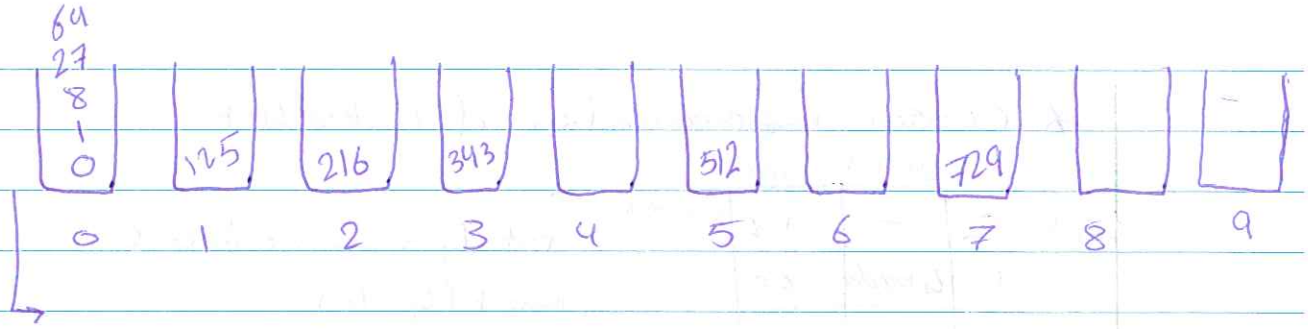
"ترتيبها تصاعدياً حسب الخانة الأولى"



↳ 0, 1, 8, 512, 216, 125, 729, 343, 64

27

"ترتيب تصاعدي حسب الخانة الثانية"



0, 1, 8, 27, 64, 125, 216, 343, 512, 729

"الترتيب حسب الطريقة الثانية"

$O(n \log n)$

~~Cursor implementation of Linked Lists:~~

```
j = 1;
for (int i=0; i<=9; i++)
{  current = A[i].next;
   while(A[i].next != Null)
   {
      A[i].next = current.next;
      digit = ((current.element / pow(10, j)) % 10;
      insert ( B[digit], current);
       current = A[i].next;
   }
   j++;
}
```

* Cursor implementation of Linkedlist :-

	element	next
header 0 →	—	1 2 3 4 5 6 7 8
1	L_1 header	2 0 2
2		3
3		4
4		5
⋮		
⋮		
n-1		0 → null

```
int L1 = Cursor Alloc ( );
insert (L1 , 10);
int L2 = cursorAlloc();
insert (L1 , 7);
insert (L2 , 15);
insert (L2 , 17);
insert(L1 , 20);
```

لما يوجد pointer فيكون
الـ pointer هو الـ index الموجود
في next

free list

	element	next
0	—	1 2 3 4 2
1	L_1 head	2 0 2
2	10	3 0 4
3	L_2 head	4 0
4		5
5		6
⋮		
n-1		0

cursorAlloc ⇒ free space يتم حجز أول

delete(L_1, 10)

L_1 ⇒ linkedlist

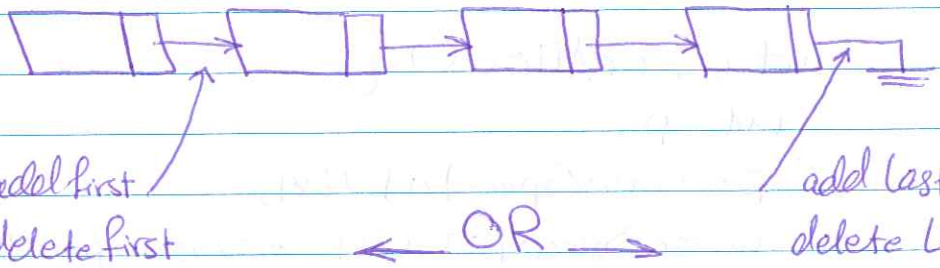
L_2 ⇒ linkedlist

عند الإضافة (insert) نكتب
تحديد في أي linkedlist

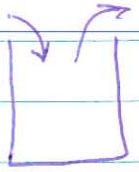
JavaCode for cursor Allocation :-

```
int cursorAlloc() {
  int p;
  p = CursorSpace[0].Next;
  CursorSpace[0].Next = CursorSpace[p].Next;
  return p;
  }

Void CursorFree (int p) {
  CursorSpace[p].Next = CursorSpace[0].Next;
  CursorSpace[0].Next = p;
  }

Boolean isEmpty (List L) {
  return (cursorSpace[L].Next == null);
  }

                                  → head
boolean isLast (int p, List L) {
  return (CursorSpace[p].Next == null);
  }

                                  → head
int find (object x, List L) {
  int p;
                    → head
  p = CursorSpace[L].Next;
  while ((p != null) && CursorSpace[p].element != x)
  { p = CursorSpace[p].Next; }
  return p;
  }
```

Stacks:-



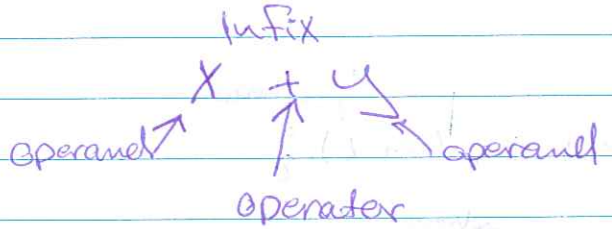
push : add
pop : delete
top : point to the first freespace



⇒ First in Last out

Stack
- array-Based
  - last → O(1)
  - first → O(n)
- Linked list Based

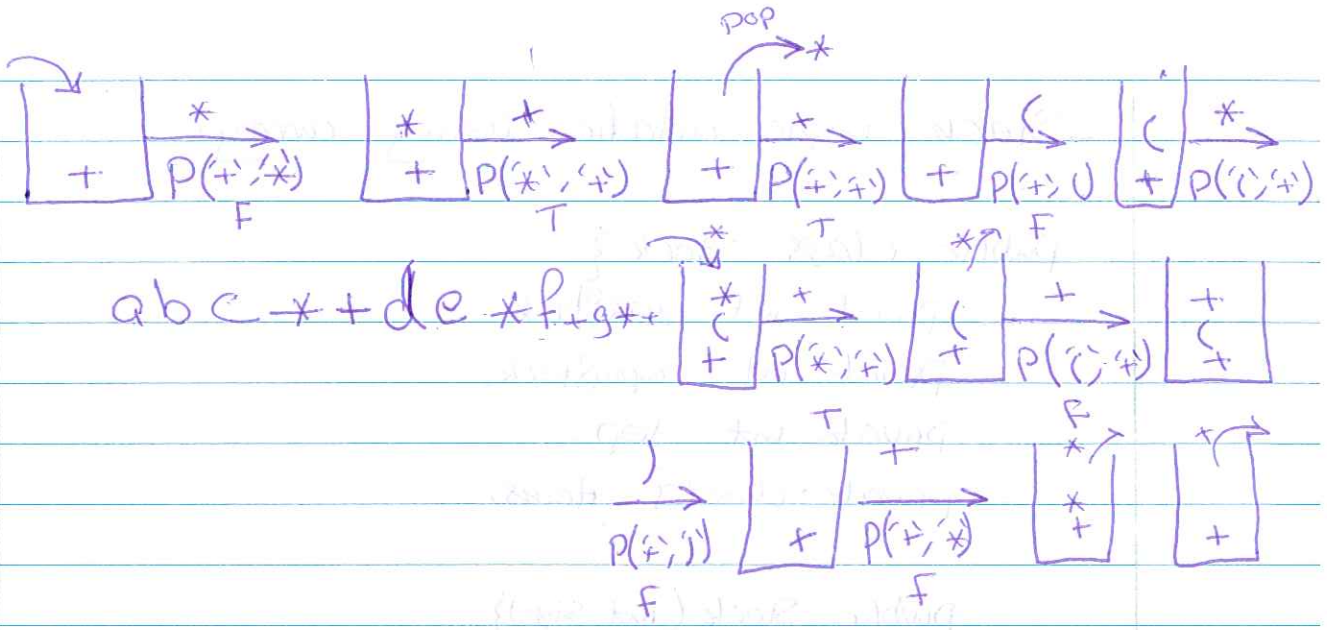
Application of stacks:-



In-fix to post-fix conversion:-

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

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



Rules:-

P('*','+') → T
P('+','+') → T
P('+',')') → f
P('+','(') → T
P('(',op) → F
P(op,'(') → F
P(op,')') → T
P(')','(') → Error
Ex (a+b-c)(a+b)

- Stack implementation using array:-

```
public class Stack {
    private int maxStack;
    private int emptyStack;
    private int top;
    private char[] items;

    public Stack (int size){
      maxStack = size;
      emptyStack = -1;
      top = emptyStack;
      items = new char[maxStack]; }

  public void push (char c){
      if(!full())
       items [++top] = c;
       else

    }

  public char pop (){
    if(! isEmpty)
      return items[top--];
      }
```

```
public boolean full () {
    return top+1 == MAXStack; }

public boolean Empty () {
    return top == emptyStack; }
```

- Stack LinkedList implementation ∞

1- at the beginning :-

```
public void push (object newItem) {
  list. add (1, newItem); }

public object pop () {
  object temp = list. get(1);
  list .remove(1);
   return temp;
  }
```

- adding and deleting from the end :-

```
public void push(Object newItem)
  {
    list.add(list.size() + 1, newItem);
  }

public object pop()
  {
    object tmp = list.get(list.size());
    list.remove(list.size());
    ~~temp = tmp~~
    return tmp;
  }
```

Array Based Queue:-

```
class queue {

    int [] items;
    int front, rear;
    }

boolean isEmpty (queue q) {

    return q.front == q.rear;
    }

int remove (queue q) {

  if (!isEmpty (q)) {
    int tmp = front;
    q.front = (q.front+1) % size;
     return q.items [tmp];
    }
    else
       empty list
    }
```

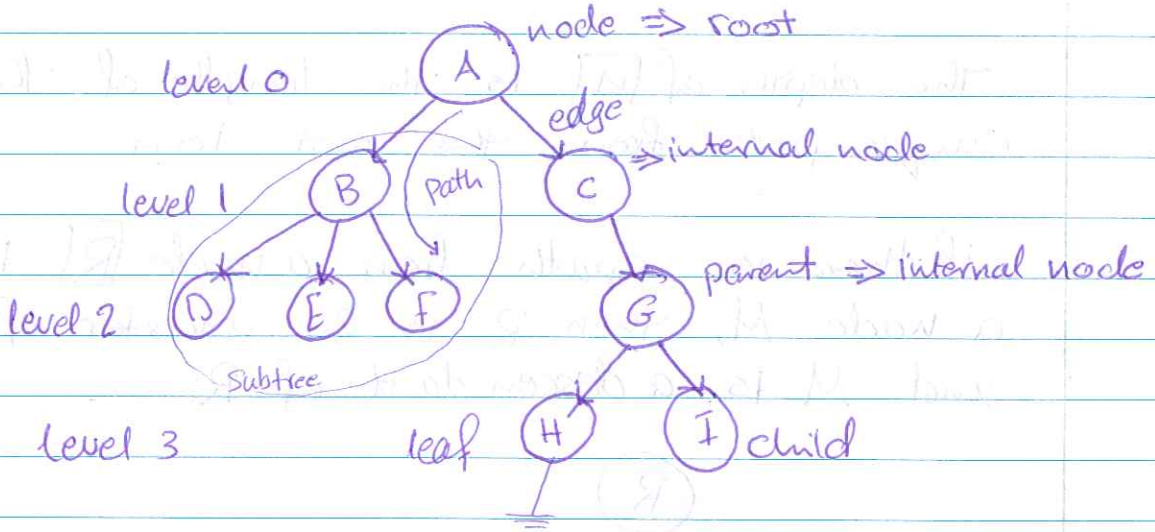
```
void insert (queue q , int x) {

    int temp ;
    temp= (q.rear+1) % size;

    if (temp == q.front)

            full
    else
    {
        q.rear =temp;
        q.items[q.rear] = x;
    }
}
```

Trees ∞

1. Tree
2. Binary Tree
3. Binary search tree (BST)
4. AVL Tree (BST) With Balance
5. splay Tree (BST without Balance)

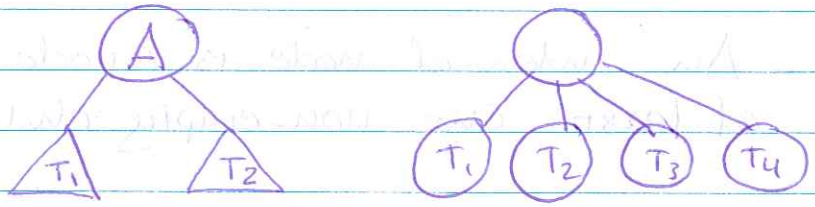
delete, find, insert

Trees :-



1) A Tree is made up of a finteset of elements called [Nodes]

2) The Tree is either empty or consists of a node called [root] together with zero or more [Subtrees] $T_1, T_2, \ldots, T_n$ each of whose roots are connected by a direct edge to the root.

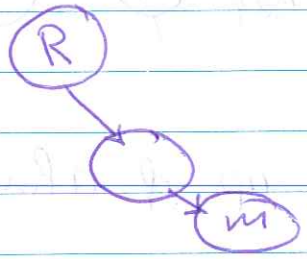


3) There is an [edge] between each [node] and its children, and a node is said to be a [parent] of its children

- depth :- is number of edges

- The depth of [n] is the length of the unique path from the root to n.

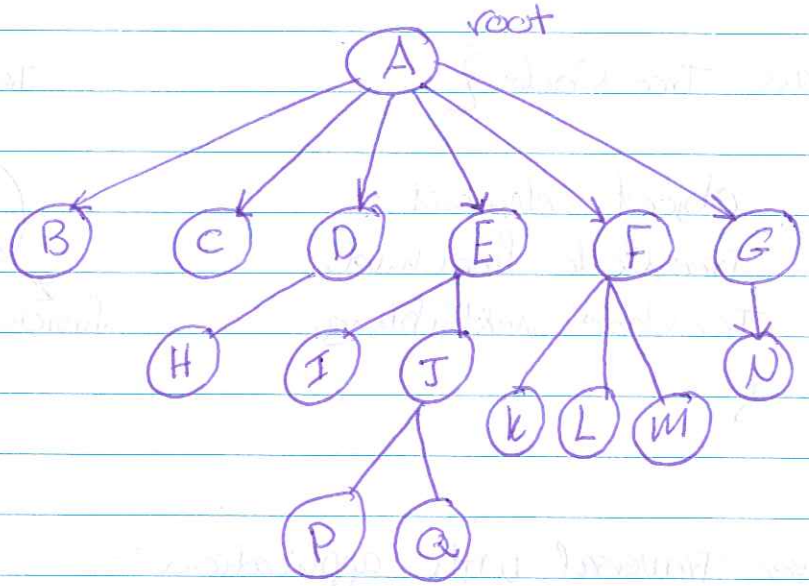
- if there is apath from a node [R] to a node M, then R is an ancestor of M, and M is a descendant of R.



- The height of [n] is the longest path from [n] to the [leaf]

- A [leaf] node is any node that has empty ~~chi~~ children

- An internal node is a node that has at least one non-empty children



depth (J) = 2 height (A) = 3 path from A to Q

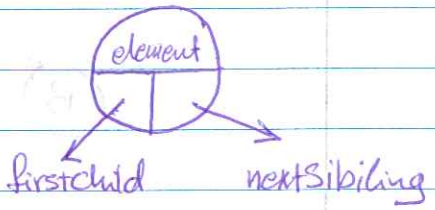
depth (A) = 0 height (E) = 2 A → E → J → Q

depth (C) = 1 height (F) = 1 "unique path"

class TreeNode {

object element;
TreeNode firstChild;
TreeNode nextSibling;
}

TreeNode

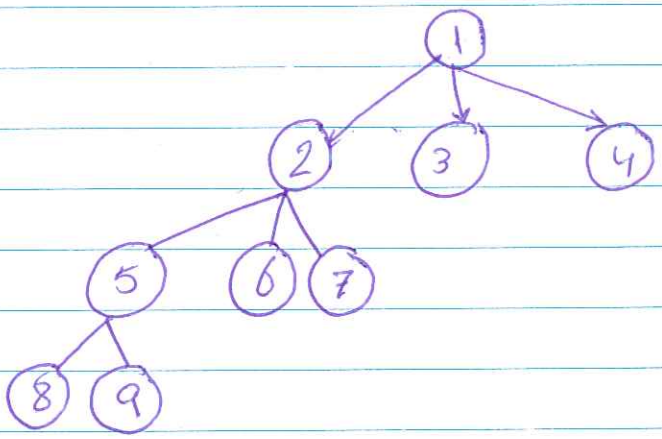


* Tree Traversal with application :-

any process of visiting all nodes in a tree in some order is called tree traversal

1. pre-order :-
Visit any given node befor visiting its children

((root → left → right))



pre-order :-

1, 2, 5, 8, 9, 6, 7, 3, 4

root ↗ ↑ right most

2 post order:-

Visit each node only after visiting each children (including its subtrees)

((left → right → root))

8, 9, 5, 6, 7, 2, 3, 4, 1

3 Inorder :-

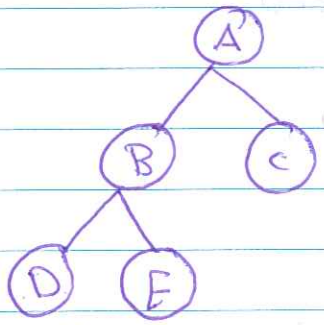
first visit the left child (including all its subtrees) then visit the node, and finally visit the right child (including its subtrees)

((left → root → right))

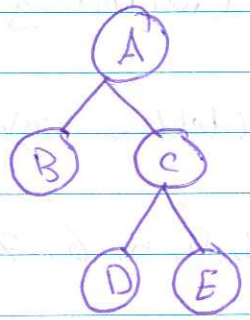
8, 5, 9, 2, 6, 7, 1, 3, 4

* Binary tree :-

1) Full Binary tree :-

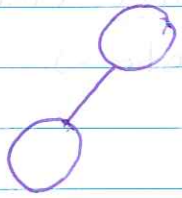


full & complete

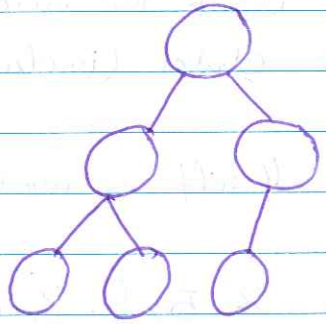


Full & not complete

2) Complete Binary tree



Complete & not full

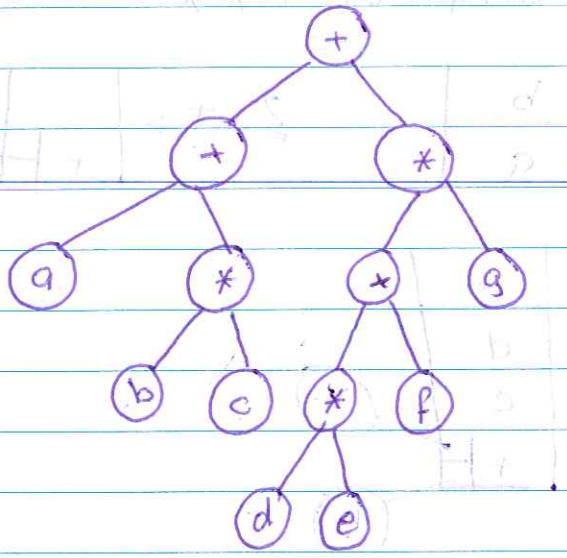


not full & ~~not~~ Complete

* Binary tree implementation:-

```
public class BSTNode{
    Comparable data;
    BSTNode left;
    BSTNode right;
    public BSTNode(Comparable d){
    data = d; left = right = null; }
```

* Expression Tree:-



1 inorder:- "left → root → right"

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

2 postorder:- "left → right → root"

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

3 preorder:- "root → left → right"

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

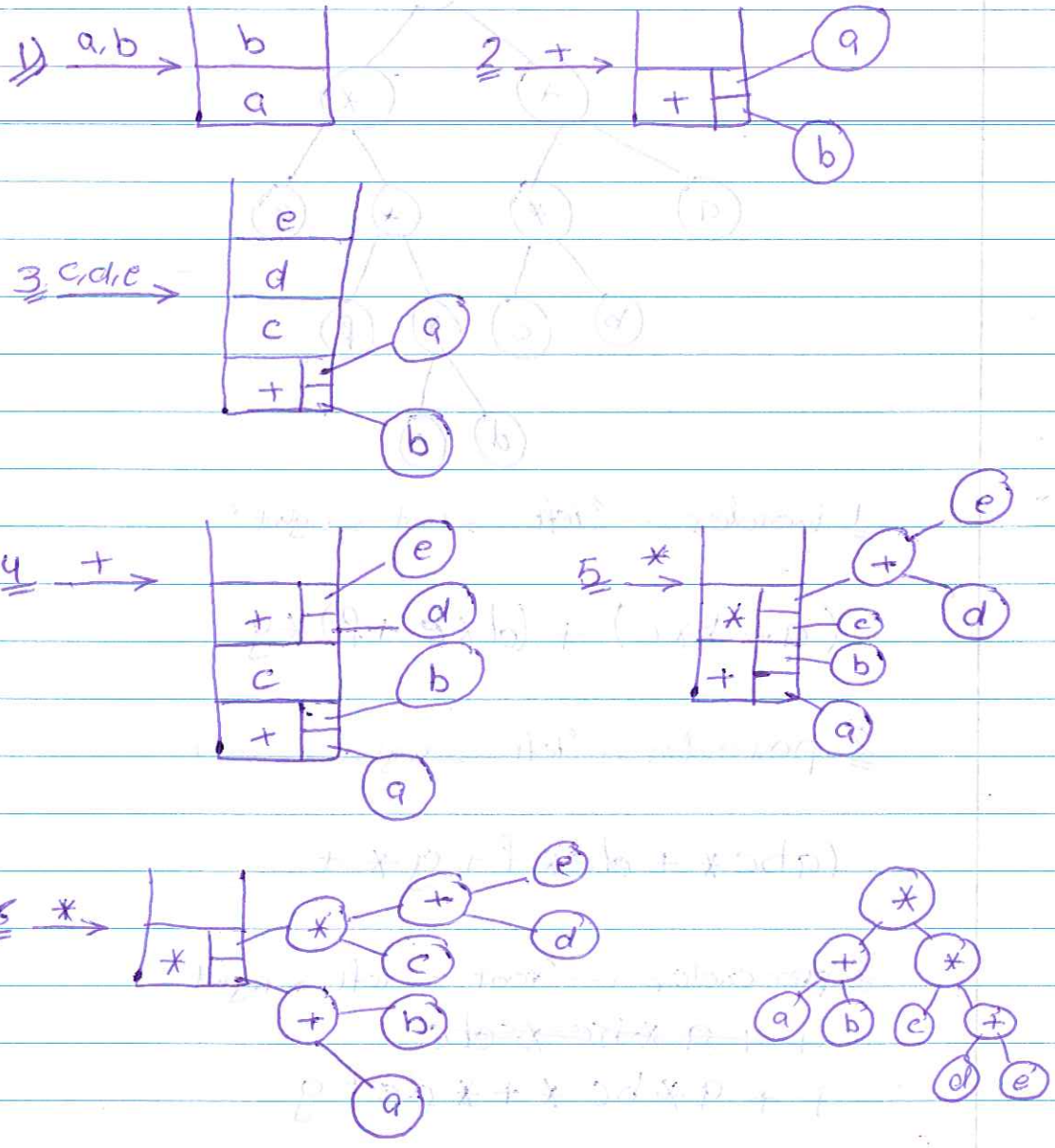
+ + a * b c * + * d e f g

* Constructing an expression tree:-

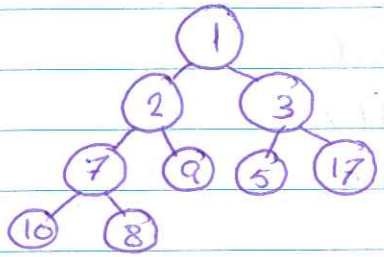
1. Convert the expression from infix to postfix

2. Scan the expression when encounter an operand push it on the stack, when you encounter an operation pop last to suptree.

Example:- ab + cde + * *



* Binary search tree :-

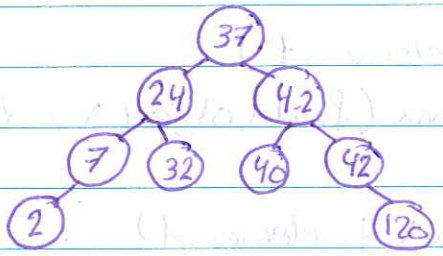


O(n)

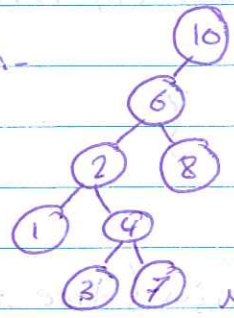
n: number of nodes

in BST, every node X in the tree, the value of all keys in the left subtree are smaller than the key value of X, and the value of all the keys in the right subtrees are greater than or equal to the key value in X $\Rightarrow$ left < root <= right

Example :- 37, 24, 42, 7, 2, 40, 42, 32, 120



Example :-



Not BST

Example :- 120, 42, 42, 7, 2, 32, 37, 24, 40

```
class tree_node {

    object element;
    tree_node left;
    tree_node right;
    }
```

* find operation in BST :-

// to find any node in BST :-

```
tree_node find (object x, tree_node t) {

    if (t == null)    // no nodes / no data
        return null;

    if(x < t.element)
        return (find (x, t.left));
    else
        if(x > t.element)
            return (find (x, t.right));
        else
            return t; }
```

* find min :-

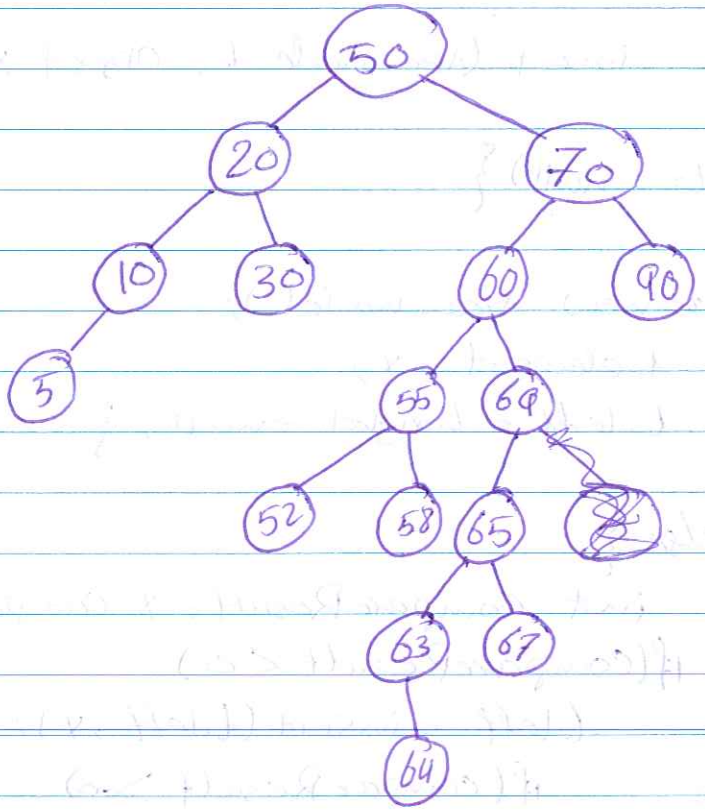
```
tree_node find_min (tree_node t) {
    if (t == null)
        return null;
    else if(t.left == null)
        return t;
    else
        return find_min (t.left); }
```

(44)

```
tree_node  insert (tree_node t, Object x){

    if( t== null){

        t=new tree_node();
         t.element =x;
         t.left = t.right =null;}

      else{
         int CompareResult = x.compareTO(t.element);
       if(CompareResult <0)
            t.left =insert(t.left,x);
  else      if(compareResult >=0)
              t.right = insert(t.right, x);

            else
            return t; // do nothing
          }
```

* find max:-

```
 tree_node find_max (tree_node t){
     if( t==null)
      return null;
    else
       if (t.right == null)
         return t;
       else
        return find_max(t.right);
    }
```




```
tree_node delete (tree_node t, object X) {
    if (t == null)
        error; // No nodes in the BST

    int compRes = X.CompareTo (t.element);
    if (compRes < 0)
        t.left = delete (t.left, X);
    else if (compareRes > 0)
        t.right = delete (t.right, X);
    else if (t.left != null & t.right != null) {
        t.element = find_min (t.right).element;
        t.right = delete (t.element, t.right); }
    else
        t = (t.left != null) ? t.left : t.right;
    return t;
}
```

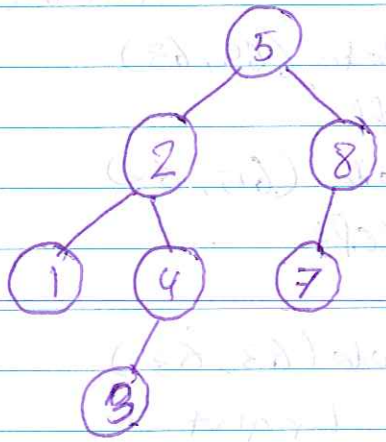
Example :- delete(T, 60)

delete (70, 60)
↳ T.left = delete(t.left, 60)
⇒ delete(60, 60)
↳ temp = find_min (60, right)
find_min(69) ⇒ 63
69
⇒ delete (69, 63)
left
⇒ delete (65, 63)
left

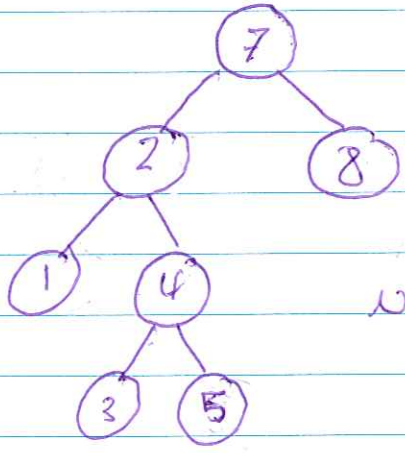
⇒ delete (63, 63)
t = t.right

AVL Tree :-

an AVL tree is identical to a BST, except that for every node in the tree, the height of the left and right subtrees can differ by at most one.



⇒ AVL tree



right = 2
left = 4
4 - 2 = 2 ⇒ Not Balan

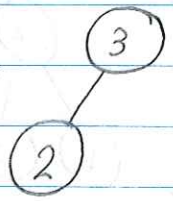
Not AVL Tree

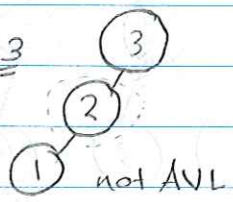
When inserting a new node, we have four options :-

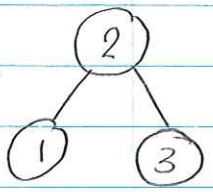
1. An insertion into left subtree of the left child
   1 + 4 ⇒ outside ⇒ single rotation
2. An insertion into right subtree of the left child
   3 + 2 ⇒ inside ⇒ double rotation
3. an insertion into left subtree of the right child
4. an insertion into right subtree of the right child

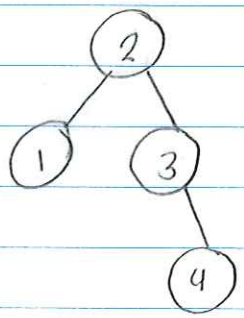
Example:- Start with an empty AVL tree, and insert the following items in a sequential order 3, 2, 1, 4, 5, 6, 7

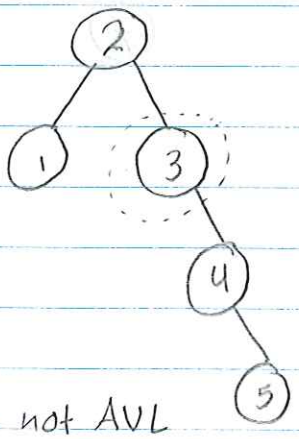
1. 

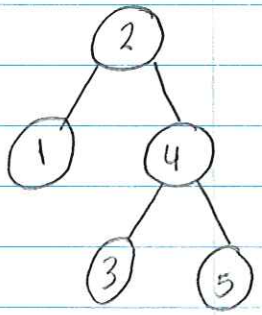
2. 

3. 

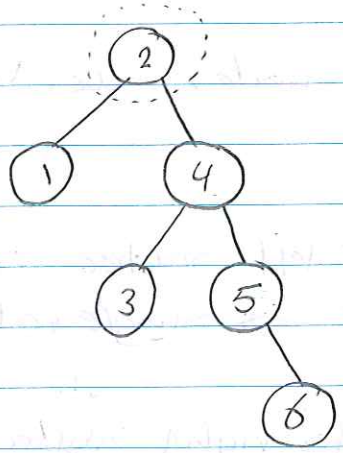
4. 

5. 

6. 

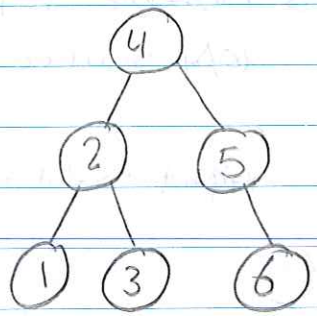
7. 

8=

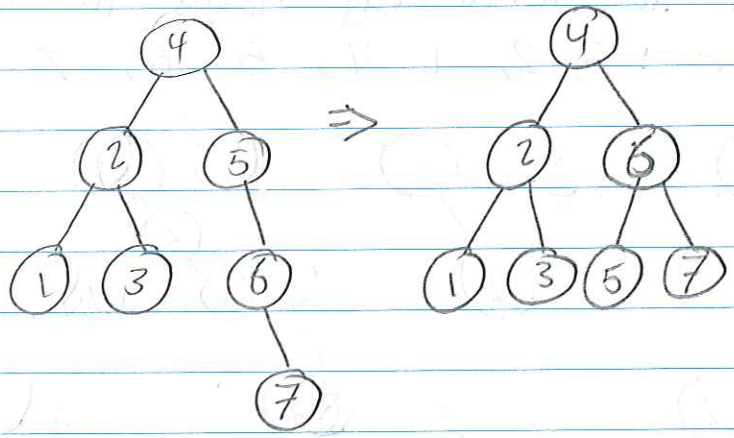


not AVL

9=



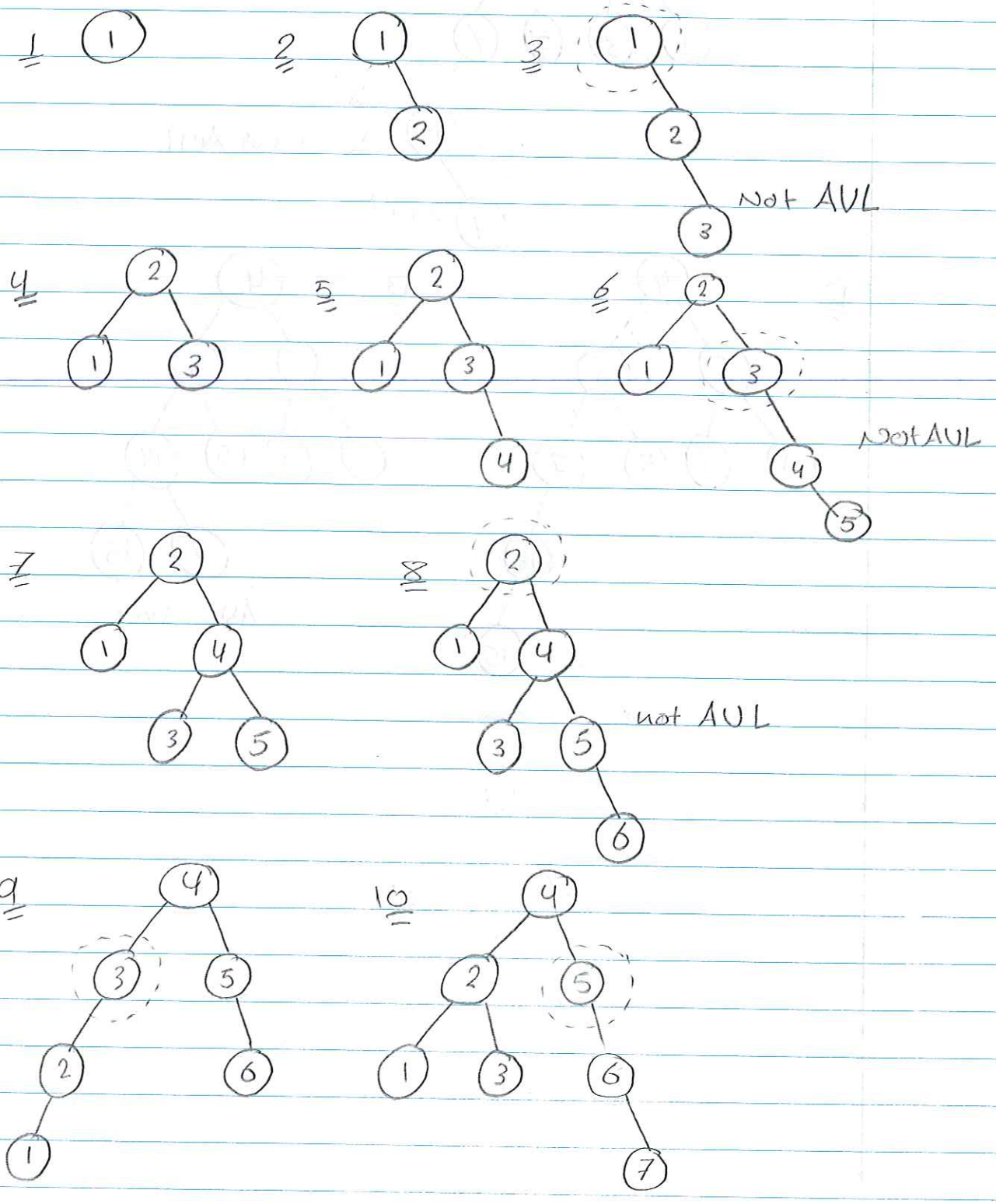
10=



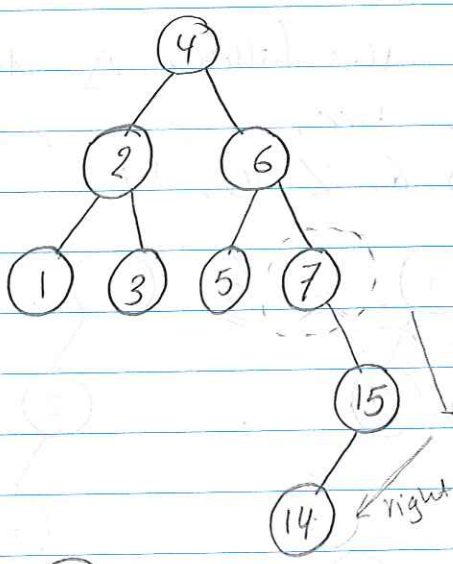
(50)

Example: insert the following items into an empty AVL tree.

1, 2, 3, 4, 5, 6, 7, 15, 14.

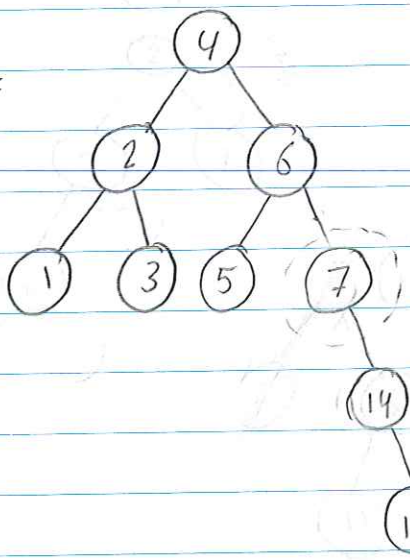


11

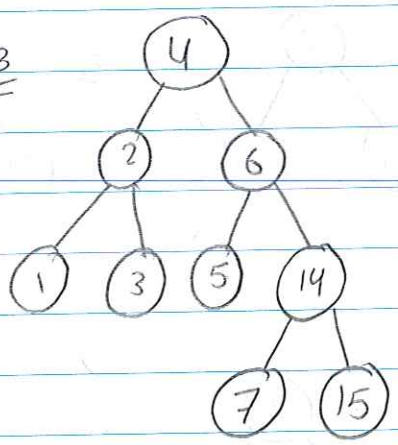


not AVL

12



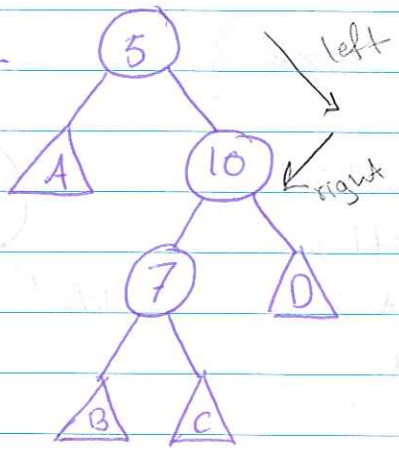
13



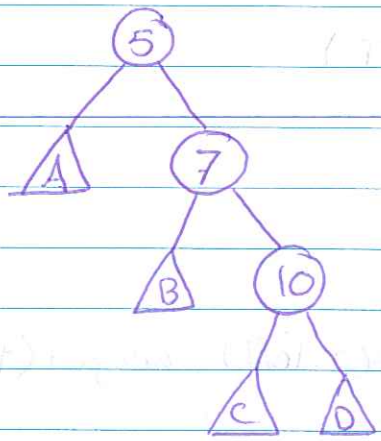
AVL tree

* Double rotation :- "MIT OCW"

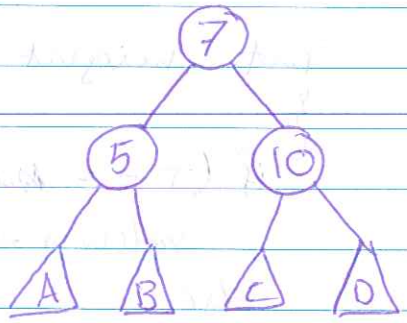
Example 1 :-



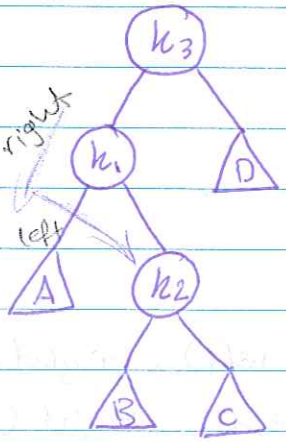
1 ≡



2 ≡

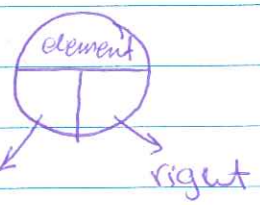


Example 2 :-



* AVL Tree Implementation :-

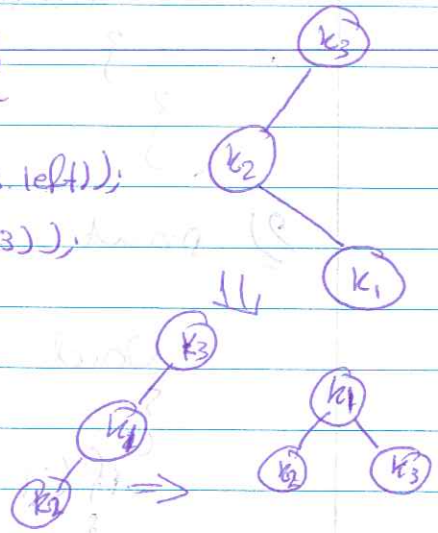
```
public class AVL ~~node~~ {

    object element;
    AVL ~~node~~ left;
    AVL    right;
     int   height;
    }
```



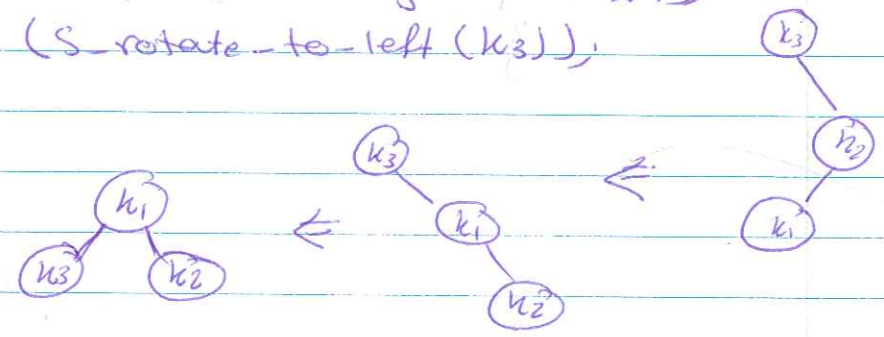

```
int height (AVL T)
{
  if (T == Null)
      return -1;
  else
   return (max (height(T.left), height(T.right))+1);
    }

AVL S_rotate_to_right (AVL k2) {

   AVL k1;
   k1 = k2.left;
   k2.left = k1.right;
   k1.right = k2;
   // update height
   k2.height = max(height(k2.left), height(k2.right))+1);
   k1.height = max(height(k1.left), height(k1.right))+1);
   return k1;
}
```

```
AVL S_rotate_to_left(AVL k2){
    AVL k1;
    k1 = k2.right;
    k2.right = k1.left;
    k1.left = k2;
    k1.height =
    k2.height =
    return k2;
}
```

```
AVL D_rotate_R(AVL k3){
    k3.left = S_rotate_to_left(k3.left);
    return (S_rotate_to_right(k3));
}
```




```
AVL D_rotate_L(AVL k3){
    k3.left = S_rotate_to_right(k3.right);
    return (S_rotate_to_left(k3));
}
```



Tree traversal :-

1) print inorder:-

```
Voiel inOroler (AVL T)
{
  if (T != null)
    {
      inorder(T.left);
      print (T.element);
      inoroler(T.right);
    }
}
```

2) print preorder :-

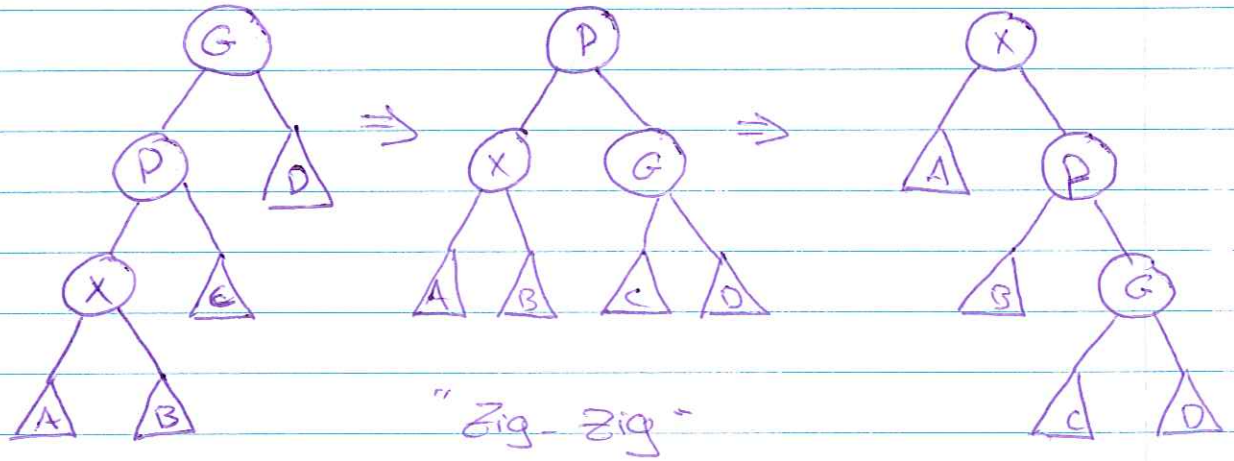
```
Voiel preOroler (AVL T)
{
  if(T != null)
    {
      print(T.element);
      preorder(T.left);
      preOroler(T.right);
    }
```

3) post order:-

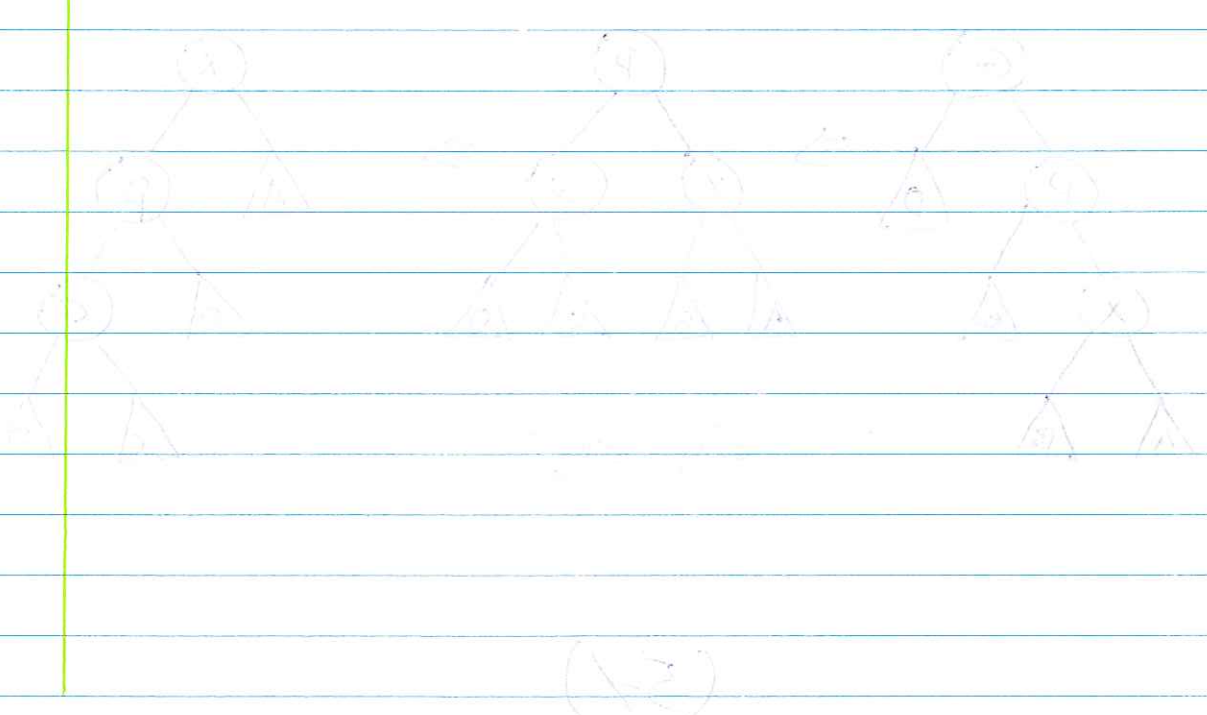
```
Void postOrder(AVL T)
{
  if (T != null)
    {
      postorder (T.left);
      postOrder (T.right);
      print (t. element);
    }
}
```

* Splay Trees :-
  is a binary search tree without the Balance condition.

1) left_left

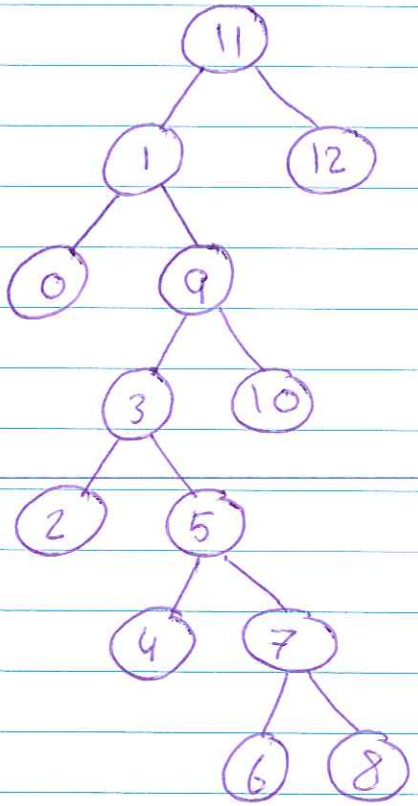


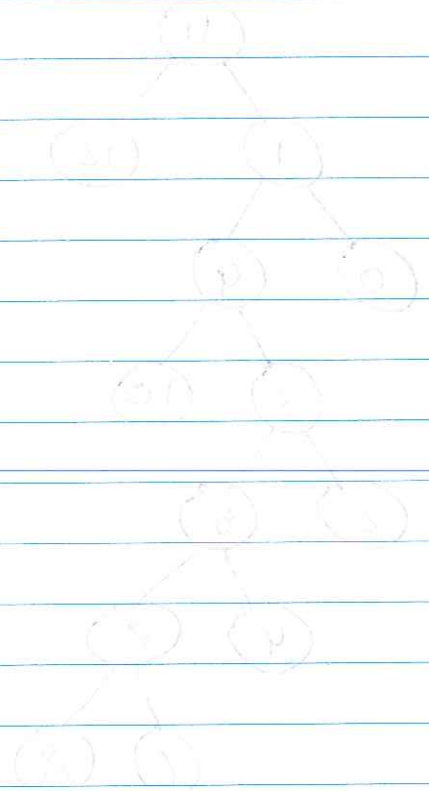
"Zig_Zig"



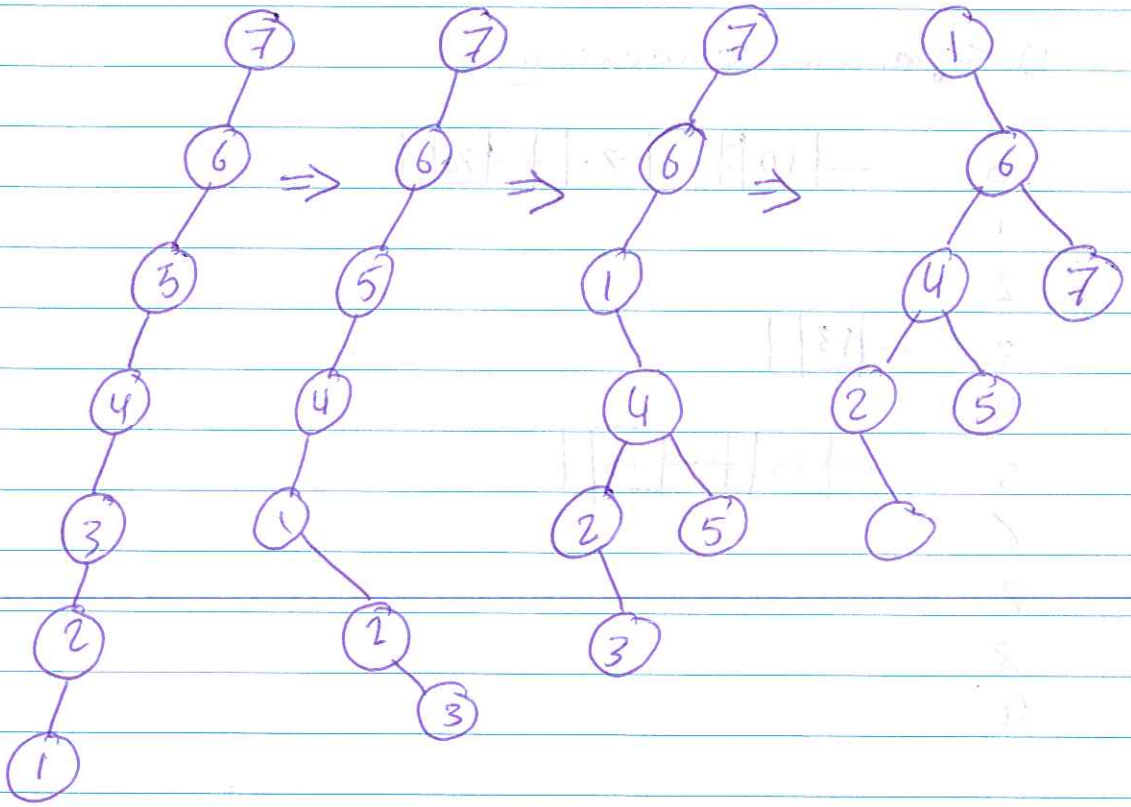
Example :-

find 7 ?



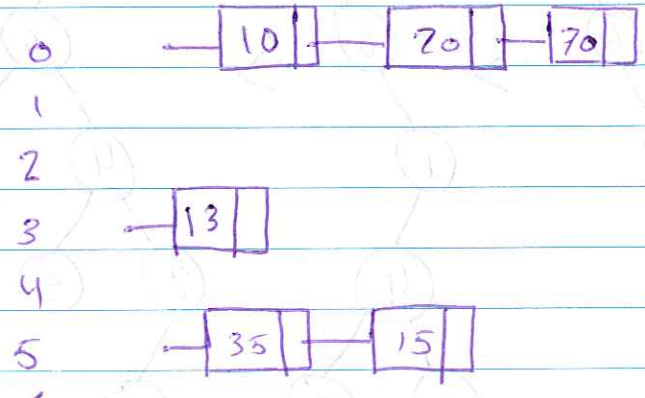


Find (1)



Hashing :-

1) Separate chaining



10, 20, 35, 15, 13, 70

10 % 10 = 0
20 % 10 = 0
35 % 10 = 5
15 % 10 = 5
13 % 10 = 3
70 % 10 = 0

"increase the number of pointers"

2) Open addressing :-

a) Linear probing

$h(x) = (x \% size + f(i)) \% size$, $i \geq 0$

Example :- 89, 18, 49, 58, 69

0	49
1	58
2	69
3	
4	
5	
6	
7	
8	18
9	89

$89 \% 10 + 0 = 9$

$18 \% 10 + 0 = 8$

$49 \% 10 + 0 = 9$

$+ 1 = 10 \% 10 = 0$

$58 \% 10 + 0 = 8$

$+ 1 = 9$

$+ 2 = 10 \% 10 = 0$

$+ 3 = 11 \% 10 = 1$

b) Quadratic probing :-

$$h(x) = (x \% size + i^2) \% size \quad , \; i \geq 0$$

c) Double hashing :-

$h(x) = (h_1(x) + i * h_2(x)) \% size$

Example :- 89, 18, 49, 58, 69

$h_1(x) = x \% size$
$h_2(x) = size - (x \% size)$
$size = 10$

$89 \% 10 + 0 * (\quad) = 9$

$18 \% 10 + 0 * (\quad) = 8$

$49 \% 10 + 0 * (\quad) = 9$ ×
$9 + 1(10 - 9) = 10 \% 10 = 0$

$58 \% 10 + 0 * (\quad) = 8$ ×
$8 + 1(10 - 8) = 10 \% 10 = 0$ ×
$8 + 2(10 - 8) = 12 \% 10 = 2$

$69 \% 10 + 0 * (\quad) = 9$ ×
$9 + 1(10 - 9) = 10 \% 10 = 0$ ×
$9 + 2(10 - 9) = 11 \% 10 = 1$

0	49
1	69
2	58
3	
4	
5	
6	
7	
8	18
9	89

30, 75, 47, 53, 98, 15, 6, 96, 27, 50, 82

1) Double Hashing 2) Quadratic Hashing

$h_1(x) = X \% \text{ size}$

$h_2(x) = \text{size} - (X \% \text{ size})$

size = ?

$\text{size} = 2 * n = 2 * 11 = 22 \Rightarrow \boxed{23}$

* String Hashing :-

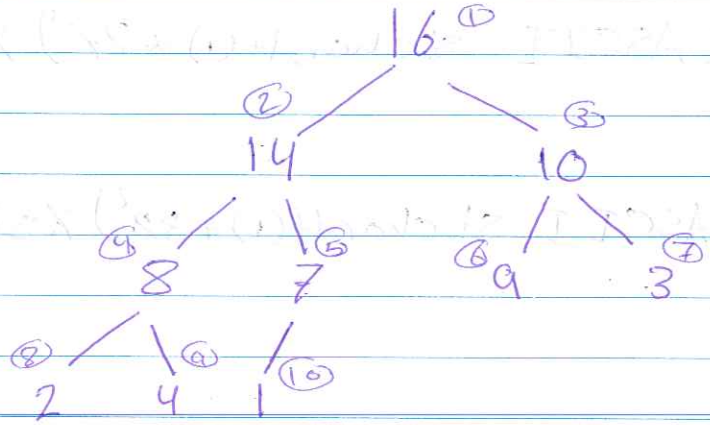
1. $h(st) = \left(\sum_{i=1}^{n} ASCII\ st.charAt(i)\right) \% size$

2. $h(st) = \left(\sum_{i=1}^{n} ASCII\ st.charAt(i) * 27^{i}\right) \% size$

3. $h(st) = \left(\sum_{i=1}^{n} ASCII\ st.charAt(i) * 32^{i}\right) \% size$

* Heaps:-

Example:- 16, 14, 10, 8, 7, 9, 3, 2, 4, 1



⇒ heap

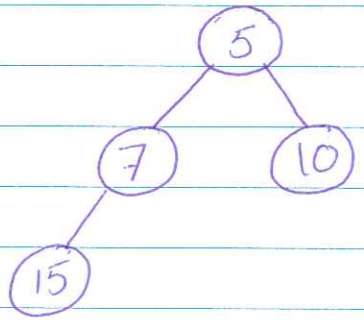
$i=1 \Rightarrow left = 1*2 = 2$

$right = 1*2+1 = 3$

$i=4 \Rightarrow left = 4*2 = 8$

$right = 4*2+1 = 9$

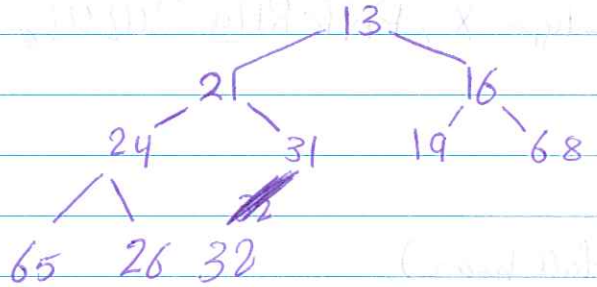
Example:- 10, 15, 7, 5 ⇒ min_heap



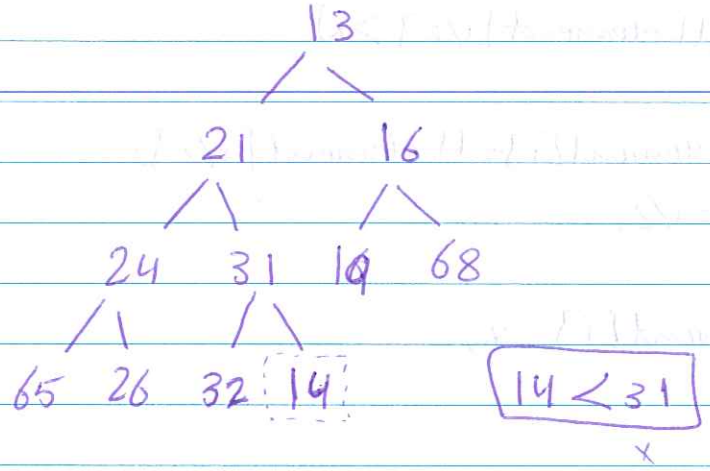
child ✗ Root से element छोटे

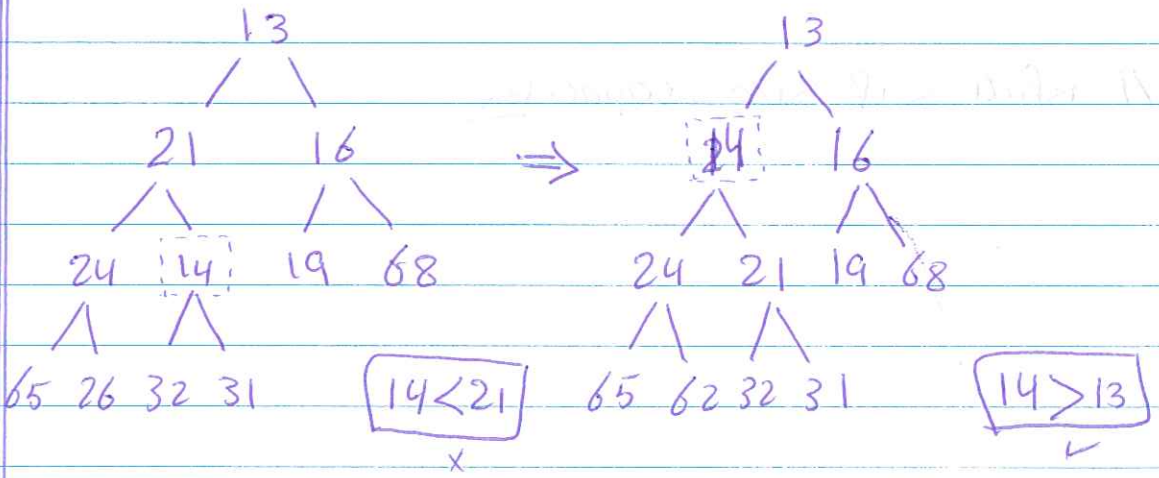
⇒ max_heap ⇒ child < Root element छोटे

Basic Heap operation :- "min-heap"



insert 14 so



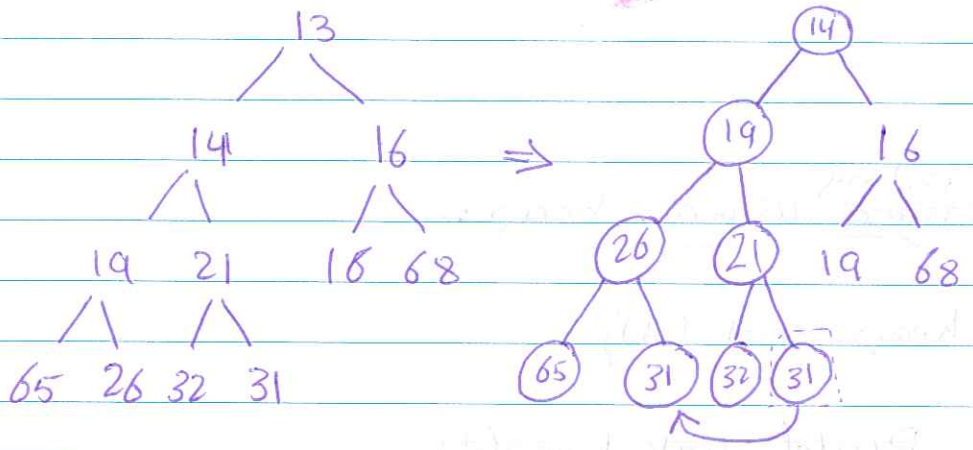


Insert code :- for min-heap

```
Void insert (element_type X, PRIORITY_QUEUE H)
{
  int i;
  if(isfull (H))
    System.out.println("full heap");
  else
    {
      i = ++H.size;
      while (H.element[i/2] >X)
        {
          H.element[i]= H.element[i/2];
          i=i/2;
        }
      H.element[i] =x;
    }
}
```

// isfull → if size = capacity

Delete Min :- "for min heap"




```
element_type delete_min (PRIORITY_QUEUE H)  O(log n)
{
    unsigned int i, child;
    element_type min_element, last_element;
    if (isEmpty (H))
    {
      System.out.print("empty Heap");
        return H.element[0];
    }
    min_element = H.element[0];
    last_element = H.element[size--];
    for(i=1 ; i*2 <= H.size ; i=child)
    {  child = i*2;
      if ((child != H.Size) && (H.element[child+1] <
                                H.element[child]))
        child ++;
      if ( last_element > H.element[child])
            H.element[i] = H.element[child];
      else
    {       break;
```

H. element [i] = Last_element;
return min_element;
}

* ~~Sorting~~ Sorting using heap :-

heap_sort (A){

Build_max_heap(A)
for (i = size(A) ; i >= 2 ; i--)
{ $O(n \log n)$
Swap (A(1), A(i));
heap_size (A) = heap_size(A) - 1;
ment_heap_prop(A, 1);
}

⇒ procedure ment_heap_prop(A, i) {

L = left(i);
R = Right(i)
if L <= heap_size and A[L] > A[i] then
Largest = L
else
Largest = i

if R <= heap_size and A[R] > A[largest] then
Largest = R;

```
if largest <> i then
    exchange (A[i], A[largest])
    mem_heap_prob (A, largest)

Build a heap tree
  heap_size_ arreys_size
  for(i = heap_size/2 ; i >= 1 ; i--)
    ment_heap_prop(A, i);
```

Sorting :-

1 merge sort:-

{5, 2, 4, 6, 1, 3, 2, 6}

{5, 2, 4, 6} {1, 3, 2, 6}

{5, 2} {4, 6} {1, 3} {2, 6}

{5} {2} {4} {6} {1} {3} {2} {6}

{2, 4, 5, 6} {1, 2, 3, 6}

left mid mid+1 right

A

2	4	5	6		1	2	3	6

left mid mid+1 right

B

```
procedure merge_sort (A, left, right)

Begin
    if (left < right)
      Begin
        mid = (left + right)/2
        merge_sort (A, left, mid)
        merge_sort (A, mid, right)
        merge (A, left, mid, right)
        end if
    end pro
```

```
procedure merge (A, left, mid, right)
    Begin
        i = left
        j = left;
        k = mid + 1
      while ((j <= mid) && (k <= right))
       Begin
          if (A[i] < A[k])
              B[i++] = A[j++];
          else
              B[i++] = A[k++];
        end while
```

```
if (j<=mid)
   for(k=j ; k<=mid; k++)
      B[i++] = A[k];
   else
    for (j=k ; j<=right; j++)
       B[i++] = A[j];
    end if
   for(i=left ; i<=right ; i++)
       A[i] = B[i]
    end for
endproc
```

time:

$$T(n) = \begin{cases} c_1 & , n=1 \\ 2T(n/2) + cn & , n>1 \end{cases}$$

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

⇒ disadvantages in Huge data
Because we need extra space
since we make anew array B

Quick sort :-

- Divide and conquer recursive algorithm:-

- The basic algorithm to sort an array S consists of the following four steps:-

1. If the number of elements in S is 0 or 1, then return.

2. pick any element V in S. This is called Pivot

3. partition S - {V} (The remaining elements in S) into two disjoint groups.

4. Return {QuickSort(S1) followed by V followed by QuickSort (S_2)}

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

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

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

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

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

0, 1, 4, 2, 5, 3 | 6 | 9, 7, 8

```
Void QuickSort(A, left, right){
    int i, j, pivot;
    if (left < right)
    {
        pivot = getpivot(A, left, right);
        i = left;
        j = right-1;
        while(i<j)
        {
            while(A[++i] < pivot)
            while(A[--j] >= pivot)
            if(i<j)
            exchange(A[i], A[j])
        }
        exchange(A[i], A[right-1]);
        QuickSort(A, left, i-1);
        QuickSort(A, i+1, right)
    }
}

int getPivot(A, left, right){
```