* Data structure is a way of organizing data in a computer in such a way that we can perform operations on these data in an effective way.
* It is a systematic way of organizing and accessing data.
* The way we organize information can have a lot of impact on the performance.
* 1) To write efficient algorithms for a given problem and 2) to store & retrive information in computer as fast as possible and 3) to compare algorithms written for same problem are advantages of data structure.
* Factors affecting running time of an algorithm.
  1) The input size.
  2) The hardware environment (processor speed, clock rate, Memory and disk)
  3) The software environment (OS, compiler and Programming Language)
* Among Time & space complexities, time is considered.
Best, worst and Average cases analysis of an algorithm-
Ex:- * In Linear search, if element is at 1st position, it is best case.
  * In Binary search, if it is at middle position, it is best case. Here, if it is at extreme ends, it is worst case.
  * In Linear search, if the element is at last position (or) if there is no element, it is worst case.

Space complexity:-
Any program will have the following segments
1) Instruction space 2) Data space 3) stack space.
* Space required for constants, variables, intermediate variables, dynamic variables, etc is called Data space.

Ex:- Float, int, array.

* Space required to store return address, return values, etc is called stack space.

* Space to store machine code generated by compiler is called Instruction space.

* Void main()
{
int a,b,c;
return (a+b+c)/3;
}

Here, it takes 16 bytes of memory

* int main()
{
int ~~k,n,i~~ K, a[5], i, P=0;
for(i=0; i<5; i++)
Scanf("%d", &a[i]);
Scanf("%d", &k);
for(i=0; i<5; i++)
if(a[i]==k)
{
P=1;
break;
}
if(P==0)
Printf("Element not found\n");
else
Printf("Element found at %d position\n", (i+1));
}

/* code for linear search */

Here, it takes 32 bytes of memory

03-07-19

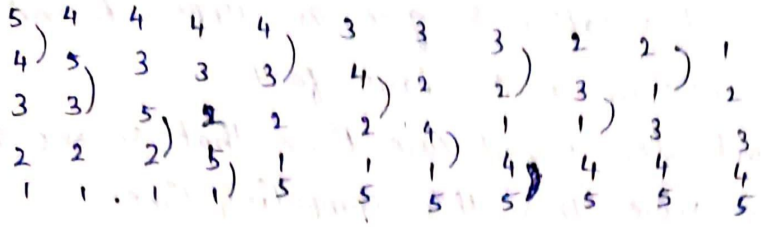
Time Complexity:-

* Methods are
1) Operation count
2) Step count
3) Assymptotic method
4) Practical method.

* The time taken by the main operation is operation count.

Ex:- In Sorting, among comparision & Swapping, comparision takes more time. So, comparision is main operation.

Ex:- For bubble sort,



4+3+2+1

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

Step count:- Time taken by each & every step is called step count. Each step takes 1 time.

1) int y; — ① } Time complexity = 2
y = a+b — ②

2) for (i=1; i<=4; i++) — ① ⑤ ④ } Time complexity = 14.
y = a+b; — ④

3) for (i=1; i<=n; i++) } Time complexity = 3n+2
y = a+b;

Assymptotic method:- The word assymptotic means the study of functions of a parameter 'n'. As 'n' becomes larger and larger without bounds, usually an algorithm that is assymptotically more efficient will be the best choice for all the inputs except very small inputs.

~~*write constant~~

~~Logarith~~

Assymptotic Functions:-

constant	– 1
Logarithmic	– log n
Linear	– n
n log n	– n log n
Quadratic	– n^2
Cubic	– n^3
Exponential	– 2^n

↓ Increasing order of time

Big Oh Notation (Oh) :-

We use Big Oh notation to give an upper bound of a function $f(n)$ within a constant factor.

Upperbound indicates worst case time, that means it does not consume more than this computing time.

* The time taken by that algorithm is constant, i.e; it's not depending on the problem size. This is the meaning of $O(1)$

Ex:- $f(n) = \frac{n(n-1)}{2} = \frac{n^2-n}{2} \rightarrow O(n^2)$

* Using step count, for bubble sort,

```
for(i=0 ; i<n-1 ; i++)          (1) (n) (n-1)
    for(j=0 ; j<n-i-1 ; j++)    (n-1) (n²) (n²)
        if a[j] > a[j+1]        (n²)
            Swap                (n² + n² + n²)
```

Time complexity $= 6n^2 + 3n - 1 \rightarrow O(n^2)$

5-7-19

* $n \rightarrow$ problem size.

* $f(n) = O(g(n))$, such that there exists two +ve constants 'c' and 'n_0' with a constraint that $|f(n)| \leq c \cdot |g(n)|$, $\forall\ n \geq n_0$.

→ Given, $f(n) = 3n+2$. Prove that $f(n) = O(n)$.

Sol $f(n) = 3n+2$

$3n+2 \leq c \cdot n$

$n=1 \Rightarrow 3 \cdot 1 + 2 \leq c \cdot 1$

$\Rightarrow 5 \leq c$, $c = 5, 6, 7, \ldots$

$n=2 \Rightarrow 3 \cdot 2 + 2 \leq c \cdot 2$

$\Rightarrow 8 \leq 2c$, $c = 4, 5, 6, 7, \ldots$

$n=4 \Rightarrow 14 \leq 4c$, $c = 4, 5, 6, 7, \ldots$

$n=10 \Rightarrow 32 \leq 10c$, $c = 4, 5, 6, \ldots$

$n = 100 \Rightarrow 302 \leq 100c$, $c = 4, 5, 6, 7, \ldots$

$\Rightarrow (3n+2) \leq 4 \cdot n$, $n \geq 2$ (or) $(3n+2) \leq 5n$, $n \geq 1$

$\Rightarrow c = 4$, $n_0 = 2$ $\Rightarrow c = 5$, $n_0 = 1$

→ Given $f(n) = 7n - 2$. Prove that $f(n) = O(n)$.

Sol $f(n) = 7n - 2$.

$7n - 2 \leq c \cdot n$

$n = 1 \Rightarrow 5 \leq c \cdot 1$, $c = 5, 6, 7, \ldots$

$n = 2 \Rightarrow 12 \leq 2c$, $c = 6, 7, \ldots$

$n = 3 \Rightarrow 19 \leq 3c$, $c = 7, 8, \ldots$

$n = 5 \Rightarrow 33 \leq 5c$, $c = 7, \ldots$

$n = 10 \Rightarrow 68 \leq 10n$, $c = 7, 8, \ldots$

$n = 100 \Rightarrow 698 \leq 100c$, $c = 7, 8, \ldots$

$\therefore 7n - 2 \leq 7 \cdot n$, for $n \geq 3$

$\Rightarrow c = 7$,

$n_0 = 3$.

→ Given $f(n) = 17n^3 - 5$, prove that $f(n) = O(n^3)$

Sol $f(n) = 17n^3 - 5 \Rightarrow 17n^3 - 5 \leq c \cdot n^3$

$n = 1 \Rightarrow 12 \leq c \cdot 1$, $c = 12, 13, 14, \ldots$

$n = 2 \Rightarrow 131 \leq c \cdot 8$, $c = \cancel{66, 67, 68, \ldots}$ $17, 18, 19, \ldots$

$n = 5 \Rightarrow 2120 \leq (125)c$, $c = \cancel{426, 425, \ldots}$ $17, 18, 19, \ldots$

~~$\therefore 17n^3 - 5 \leq 12 \cdot n$, for $n \geq 1 \Rightarrow c = 12$, $n_0 = 1$~~

$n = 100 \Rightarrow 16,999,995 \leq (10^6)c$, $c = 17, 18, 19, \ldots$

$\therefore 17n^3 - 5 \leq 17n$, for $n \geq 2$

$\Rightarrow c = 17$,

$n_0 = 2$

→ Given $f(n) = 10n^2 + 4n + 3$. Prove that $f(n) = O(n^2)$

Sol $f(n) = 10n^2 + 4n + 3$

$\Rightarrow 10n^2 + 4n + 3 \leq c \cdot n^2$

$n=1 \Rightarrow 17 \leq c \cdot 1$; $c = 17, 18, 19, \cdots$

$n=2 \Rightarrow 51 \leq c \cdot 4$; $c = ~~26, 27, \cdots~~ 13, 14, 15, \cdots$

$n=5 \Rightarrow 273 \leq c(25)$; $c = ~~55, 56, \cdots~~ 11, 12, 13, \cdots$

$n=10 \Rightarrow 1043 \leq 100c$; $c = ~~105, 106, \cdots~~ 11, 12, 13, \cdots$

$n=100 \Rightarrow 100,403 \leq 10000 \cdot c$; $c = 11, 12, 13, \cdots$

$10n^3 + 4n + 13 \leq 11n^3$, for $n \geq 5$; $c = 11$, $n_0 = 5$

* Reorder the following efficiencies from smallest to largest.

a. 2^n b. $n!$ c. n^5 d. 10,000 e. $n \log_{(2)} n$

Ans 10,000 $< n \log n < n^5 < 2^n < n!$ [Answer is reverse order]

* Reorder the following efficiencies from smallest to largest.

a. $n \log_2 n$ b. $n + n^2 + n^3$ c. 2^4 d. $n^{0.5}$

Ans $2^4 < n^{0.5} < n \log n < n^3$ [Answer is reverse order]

* Determine big Oh notation for the following:

a. $6 \log_2 n + 9n$ c. $5n^2 + n^{3/2}$

b. $3n^4 + n \log n$

Ans a. $O(n)$ b. $O(n^4)$ c. $O(n^2)$

(efficiency = growing faster)

Omega Notation (Ω):-

It gives lower bound.

The lower bound would imply that below this time the algorithm cannot perform better. The algorithm will take atleast this much of time (i.e; minimum time). It is represented using "Ω". It represents best case.

$f(n) = \Omega(g(n))$, if there exists +'ve constants 'c' and 'n_0', such that, $\forall\ n > n_0$, $|f(n)| \geq c \cdot |g(n)|$.

→ $f(n) = 3n+2$. Prove that $f(n) = \Omega(n)$.

Sol. $3n+2 \geq c \cdot n$

$n=1 \Rightarrow 5 \geq 1 \cdot c$, $c = 5,4,3,2,1$.

$n=2 \Rightarrow 8 \geq c \cdot 2$, $c = 4,3,2,1$.

$n=3 \Rightarrow 11 \geq c \cdot 3$, $c = 3,2,1$.

$n=10 \Rightarrow 32 \geq c \cdot 10$, $c = 3,2,1$

$n=100 \Rightarrow 302 \geq c \cdot 100$, $c = 3,2,1$.

$\therefore (3n+2) \geq 3n$, for $n \geq 1$

$\Rightarrow c = 3$, $n_0 = 1$

Theta(θ) Notation:-

For some functions, the lower bound & upper bound may be same. i.e; Ω & O will have same function. For example, finding the min or max in a given array takes computing time $\Omega(n)$ and $O(n)$. We have special notation to denote functions having same lower & upper bound. That is called θ notation.

$f(n) = \theta(g(n))$, if there exists +'ve constants c_1 & c_2, such that, $\forall\ n > n_0$ $c_1 \cdot |g(n)| \leq f(n) \leq c_2 \cdot |g(n)|$

→ $f(n) = 3n+2$. prove that $f(n) = \theta(n)$.

Sol $c_1 \cdot n \leq 3n+2 \leq c_2 \cdot n$

$\Rightarrow 3n+2 \leq 4 \cdot n$ for $n \geq 2$; $c_2 = 4$

$3n+2 \geq 3 \cdot n$ for $n \geq 2$; $c_1 = 3$

$3 \cdot n \leq 3n+2 \leq 4n$, for $n \geq 2 \Rightarrow n_0 = 2$

Linear loops:-

1) `i=1`
`loop (i<=1000)`
`application code`
`i=i+1;`
`endloop` ↓ n

2) `i=1000`
`loop (i>=1)`
`{ ...`
`i=i-1;` ↓ n
`}`

3) `i=1000`
`loop (i>=1)`
`{`
`code`
`i=i-2;`
`}`
↳ $\frac{n}{2} \Rightarrow O(\frac{n}{2}) = O(n)$

Logarithmic loop:-

1) i = 1
loop (i <= 100)
{
code
i = i * 2;
}

[1, 2, 4, 8, 16, 32, 64 = 7 ≈ $\log_2^{100}$]

2) i = 1000
loop (i >= 1)
{
...
i = i/2;
}

[1000, 210, 105, 62, 31, 15, 7, 3, 1 → 9 ≈ $\log_2^{1000}$]

3) Binary Search → $\log_2^n$ = O(log n)

Nested loop:-

1) Linear logarithmic loop

i = 1
loop (i <= 100)
{
j = 1;
loop (j <= 100) } log n } n
{
j = j * 2;
}
i = i + 1;
}

↳ nlogn

09-7-19

2) Quadratic loop:-

loop (i <= 1000)
{
j = 1
loop (j <= 1000)
{ j = j + 1;
}
i = i + 1;
}

↳ O(n²).

3) Dependent Quadratic

i = 0
loop (i < n)
{
j = i + 1
{
...
}
i++
}
i++

i = 0	1	2	3
j 1	2	3	4
2	3	4	
3	4		
4			

↳ $\frac{n(n-1)}{2} = \frac{n^2-n}{2}$ = O(n²)

4) cubic

loop
{
loop
{ loop
{
...
}
}
}

↳ O(n³)

* Consider the following functions:-

(i) f(n) = 2^n; (ii) g(n) = n^2 (iii) h(n) = n log n.

which of the following statements about the assymptotic behaviour of f(n), g(n), h(n) is true.

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

↓

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

False

b) $h(n) = O(f(n))$

↓

$n \log n = O(2^n)$

True

c) $g(n) = \Omega(f(n))$

↓

$n^2 = \Omega(2^n)$

False.

*

DATA STRUCTURES

DS

(Data is stored one after other)

Linear — Non-Linear

Ex:- Array, stack, Queue, Linked List. — Tree, Graph

Stack:-

This can be implemented with the help of array/linked list

Stack implementation using array:-

* LIFO - Last in First out
  FILO - First in Last out

* ADT - Abstract Data type — stack -> It is a class.

* 2 methods to update data from stack are

1. push (e) :- Add element 'e' to the top of the stack. If the stack is full, it should show an overflow message.
2. pop () :- It removes and returns the top most element of the stack, if the stack is empty, it shows an underflow message.

Accessing methods:-

1. top () :- returns the top element of the stack without removing.
2. size () :- returns number of elements in stack.
3. isEmpty () :- returns "1" when stack is empty & returns "0" when stack is not empty.

* To ease the programming, we use

1. isFull () :- returns '1' when stack is full & returns '0' when stack is not full.

2. display() :- It displays all the elements of the stack from bottom to top.

* Overflow condition is top == (maxsize-1)

→ void Push (int e)

```
{
  if (top == (maxsize-1))
      System.out.println("overflow");
  else
    {
      top++;
      a[top] = e;
    }
}
```

11-7-19

→ int isFull()

```
{
  if (top == (maxsize-1))
      return 1;
  else
      return 0;
}
```

→ 'bool' datatype can also be used for this.

** import java.util.*;

```
class stackx
{
  private int maxsize;
  private int[] a;
  private int top;
  public stackx(int size)
  {
    maxsize = size;
    a = new int [maxsize];
    top = -1;
  }
  public int isFull()
  {
    ....
    ....
  }
```

```
Public void push(e)
{

}
}
Public class stackpgm
{
  Public static void main (String args[])
  {
  stackx  s1 = new stackx (5);
  Scanner  s=new Scanner (System. in);
  do
   {
    System.out.println ("1.push, 2.pop...");
    option = s.nextInt ();
    switch (option)
     {
       case 1 ; S.O.Pln ("Enter element to be pushed to stac
        :  ⟶ e=s.nextInt ;
       case 2;  s1.push(e);
     } :::
   }
  }
 }
```

```
*→ Public int pop()
   { if ( top == -1)
        System.out.println ("Underflow");
     else
       { int x = a[top];
         top--;
         return x;
       }
   }
```

```
*→ Public int isEmpty ()
   {
     if (top == -1)
       return 1;
     else
       return 0;
   }
```

```
public int top_element()
{
    if (top == -1)
        return -1;
    else
        return a[top];
}
public int size()
{
    return (top+1);
}
public void display
{
    for (i=0; i<=top; i++)
        System.out.println(a[i]);
}.
```

Stack in

12-7-19

* To add an element, i.e; push function, time complexity is constant.

→ push(e) — O(1)
  pop() — O(1)
  top_element() — O(1)
  size() — O(1)
  isEmpty() — O(1)
  isFull() — O(1)
  display — O(n)

→ class java.util. ~~Scanner~~ Stack;

Size()	Size()
isEmpty()	empty()
Push(e)	push(e)
pop()	pop()
top_element()	peak()

Method	Return value	Stack
Push(5)	—	5
Push(3)	—	5 3
Size()	2	5 3
isFull()	0	5 3
Push(Pop())	3	5 3
Push(isEmpty())	0	5 3 0

Applications of stack:-

1. conversion of Infix expression to postfix expression.
2. conversion of Infix expression to prefix expression.
3. Evaluation of prefix & postfix expression.
4. checking paranthesis match (bracket match).
5. Reversing elements in array.
6. Maintaining Activation record at the time of recursive function.

* In infix expression, operator comes in between operands.

Ex:- a+b

* +ab → prefix expression

ab + → postfix expression.

* 5+2*4

5+[24*]

524*+

* 5+2*4

5+[*24]

+5*24

* a+b*(c-k)^(m+n)*f

a+b*[ck-]^[mn+]*f

a+b*[ck-mn+^]*f

a+[bck-mn+^*]*f

a+[bck-mn+^*f*]

abck-mn+^*f*+

** (A+B)*(C-D)^E*F

[AB+]*[CD-]^E*F

AB+*[CD-E^]*F

[AB+CD-E^*]*F

AB+CD-E^*F*

[post fix]

** (A-B^D)/(E-F)*G

[ABD^-]/[EF-]*G

ABD^-EF-/G*

(post-fix)

* A* (B+D) / E-F + (G+H-K)

A* [BD+] / E-F + [GH+K-]

[ABD+*] / E-F + [GH+K-]

[ABD+* E/] - F + [GH+K-]

ABD+* E/F- GH+K-+

[post fix]

* A* (B+D) / E-F + (G+H-K)

[+BD] / E-F + [+G-HK]

[*A+BD] / E- [+F+G-HK]

-/*A+BDE+F+G-HK.

* (A-B^D) / (E-F) *G

[-A^BD] / [-EF] *G.

[-A^BD] / [*-EFG]

/-A^BD* - EFG

[Pre-fix]

* (A+B)*(C-D)^E*F

[+AB] * [-CD] ^E*F

[+AB] * [^-CDE] * F

** +AB^-CDEF

[Pre-fix]

* A* (B+D) / E-F + (G+H-K)

A* [+BD] / E-F + [-+GHK]

[*A+BD] / [+-EF-+GHK]

/*A+BD+-EF-+GHK

[Pre-fix]

* a+b* (c-k)^(m+n)*f

a+b* [-CK] ^(+mn)*f

a+b* [^-CK+mn] *f

a+ [**b^-CK+mnf]

+a** b^-CK+mnf

[Pre-fix]

16-7-19

Evaluation of post fix expression using stack:-

1) Read post fix expression as a string

2) Process each character from string. If it is an operand, push it to the stack. If it is an operator, pop twice and perform the operation on 2nd poped out element operator 1st poped out element.

3) Then push the result back to the stack.

4) When the end of the string reaches, print the top-most element of the stack, which is the result of the expression.

Ex:- 5 + 3 x 2

5 3 2 * +

x = POP()
y = POP()
y * x
Push (3*2)

x = POP()
y = POP()
y + x
Push (11)

* AB+CD-E^*F*

4 3 + 2 1 - 2 ^ * 1 *

Incoming Symbol	Stack content	Remarks
4	4	operand, push into stack
3	4 3	operand, push into stack
+	7	pop twice, operation, Push (4+3)
2	7 2	operand, push into stack
1	7 2 1	operand, push into stack
-	7 1	Pop twice, operation, Push (2-1)
2	7 1 2	operand, push into stack
^	7 1	Pop twice, operation, Push (1^2)
*	7	Pop twice, operation, Push (7*1)
1	7 1	operand, push into stack
*	7	Pop twice, operation, Push (7*1)

Here, integer stack has to be declared.

Evaluation of Postfix expression:-

* * * + a b ^ - c d e f

* * + 4 3 ^ - 2 1 2 1

Incoming symbol	stack content	Remarks
1	1	operand, push into stack.
2	1 2	operand, push into stack
1	1 2 1	operand push into stack
2	1 2 1 2	operand, push into stack.

-	1 - 2 1		Pop twice, operation, Push (2 - 1)
^	1 1		Pop twice, operation, Push (1 ^ 2)
3	1 1 3		operand, push into stack
4	1 1 3 4		operand, push into stack.
+	1 1 7		Pop twice, operation, Push (4 + 3)
*	1 7		Pop twice, operation, Push (7 * 1)
*	7		Pop twice, operation, Push (7 * 1)

Evaluation.

Conversion of Infix to postfix:-

(, # → 0

+, - → 1

*, / → 2

^ → 3

* → K + A - B / D * E #

Incoming Symbol	stack	o/p postfix array	Remarks
Initial condition	#		
K	#	K	operand, store in postfix array
+	# +	K	Precedance (+) > prece (#), Push (+ 0) to stack
A	# +	K A	operand, store in postfix array
-	# -	K A +	Pre (-) ≯ Pre (+), POP and store in postfix array,
B	# -	K A B +	Pre (-) > Pre (#), Push (-) operand, store in postfix array
/	# - /	K A + B	Prec (/) > Pre (-), Push (/)
D	# - /	K A + B D	Operand, store in postfix array
*	# - *	K A + B D /	Prec (*) ≯ Prec (/), POP

			and store in postfix array
E	# - *	KA+BD/E	Prec (*) > Prec (-), Push operand, store in postfix array
#		KA+BD/E*-	Pop one by one and store in Postfix array till you reach #.

1) Scan the infix expression from left to right.
2) If it is an operand, directly store it in the postfix array, else if the precedence of the incoming symbol is greater than precedence of top element of stack, then push the incoming symbol to the stack, else Pop operator from stack until precedence of incoming symbol is less ~~or equal~~ to the precedence of operator on top of the stack.
3) Finally, push the incoming operator to the stack.
4) If the incoming Symbol is '(', directly push it to the stack without checking precedence.
5) If the incoming symbol is ')', pop out all the elements from stack until an opening bracket is encountered.
6) Repeat Step 2 to 5 until you complete the infix expression.
7) When '#' comes, Pop everything from stack and store one by one in the postfix array.

19/7/19

*→ A + (B - C/E) * D #

Symbol	Stack	Postfix array	Remarks
Initial condition	#		
A	#	A	Operand, store in postfix array
+	# +	A	Prec(+) > prec(#), Push(+) to stack

(	#+(	A	Push (( ) to stack.
B	#+(	AB	Operand, store in Postfix array.
-	#+(-	AB	Prec(-) > prec(( ), Push to stack.
C	#+(-	ABC	store in postfix array.
/	#+(-/	ABC	prec(/) > prec(-), Push to stack.
E	#+(-/	ABCE	Store in postfix array
)	#+	ABCE/-	Pop out all elements until an '(' is encountered.
*	#+*	ABCE/-	prec(*) > prec(+), Push it to stack.
D	#+*	ABCE/-D	Operand, store in Postfix array
#	#	ABCE/-D*+	Pop one by one & store in postfix array till you reach #.

① (A+B) * (C-D) ^ E * F to postfix

② A/B * C(-D+K to postfix.

①sol symbol	Stack	Postfix array	Remarks.
Initial condition	#		
(	#(		Directly push '(' to stack.
A	#(	A	Operand, store in Postfix array.
+	#(+	A	prec(+) > prec(( ), Push it to Stack.
B	#(+	AB	operand, store in Postfix array.
)	#	AB+	Pop out all elements until '(' is encountered.
*	#*	AB+	Prec(*) > prec(#), push it to stack.
(	#*(	AB+	Push '(' to stack.

C	# * (	AB+C	Operand, store in Postfix array.
-	# * (-	AB+C	prec(-) > prec(C), push (-) to stack.
D	# * (-	AB+CD	Operand, store in Postfix array.
)	# *	AB+CD-	Pop out all elements until, we encounter '('.
^	# * ^	AB+CD-	Prec(^) > Prec(*), Push to stack.
E	# * ^	AB+CD-E	Operand, store in Postfix array.
*	# * *	AB+CD-E^*	pop out until Prec(top) < Prec(*)
F	# *	AB+CD-E^*F	Operand, store in Postfix array.
#	#	AB+CD-E^*F*	Pop one by one & store in Postfix array until '#' come

```
class postfixval
{
  Public static void main(String args[])
  {

    Scanner scan = new Scanner(System.in);
    int i, res;
    char ch;
    int OP1, OP2;
    String str = "abc-*";
    for(i=0; i< str.length(); i++)
    {
      if(str.charAt(i) >= 'a' && str.charAt(i)
                                <= 'z')
      {
        System.out.print("enter the value for "
                          + str.charAt(i));
        int data = scan.nextInt();
        S.push(data);
      }
```

```
        else
        {
            op1 = s.pop(); //second operand is on top of 1st operand;
            op2 = s.pop();
            if (str.charAt(i) == '+')
                res = op2 + op1;
            else if (str.charAt(i) == '-')
                res = op2 - op1;
            else if (str.charAt(i) == '/')
                res = op2/op1;
            else if (str.charAt(i) == '*')
                res = op2 * op1;

            s.push(res);
        }
    }
    System.out.println(s.pop());
  }
}
```

→ s.push(str.charAt(i) - '0');

18-7-19

Public class Infixtopostfix

```
{
    //A utility function to check if the given character is operand
    static boolean isAlpha(char ch)
    {
        if ((ch >= 'a' && ch <= 'z') || (ch >= 'A' && ch <= 'Z'))
            return true;
        else
            return false;
    }

    static int prec(char ch)
    {
        switch (ch)
        {
```

```
case '(':
case '#':
    return 0;
case '+':
case '-':
    return 1;
case '*':
case '/':
    return 2;
case '^':
    return 3;
}
}
static void infixtoPostfix(String infix)
{
  int i,k=0;
  Stack st = new stack();
  st.push('#');
  for(i=0; i < infix.length(); i++)
  {
    char infi = infix.charAt(i);
    if(isAlpha(infi))
      System.out.print(infi);
    else if(infi=='(')
      st.push(infi);
    else if(infi==')')
    {
      while(st.topelement()!='(')
        System.out.print(st.pop());
      char m = st.pop();
    }
    else if(infi=='#')
    {
      while(st.topelement()!='#')
        System.out.print(st.pop());
    }:
```

```
else
 {
   while (!prec(infi)) <= prec(St.topelement()))
     {
       System.out.print(st.pop());
     }
     st.push(infi);
  }
}
Public static void main(String args[])
{
 String exp = "a+b*(c+d)#";
 infixTopostfix(exp);
 //return 0;
}
}
```

Validity of an expression:-

1) Infix:-

[a+b → 1) Rank = 1 (No. of operands − No. of operators)

2) c = 0

a+b
↓
c=0+1 → c=0+1=1
⇒c=1
c=1-1
⇒c=0]

(i) Rank of the expression should be '1'
(Rank = No. of operands − No. of operators).

(ii) set counter to 0. Increment the counter by "1" when it find an operand & decrement the counter by "1", when it finds an operator. The counter should get value '1' or '0'. If it gets any other value, it's an invalid expression.

19-7-19

2) Prefix:-

(i) Rank should be '1'.

(ii) Initialise counter to 0. If operand, counter+1; if operator, counter−1. Always, counter should be "<=1".

Ex:- (1)

c	+ab
0	-1 0 1

✓

(2)

c	ab+
0	1 2

✗

3) Postfix:-

(i) Rank = 1

(ii) counter >= 1

Ex:- (1)

ab+
0 1 2 1

✓

(2)

c	+ab
0	-1

✗

* → a+b*c → c*b+a#

Infix to prefix:-

* Accept a string & reverse it. Follow the same steps of infix to postfix conversion. Finally, reverse the resulti string.

Incoming symbol	stack	Postfix array
Initial condition	#	
c	#	c
*	#*	c
b	#*	cb
+	#+	cb*
a	#+	cb*a
#	#	cb*a+

∴ prefix expression is +a*bc

* {[()]} valid ; {[()} invalid

Paranthesis match:-

if {, (, [

Push();

else if (')')

```
a = POP()
if (a == '(')

else if ('[')
    :
```

Recursion:-

* Function which calls itself is called a Recursive function
  1) There should be a terminating condition.
  2) The function call should near the stopping condition.

* Types of Recursion are
  1) Linear Recursion
  2) Binary Recursion
  3) Multiple Recursion.

* The recursion function call will be once in the body of the function in 1).

* Linear recursion:- The recursive function call statement will be called only once in the body of the function.
  Ex:- factorial

* Binary recursion:- If there are 2 recursive function call statements, then it is binary recursion.
  Ex:- fibonacci [fib(n-1) + fib(n-2)]

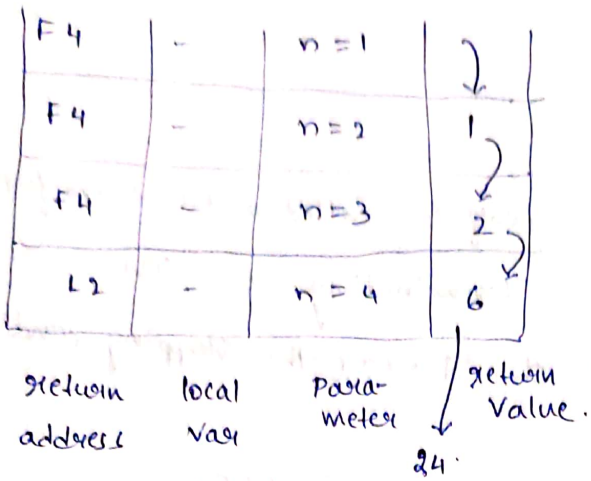
* Multiple recursion:- More than 2 recursive function call statements.

fact:-

```
F1  int fact (int n)
F2  { if (n==1)
F3        return 1;
    else
F4        return n * fact(n-1);
    }
```

* Activation record maintains return address, value of local variables, value of parameters, return value.

Eg:- L1 main()
{
L2 fact(4)
}

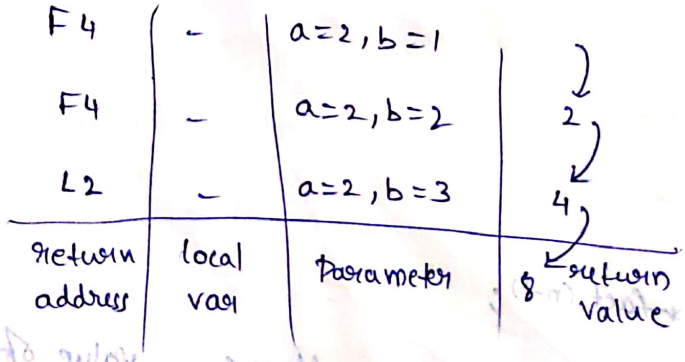


* Write the function for finding (a^b), using recursion & show how the activation records are maintained in the stack.

```
F1 int power(int a, int b)
   {
F2   if (b==1)
        return a;
F3   else
F4      return a * power(a, b-1);
F5 }
```

Activation record :-

```
L1 main()
   {
L2 Power (2,3);
   }
```

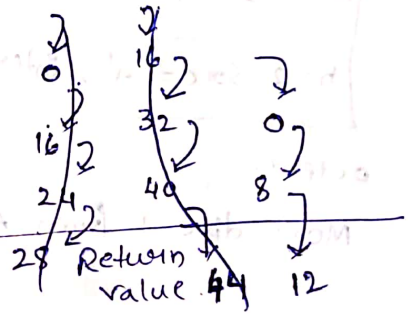


23-7-19

* Maintain the activation record for the following recursive function.

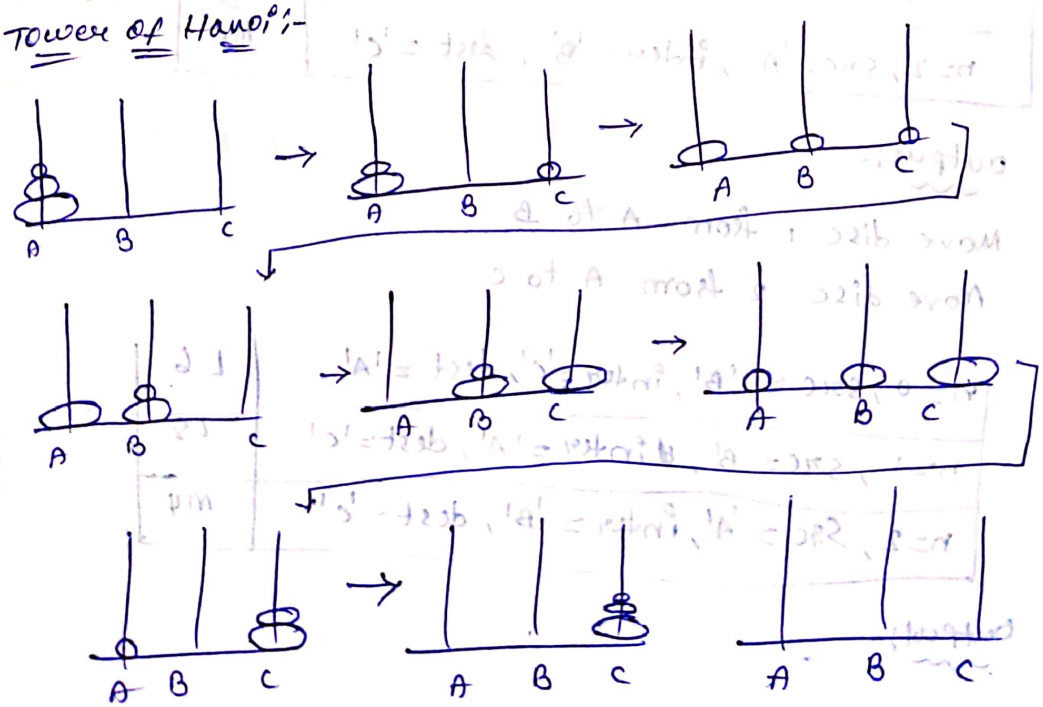
```
F1 fun (int a, int b)
   {
F2   if (b == 0)
        return 0;
F3   if (b%2 == 0)
        return fun (a+a, b/2);
F4   else
        return (a+fun(a+a, b/2));
   }
L1 main()
   {
L2   fun(4,3);
   }
```

~~F2~~	-	~~b=0, a=16~~	
F4	-	a=16, b=0	
F4	-	a=8, b=1	
L2	-	a=4, b=3	
Return address	Local Variable	Parameter	



* Let 'n' be no. of discs. Then $(2^n - 1)$ moves are required

Towers of Hanoi:-



```
L1  void hanoi(int n, char src, char inter, char dest)
L2  {
L3      if (n != 0)
L4      {
            hanoi(n-1, src, dest, inter);
L5
L6          System.out.println("move disc "+n+" from "+src+" to "+dest);
L7          hanoi(n-1, inter, src, dest);
L8      }
    }
M1  main()
M2  {
M3      hanoi(2,'A','B','C');
M4  }
```

n=0, src='A', inter='B', dest='C'	L6
n=1, src='A', inter='C', dest='B'	L6
n=2, src='A', inter='B', dest='C'	M4

output:-

Move disc 1 from A to B

n=0, src='C', inter='A', dest='B'	L8
n=1, src='A', inter='C', dest='B'	L6
n=2, src='A', inter='B', dest='C'	M4

output:-

Move disc 1 from A to B

Move disc 2 from A to C

n=0, src='B', inter='C', dest='A'	L6
n=1, src='B', inter='A', dest='C'	L8
n=2, src='A', inter='B', dest='C'	M4

Output:-

Move disc 1 from A to B
Move disc 1 from A to B
Move disc 2 from A to C
Move disc 1 from B to C

n = 0, src = 'A', inter = 'B', dest = 'c'	L8
n = 1, src = 'B', inter = 'A', dest = 'c'	L8
n = 2, src = 'A', inter = 'B', dest = 'c'	ML

24-7-19

* Prefix to infix is same as evaluation but brackets are to be kept.

QUEUE :-

* It is FIFO / LILO (Last in last out)
* It is a linear data structure.
* It has 2 variables, front, rear → to find last insertion element
  ↓ to find 1st inserted element.
* Insertion function is through rear end. (enqueue)
* Deleting function is through front end. (dequeue)
* enqueue() and dequeue() are updating methods.
* isEmpty(), isFull(), Front_Element(), display(), size() are accessing methods.
* Only if we are pushing 1st time, both front, rear are incremented to '0'. [Initially, rear = -1, front = -1]

```
* enqueue (int x)
{
  if (r == maxsize-1)
        S.o.pln("Overflow");
  else if (f == -1 || r == -1)
        { f++;
          r++;
          a[r] = x;
        }
```

```
        else
        {
            r++;
            a[r]=x;
        }
    }
```

25-7-19

* class Queue

```
{
    Private int maxsize;
    Private int front, rear, a[];
    Public void Queue (int s)
    {
        maxsize = s;
        a = new int[s];
        f = r = -1;
    }
Public void enqueue (int x)
{
    if( r == (maxsize-1)
        S.o.pln ("overflow");
else if (f==-1 || r==-1)
    {
        f++; r++; a[r] = x;
    }
    else
    {
        r++; a[r] = x;
    }
}
Public int dequeue ( )
{
    if (f == -1)
        S.o.pln ("underflow");
    else if (f==r)
    { int x = a[f];
        f = r = -1;
        return x;
    }
else
```

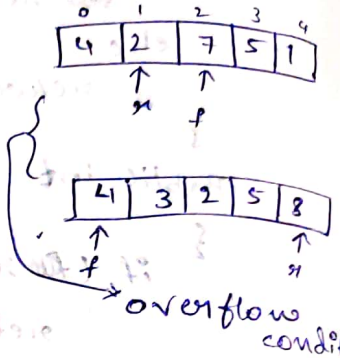
* constructer is used to initialise data members. It should be made Public because, we need to have access from main.

```
{
    int x = a[f];
    f++;
    return x;
  }
}
Public int isEmpty()
{
  if (f == -1)
     return 1;
  else
     return 0;
}
Public int isFull()
{
  if (r == (maxsize-1))
     return 1;
  else
     return 0;
}
Public void display()
{
  for(int i=f; i<=r; i++)
     S.O.P("%d", a[i]);
}
Public int size()
{ if (f==-1)
     return 0;
  else
     return (r-f+1);
}
Public int Front_element()
{
  if (f==-1)
     return -1;
  else
     return a[f];
}
}
```

* If we have only 1 element at the last position, then if we try to insert 1 more value, it tells overflow. This is one of the drawbacks. This is a simple queue. To overcome this, we use circular queue.

Circular Queue:-

```
Void enqueue (int x)
{ if (f==(r+1) % maxsize)
      S.o.pln ("overflow");
  else if ((f==-1) || (r==-1))
      {
        f++; r++;
        a[r] = x;
      }
  else
      {
        r++;          r=(r+1)% maxsize;
        a[r] = x;     a[r] = x;
      }
}
```



30-7-19

circular dequeue:-

```
int dequeue ()
{
  if (f==-1)
      S.o.pln ("Underflow");   (or)  return -1;
  else if (f==r)
      {
        int p = a[f];
        f = r = -1;
        return p;
      }
  else
      {
        int x = a[f];
        f = (f+1) % maxsize;
        return x;
      }
}
```

* To find size,

```
* int size
  {
    if (f==-1)
        return 0;
    else if (f<=r)
        return (r-f+1);
    else
        return
          (r-f+1+maxsize);
  }
```

* r is at 3rd position (starting from 0), f is at 0th position → no. of elements = 4

* display()

```
{ for (i = f; i != r; i = (i+1) % maxsize)
    {
      S.O.println(a[i]);
    S.O.println(a[i]);
}
```

* Double Ended Queue (deque (pronounced as dek)):-

It has 4 functions for updation. They are

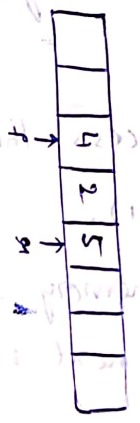
1) insert_front()
2) insert_rear()
3) delete_front()
4) delete_rear()

* int delete_rear()

```
{ if (f==-1)
    S.O.p("underflow");
  else if (f == r)
    {
      int x = a[r];
      r = f = -1;
      return x;
    }
  else
    { int x = a[r];
      r--;
      return x;
    }
}
```

* void insert_front(int x)

```
{
  if (f==0)
    S.O.p("Overflow");
  else if (f==-1 || r==-1)
  {
    r++;
    f++;
    a[f]=x;
  }
  else
    {
      f--;
      a[f]=x;
    }
}
```



* Except for display(), all others in Queue a have time complexity of O(1).

LINKED LIST :-

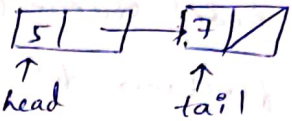
* A node has two fields that has 2 fields

1) data storage
2) To store address of next node
↓
To establish connection between nodes.

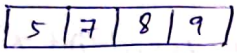
* A pointer that is initialised to null is called null pointer.

* 

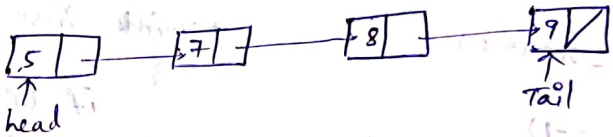
. To establish connection b/w the two



* head indicates head node & tail indicates tail node.

* 

is in linked list as



* Draw back of this is we can't access the elements directly like that of an array [a[0], a[5]]. Here, we have only 1 access point.

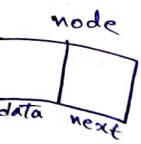
* Time complexity to access the element in linked list is O(n) & in array is O(1).

* Arrays occupy less memory space compared to linked list. Since it stores address of next node, to store an element, it takes space twice that of the element size.

~~1) can't access the elements directly like arrays. 2) In array, time complexity to~~

Advantages of linked list :-

* Dynamic memory allocation → run time → Memory can be saved.
* Static memory allocation → compile time
* If we implement stack and queue using linked lists, we'll not have the case of overflow.
* Linked list is dynamic memory allocation. So, there is



s.

ull pointer.

two

lements

e have

list is

nked list

t, it

~~ty to~~

Saved

e'll

no wastage of memory.

* In array, to add an element in between, time complexity is O(n), whereas in linked list, it is O(n). so, time complexity is same.

31-07-19

```
* import java.util.* ;
  class LNode  // class definition for a node in LL
  {   Public int info; // stores the data
      Public LNode next ; // Link pointer to the next node in list
      Public LNode (int item) // constructor
        {   info = item;
            next = null;
        }
  }
  class SList
  {   Public LNode head,tail; //pointer to head & tail nodes.
      Public SList()
      {   head = tail = null;
      }
      Public boolean isEmpty()
      {
```

/* Methods used to ~~initialise~~ ~~are~~ update are

1. insertfront (int x). 2. insert ~~after~~ last (int x)
3. insertafter (int x, int y) 4. delete front()
5. deletelast() 6. deleteElement()
7. deleteAfter (int x) 8. int deleteBefore (int x)
9. isEmpty() 10. display() 11. int count()
12. int Search (int x) */

```
        return head == null;
      }
```

Time complexity = O(1)

```
      Public void insertfront (int x)
      {   LNode temp = new LNode (x);
          if (head == null)
              head = tail = temp;
```

:xt != null.
*/

if the found, esent.

LINKED

* A node

1,

2

01-08-19

* A poin

* [5]

* head

tail and

* [5]

Time complexity = O(1)

Time complexity = O(1)

else
{ temp.next = head;
head = temp;
}
}

Public void insertlast (int x)
{
LNode temp = new LNode(x);
if (head == null)
head = tail = temp;
else
{ tail.next = temp;
tail = temp;
}
}

Public int deletefront()
{
if (head == null)
return -1;
else if (head == tail)
{
int x = head.info;
head = tail = null;
return x;
}
else
{ int x = head.info;
head = head.next;
return x;
}
}

Public int deletelast()
{
if (head == null)
return -1;
else if (head == tail)
{
int x = head.info;
head = tail = null;
return x;
}

* Drac
directly
only
* Time
O(n)
* Arrays
Since
1) can't
dise
takes
Advanta
*Dynamic
*Static
*If we
not hav
Linked

else
{

Time complexity = O(n)

Public

{

/* To reach

* Imple
LL is
return

```
else
{
    int x = tail.info;
    LNode t = head;
    return x;
    while (t.next != tail)
    {
        t = t.next;
    }
    t.next = null;
    tail = t;
    return x;
}
}
```

Time complexity = O(n)

```
Public void display()
{
    LNode t = head;
    while (t != null)
    { S.O.Pln (t.info);
      t = t.next;
    }
}
```

/* To reach 3rd from last, we write t.next.next != null. */

```
Public int count()
{
    LNode t = head;
    int c = 0;
    while (t != null)
    {
        c++;
        t = t.next;
    }
    return c;
}
```

* Implement search function which returns -1 if the LL is not existing, return 0 if element is not found, return the node number if the element is present.

```
Public int Search()
{
    LNode t = head;
    int c = 1;          (Or) c = 0
    if (head == null)
        return -1;
    while (t != null)
    {
        if (t.data info == key)
            return c;        (Or) return t;
        c++;
        t = t.next;
    }
    return 0;
}
Public void insertafter (int x, int y)
{
    if (head == null).
        System.out.println ("Can't insert");
    else
    {
        LNode temp = new LNode (y);
        LNode t = head;
        while (t != null && t.data next != x)
            t = t.next;          /* Always prefer t!=null first */
        if (t == null)
            System.out.println ("Can't insert");
        else
        {
            temp.next = t.next;
            t.next = temp;
        }
    }
}
Public void insertbefore (int x, int y)
{
    if (head == null)
        System.out.println ("Can't insert");
    else
    {
        LNode temp = new LNode (y);
        LNode t = head;
```

```
if (head.info == x)
  {
    temp.next = head;
    head = temp;
  }
else
 {
   while (t.next != null && t.next.info != x)
     t = t.next;
   if (t.next == null)
     System.out.println ("Can't Insert");
   else
     {
       temp.next = t.next;
       t.next = temp;
     }
 }
```

16-08-19

```
int deleteafter (int x)
{
  if (head == null)
     return -1;
  if (x == tail.info)
     return -2;
  Lnode t = head;
  while (t != null && t.info != x)
     t = t.next;
  if (t == null) return -3;
  else
  {
     int x = t.next.info;
     t.next = t.next.next;
     if (t.next == null)
        tail = t;
     return x;
  }
}
```

```
int deletebefore(int x)
{
  if (head == null) return -1;
  if (x == head.data) return -2;
  Lnode t1 = head;
  Lnode t2 = t1;
  while (t1.next != null && t1.next.info != x)
  {
    t2 = t1;
    t1 = t1.next;
  }
  if (t1.next == null) return -3;
  else
  {
    int y = t1.info;
    t2.next = t1.next;
    return y;
  }
}
```

[or]

```
if (t1 == head)
{ int y = deletefront();
  return y;
}
else
{ int y = t1.data;
  t2.next = t1.next;
  return y;
}
```

t1 = head
t2 = t1
everything is same except

```
{
int y = t1.data;
head = t1.next;
t1.next = null;
return y;
}
```

Time complexities :-

	head & tail	head
insertfront()	O(1)	O(1)
insertlast()	O(1)	O(n)
deletefront()	O(1)	O(1)
deletelast()	O(n)	O(n)
deletebefore()	O(n)	O(n)
deleteafter()	O(n)	O(n)

insert after ( )	O(n)	O(n)
insert before ( )	O(n)	O(n)
delete node ( )	O(n)	O(n)
search ( )	O(n)	O(n)
count ( )	O(n)	O(n)
display ( )	O(n)	O(n)

20-8-19

① Skip alternate nodes and store in another linked list :

② display the even elements from the linked list.

③ print the odd position elements.

② Sol.

```
void even_elements()
{
    Lnode t = head;
    while (t!=null)
    {
        if (t.info %2==0)
            S.o.p(t.info+" ");
        t = t.next;
    }
}
```

③ Sol

```
void odd-positions()
{
    Lnode t = head;
    while (t!=null && t.next!=null)
    {
        S.o.p(t.info+" ");
        t=t.next.next;
    }
    while (t!=null)
    {
        S.o.p(t.info+" ");
        if (t==tail)
            t=t.next;
        else
            t=t.next.next;
    }
}
```

① Sol.

```
Main()
{
  List L, L1;

  L.alternate(L1);
  L.display();
}
alternate(list L1)
{
  Lnode t = L1.head;
  Lnode temp = new Lnode(t.data);
  head = temp; // here, head is of list L.
  t = L.next.next;
  while(t != NULL)
  {
    temp.next = new Lnode(L.data);
    temp = temp.next;
    t = t.next.next;
  }
  tail = temp;
}
```

Drawback is it fails for odd no. of elements

④ consider the linked lists L1, L2 have been created. Now, concatenate both the linked lists & store the resultant linked list in L1.

Sol

```
Concatenate(List L2)
{
  tail.next = L2.head;
  tail = L2.tail;
}
```

⑤ consider the linked list L1 and split that linked list into 2 linked lists L1 and L2.

Sol

```
main()
{
 List L1, L2;
 L1.split(L2);
```

```
split ( List L2)
{
    Lnode t = head;
    int n = count();
    for(int i=1; i<n/2; i++)
        k = k.next;
        L2.head = k.next;
        k.next = NULL;
         L2.tail = tail;
          tail = k;
}
```

Drawback is
if only 2 nodes
it fails.

21-8-19

Reversing a linked list :-

```
Void reverse()
{
    Lnode prev = null;
    Lnode current = head;
    Lnode n;
     while ( current != null)
      {
          n = current.next;
          current.next = prev;
           prev = current;
            current = n;
       }
      tail = head;
      head = prev;
}
```

Split the linked list into 2 without using count();-

```
split (List L2)
{
    Lnode k1 = head;
    Lnode k2 = k1.next;
     while (k2.next != Null)
      {
          k2 = k2.next.next;
           k1 = k1.next;
       }
      L2.head = k1.next;
```

```
        L2.tail = tail;
        k1.next = null;
        tail = k1;
    }
```

split the list into 3

```
split (list l2, list l3)
{
    Lnode k1 = head;
    Lnode k2 = k1.next;
    Lnode k3 = k2.next;
    while (k3.next != NULL)
    {
        k3 = k3.next.next.next;
        k2 = k2.next.next;
        k1 = k1.next;
    }
    L2.head = k1.next;
    k1.next = NULL;
    tail = k1;
    L3.head = k2.next;
    k2.next = NULL;
    L2.tail = k2;
    L3.tail = k3;
}
```

Insert by order;-

27-8-19

Recursive function display:-

```
Void disp (node t)
{
    if (t == NULL) return ;
    else
    {
        s.o.p ln (t.data);
        disp (t.next);
    }
}
```

* Implement stack with the help of a singly linked list.

(a).

```
push()
{
  insertlast();
}

pop()
{
  deletelast();
}
```

(or)

```
Push
{
  insertfront();
}

pop()
{
  deletefront();
}
```

Queue:- [using linked list]

```
Enqueue()
{
  insertfront();
}
dequeue()
{
  deletelast();
}
```

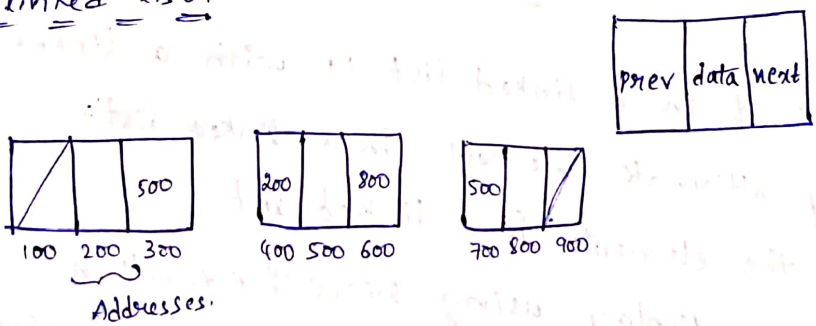
Here,
tail.data is front element.

[or]

```
Enqueue()
{
  insertlast();
}
dequeue()
{
  deletefront();
}
```

Here,
head.data is front element.

Doubly linked list:-




```
class duode
{
  Public:
   duode next;
    int data;
    duode prev;
    duode(int x)
    {
      data = x; next = prev = null;
    }
};
```

```
class dlist
{
  dnode head, tail;
  public:
  dlist()

  void insertfront(int x)
  {
    dnode temp = new dnode(x);
    if(head == NULL)
    { head = tail = temp;
    }
    else
    {
      temp.next = head;
      head.prev = temp;
      head = temp;
    }
  }
```

* `tail.prev.next = null;`
  `tail = tail.prev;`

Problems:-

1. Replace $ in a linked list L1 with a linked list L2.
2. Display alternate elements in a linked list.
3. sort the elements in a linked list
4. Reverse display using recursive function.
5. Swap the nearby elements in a linked list.

   [1→2→3→4→5→6
     2→1→4→3→6→5]
6. Find largest & smallest elements from linked list.
7. Find the union of 2 linked lists.
8. Find the intersection of 2 linked lists.
9. ~~List~~ Separate -'ve & +'ve elements [from L1 to L2, L3]

(10) Merge 2 linked lists in order.

29-8-19

circular linked list:-

```
class cnode
{
    int data;
    cnode next;
    Public cnode(int x)
    {
        data=x;
        next=null;
    }
}
class clist
{
    cnode tail;
    Public clist()
    {
        tail=null;
    }
insertfront(int x)
{
    cnode temp=new cnode(x)
    if(tail==null)
    {
        tail=temp;
        tail.next=tail;
    }
    else
    {
        temp.next=tail.next;
        tail.next=temp;
    }
}
insertlast(int x)
{
    cnode temp=new cnode(x)
    if(tail==null)
    {   tail=temp;
        tail.next=tail;
    }
```

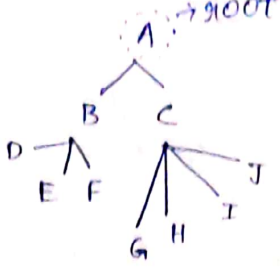
```
deletefront()
{
    if(tail==null)
        return -1;
    else if(tail.next==tail)
    {
        x=tail.data;
        tail=null;
        return x;
    }
    else
    {
        x=tail.next.data;
        tail.next
            =tail.next.next;
        return x;
    }
}
```

```
deletelast()
{
    if(tail==null)
        return -1;
    else if(tail.next==tail)
    {
        x=tail.data;
        tail=null;
        return x;
    }
    else
    {
        x=tail.data;
        cnode t=tail.next;
        while(t.next!=tail)
            t=t.next;
        t.next=tail.next;
    } tail=t; return x;
```

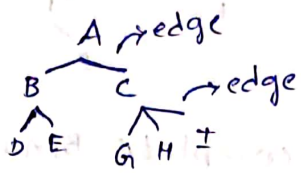
```
else
{
    tail.next = temp;
    temp.next = tail.next;
    tail.next = temp;
    tail = temp;
}
}
int deleteafter(int x)
{
if (tail == null) return -1;
if (x == tail.data)
{
    if (tail.next == tail)
    {
        int p = tail.data;
        tail = null;
        return p;
    }
    else
    {
        int p = tail.next.data;
        tail.next = tail.next.next;
        return p;
    }
}
cnode t = tail.next;
while (t != tail && t.data != x)
    t = t.next;
if (t == tail) return -3;
else
{
    int p = t.next.data;
    t.next = t.next.next;
    if (t.next == tail)
        tail = t;
    t.next = t.next.next;
} }  return p;
```

```
public int deletebefore(int
{
if (tail == null) return
if (x ==
cnode t1 = tail.next, t2
if (x == tail.next.data
{
    int p = tail.data;
    while (t.next != tail)
        t = t.next;
    t.next = tail.next;
    tail = t;
    return p;
}
while (t1.next.data != x)
{ if (t1 == tail) break;
    t2 = t1; t1 = t1.next;
}
if (t1 == tail) return
else
{
    int p = t1.data;
    if (t1 == tail.next)
        tail.next = t1.next;
    else
        t2.next = t1.next;
    return p;
}
}
```

Tree



* It is a non-linear data structure.
* Data is organised hierarchically.
* Root :- The node which is not having any parent is called as root.
* Leaf :- The node which is not having child is called as leaf.
* Internal node :- The node which has any child is called as internal node.

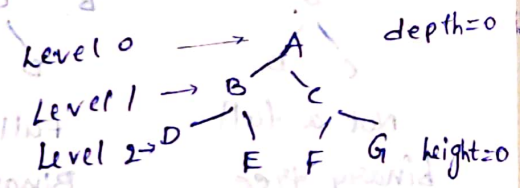


* External node :- ~~The node to A~~ A leaf is an external node.
* Here, (A, B, C) are internal nodes
  (D, E, G, H, I) are external nodes (or) leaves.
* Sibling :- The child of same parent is called sibling.
* G & H are siblings ; H and D are not siblings.

30-8-19

* In linked list, time complexity to search an element is O(n). In tree, since the data is distributed in several branches hierarchically, the time complexity to search an element can be reduced.
* ~~Tree~~ Edge :- An edge of a tree T is a pair of nodes (u, v) such that

* Path, ancestor, ordered tree.
* Depth of P :- Number of ancestors of P other than P itself.
* Depth of root is 0
* Height of leaf is 0



* depth(P) = 1 + depth(parent of P) ; depth(root) = 0.
  This is to write a recursive function.

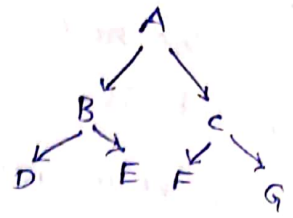
Types:-

* Binary tree ; Binary search Tree ; AVL Tree ;
  B tree ; B+ tree

Binary tree:-

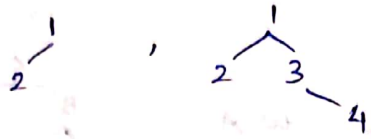
- At most 2 children

* A node can have 0 (or) 1 (or) 2 children.



Binary tree

Ex:-

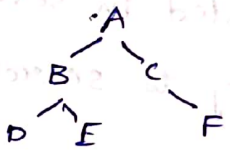


→ complete Binary tree — fill by levels

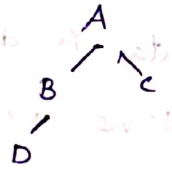
→ Full Binary tree - 0 or 2 child

→ perfect Binary tree - All node has 2 child & leaves are filled in same level

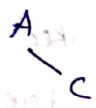
* complete Binary tree
  ↳ Top to bottom, left to right [filling order]



Not a complete Binary tree

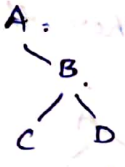


complete Binary tree

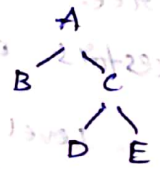


Not a complete Binary tree

* Full Binary tree
  ↳ Proper binary tree.

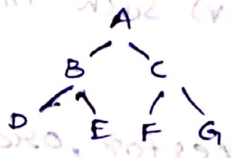


Not a full binary tree

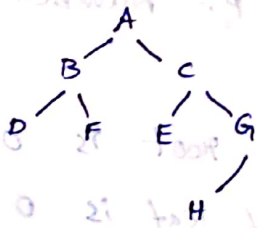


Full Binary tree

* perfect binary tree

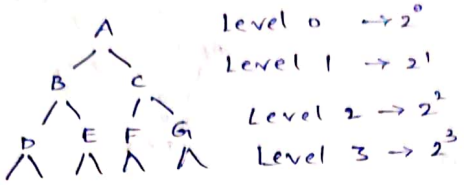


perfect binary tree



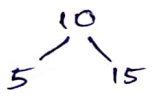
Not a perfect binary tree

* ordered binary tree.
* For a binary tree, each level can have maximum 2^l nodes.
* Number of external nodes in a binary tree = No. of internal nodes + 1

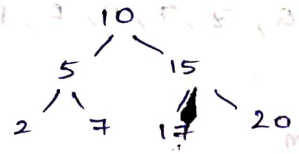


↓
This is for a full binary tree only.

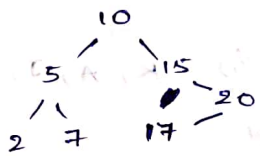
Binary Search Tree (BST):-
Here, left child will have value less than root node and right child will have value greater than root node.



Binary Search Tree



Not a BST

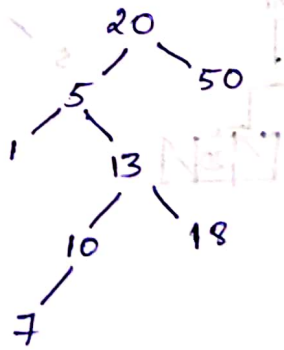


BST

* It is an ordered binary tree where all the values are unique & distinct. Always, the left child will have values < root node, right child will have values > root node.

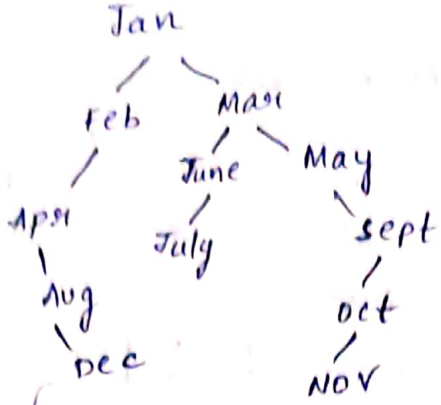
* construct a BST with the values 20, 5, 13, 50, 10, 1, 7, 18.

Sol =



* construct a BST with Jan, Feb, ..., Dec.

Sol:

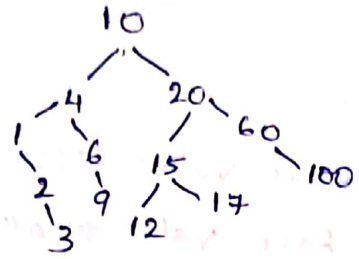


3-9-19

* construct a BST with

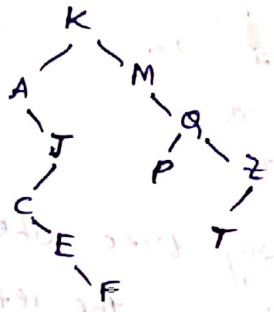
(i) 10, 4, 20, 15, 6, 1, 2, 17, 60, 12, 3, 9, 100.

Sol



(ii) K, A, J, M, C, Q, Z, T, P, E, F

Sol



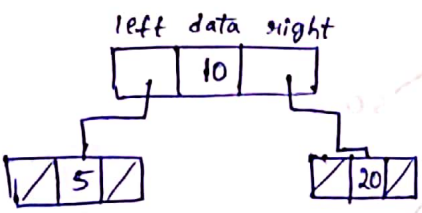
* we have 3 methods for tree traversals

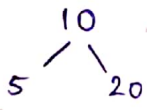
1) Pre order — Root, left, right
2) In order — Left, Root, right
3) Post order — Left, right, root

Ex:-

1) Pre order

10, 5, 20





2) In order

5, 10, 20

3) Post order → 5, 20, 10.

* For (i) problem,

Pre order → 10 4 1 2 3 6 9 20 15 12 17
60 100

In order → 1 2 3 4 6 9 10 12 15 17
20 60 100.

Post order → 3 2 1 9 6 4 12 17 15 100
60 20 10

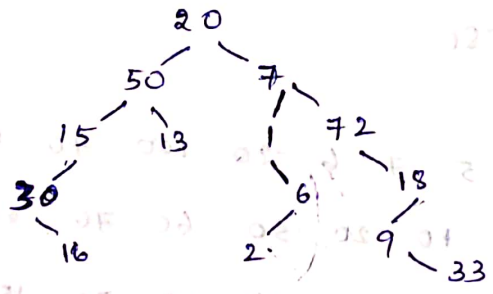
* To get values in ascending order, we use in order traversal.
* To get values in Descending order, we use right, root, light.
* For (ii) Problem,

Pre order → K A J C E F M Q P Z T

In order → A C E F J K M P Q T Z

Post order → F E C J A P T Z Q M K

*Do all the traversals for the following binary tree



Sol) Pre order → 20 50 15 30 16 13 7 1 6 2
72 18 9 33

In order → 30 16 15 50 13 20 1 2 6 7
72 9 33 18

Post order → 16 30 15 13 50 2 6 1 33 9
18 72 7 20

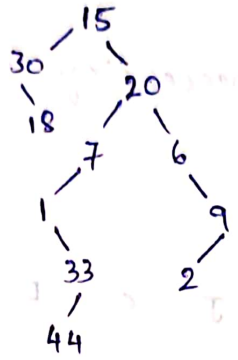
* To construct a binary tree from traversals, then we need either pre order - In order (or) In order - post order combinations.

* If Pre order, In order combination is given, then mark the root word in "In-order" and search for the elements in pre-order.

* If Post order, In order combination is given, then mark the root word in "In-order" & search for the elements in reverse order in Post-order.

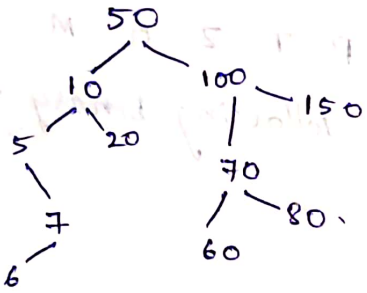
* Post order: 18, 30, 44, 33, 1, 7, 2, 9, 6, 20, 15

In order: 30, 15, 15, 1, 44, 33, 7, 20, 6, 2, 9.



04-09-19

*

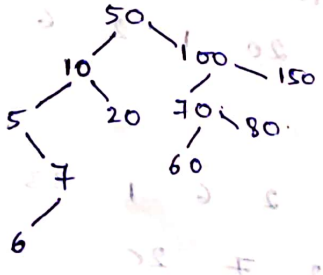


Pre order: 50 10 5 7 6 20 100 70 60 80 150

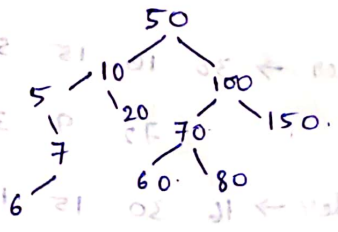
In order: 5 6 7 10 20 |50| 60 70 80 100 150.

Post order: 6 7 5 20 10 60 80 70 150 100 50.

Pre-order & In-order



Post-order & In-order



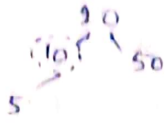
Deletion:-

case 1 : only root node

case 2 : leaves

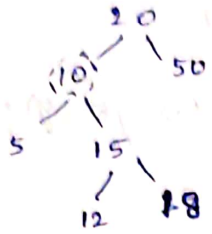
case 3 : one child node

case 4 : Node with 2 children



↓

Here, replace with inorder predecessor (10) inorder successor.



Inorder predecessor → 1 step left, extreme right

Inorder successor → 1 step right, extreme left.

* These two are always leaves / have 1 child.

Inorder predecessor :-

In the inorder traversal, the node that comes before that.

To find, 1 step left, then extreme right.

Inorder successor :-

In the inorder successor, the node that comes after that.

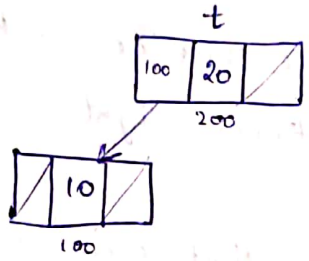
To find, 1 step right, then extreme left.

```
* class Tnode
  {
    int data;
    Tnode left, right;
    Public Tnode(int x)
     { data = x;
       left = right = null;
     }
  }
  class BST          /* BST is an ADT */
  { Tnode root;
    public BST()
    {
      root = null;
    }
    Tnode Insert(node t, int y)
     { if (t == NULL)
```

```
{
    t = new Tnode(y);
    return t;
}
else
{
    if (y > t.element) data)
    {
        t.right = insert(t.right, y);
    }
    else
    {
        t.left = insert(t.left, y);
        return t;
    }
}
```

05-09-19

*



Here, t.left = 100, which is address of "10".

```
* void non_rec_insert(int y)
  {
    int LR = 0;
    Tnode temp = new Tnode(y);
    if (root == null)
        root = temp;
    else
    {
        Tnode p, t = root;
        while (t != NULL)
        { if (y > t.data)
          {
              p = t;
              LR = 1;
              t = t.right;
          }
          else
          {
              p = t; LR = 0; t = t.left;
          }
        }
```

```
    if (i e = = 1)
        p.right = temp;
    else
        p.left = temp;
  }
 }
```

Inorder Traversal:-

```
void inorder (Tnode t)
{
  if (t != null)
  {
    inorder (t.left);
    System.out.println (t.elem.data);
    inorder (t.right);
  }
}
```

Preorder Traversal:-

```
void preorder (Tnode t)
{ if (t != null)
  {
    System.out.println (t.data);
    preorder (t.left);
    preorder (t.right);
  }
}
```

Post order Traversal:-

```
void post order (Tnode t)
{
  if (t != null)
  {
    Postorder (t.left);
    postorder (t.right);
    System.out.println (t.data);
  }
}
```

Check if a node is external node:-

```
bool isexternal (node t)
 {
    if ((t.left == Null) && (t.right == Null))
      return 1;
    else
      return 0;
 }
```

06-09-19

```
* bool search (node t, int y)
  {
    if (t==null) return 0;
     else if (t.data == y) return 1;
     else
      {
         if (y>t.data)
             return (search (t.right,y));
         else
             return (search (t.left,y));
      }
  }
* void delete (int x)
  {
     node t = root, pt = root;
      if ((root.data == x) && (root.left==null)
                          && (root.right == null))
      {                                  // case 1.
         root = null;
      }
      else
      {  while ((t!=null) && (t.data != x))
          {
             if (x > t.data)
              {
                 pt=t; t=t.right;
              }
             else
              {  pt=t; t=t.left;
              }
          }
```

```
if (t == null)
    S.o.println("Element not found");
else if (t.left == null && t.right == null)
 {                                    // case 2
    if (x > pt.data)
        pt.right = null;
    else
        pt.left = null;
 }
```

case-3 if

```
else if (t.left != null && t.right == null)
 {
    if (t.data < pt.data)
        pt.left = t.left;
    else if (t.element/data > pt.data)
        pt.right = t.left;
 }
else if (t.left == null && t.right != null)
 {
    if (t.data < pt.data)
        pt.left = t.right;
    else
        pt.right = t.right;
 }
```

case 4

```
else if (t.left != null && t.right != null)
 {  node pt = t, t1;
    t1 = t.right;
    while (t1.left != null)
     { pt = t1; t1 = t1.left;
     }
        t.data = t1.data;
        if (pt == t)   pt.right = t1.right;
```

```
else
    pt.left = tl.right;
}
}
```

10-9-19.

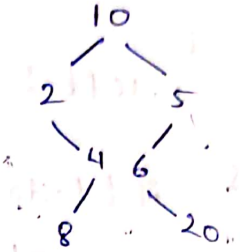
Problems:-

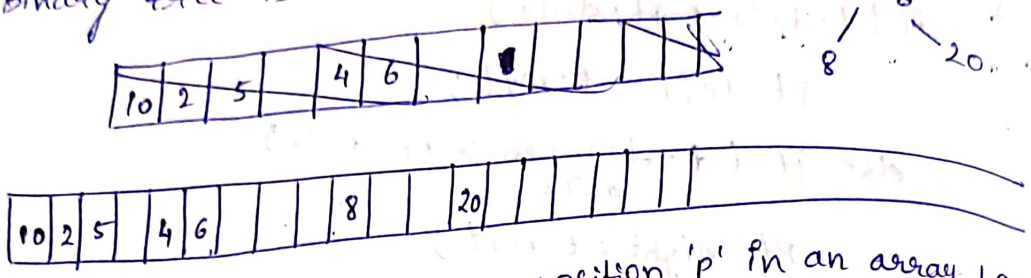
① Swap the sibling in a binary tree.

② Find the parent of a node.

③ Find the number of internal nodes & external nodes.

* Array representation of adjacent binary tree is





* If the parent is at the position 'p' in an array based implementation of binary tree, then

left child is at position → 2p+1 and

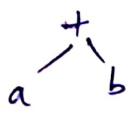
right child is at position → 2p+2.

* similarly, if 'l' is left position, then parent node is $\left(\frac{l-1}{2}\right)$ & left child.

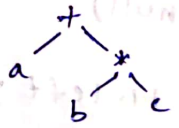
* If the right child is at position 'r', then parent node is at the position $\left(\frac{r-1}{2}\right)$.

Expression Tree:-

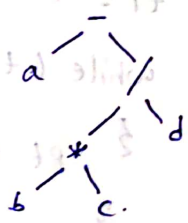
* a+b



* a+b*c

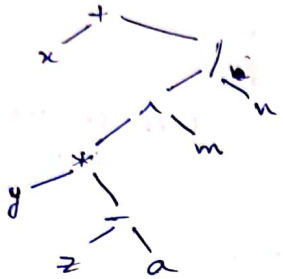


* a-b*c/d

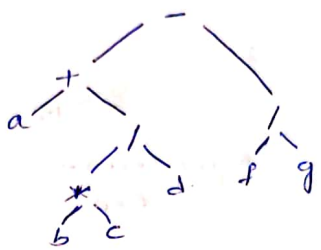


* a + (b - c * d) + (k + f) / m

* x + (y * (z - a)) ^ m / n



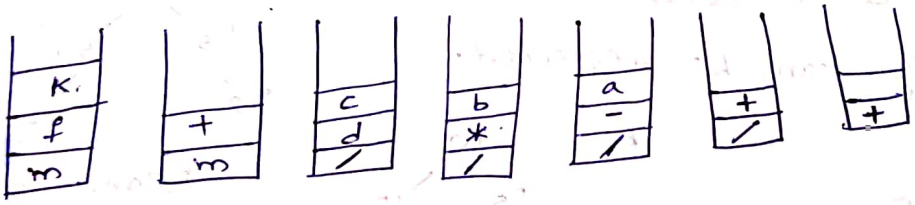
* a + b * c / d - f / g

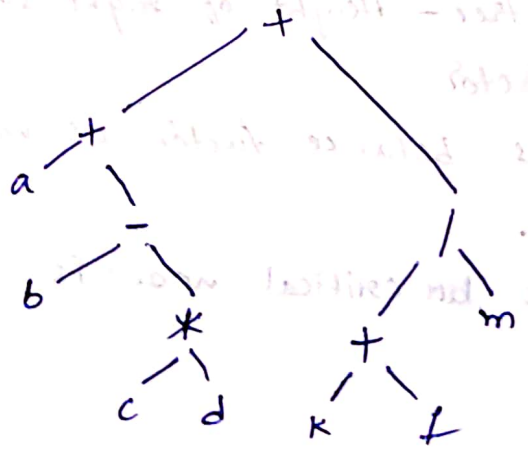


* prefix expression of a + (b - c * d) + (k + f) / m is

a + (- b * c d) + / + k f m

+ + a - b * c d / + k f m

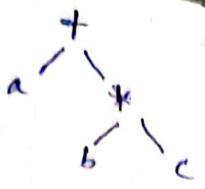




* a+b*c

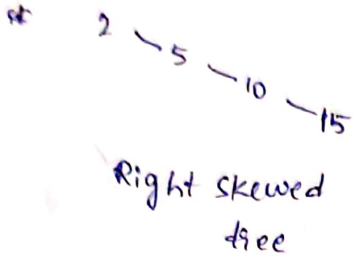
+a*bc

abc*+

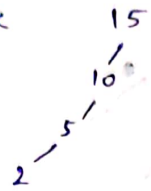


* In the worst case, BST can be constructed as right skewed tree (or) left skewed tree. So, it will have same complexity like a linked list.

* Search - O(n)



Right skewed tree



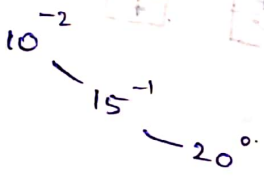
Left skewed tree.

Balanced BST (or) AVL Tree:-

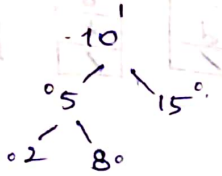
* Balance factor:- The difference between no. of levels of left tree and No. of levels on right side.

* If balance factor for every node is either 0 or 1, then it is a Balanced BST.

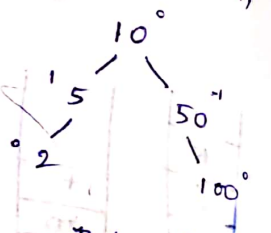
* -ve sign can be neglected.



Not a balanced BST.
Critical node is 10.



Balanced BST



Balanced BST

* Height of Left subtree - Height of right subtree = Balance factor.

* The node that has balance factor as more than 1 is called critical node.

* The BST which has a critical node is called unbalanced BST.

* ① → construct an AVL tree, 10, 100, 20, 2, 7, 9.

* To construct an AVL tree, we need 4 rotations.

1) RR rotation → only one rotation
2) LL Rotation → only one rotation
3) RL Rotation → 2 rotations
4) LR Rotation → 2 rotations

RR rotation:- The rotation will be to the left side.

LL Rotation:- Since the left of to left side is heavy, we are rotating to the right side.

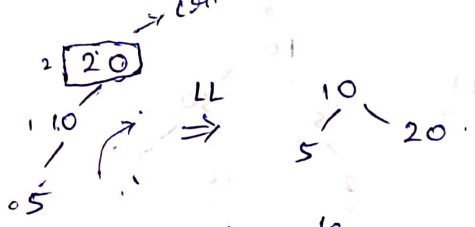
RL Rotation:- left of right side is heavy. so, 1st we are rotating to the right side, then to the left side.

LR rotation:- Right of left side is heavy. so, 1st we are rotating to the left side, then to the right side.

12-9-19

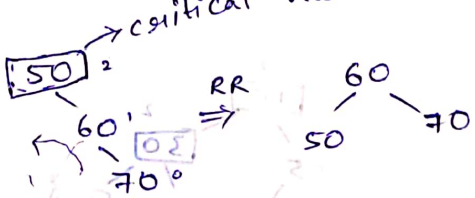
* LL rotation:-

20, 10, 5



* RR rotation:

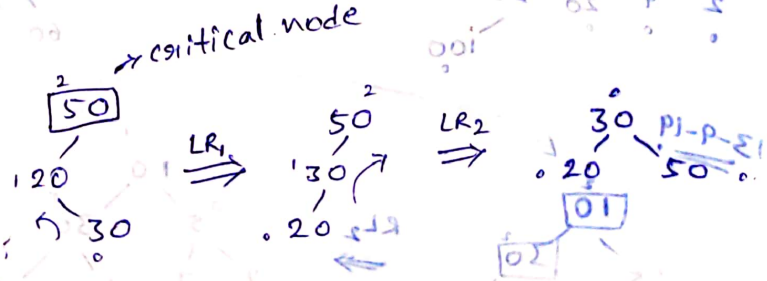
50, 60, 70



Here, right of right is heavy.

* LR rotation:-

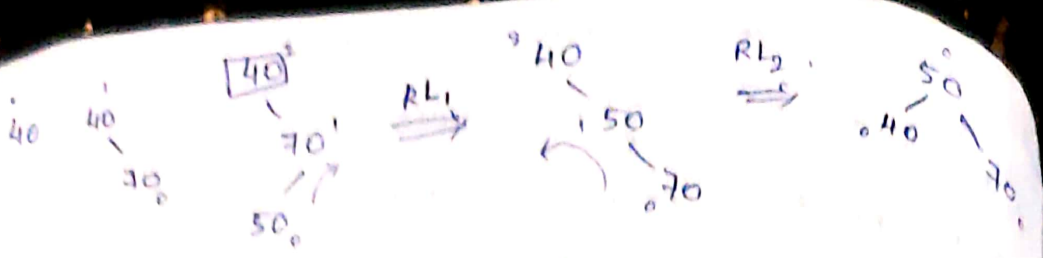
50, 20, 30



Here, 50 became a critical node bcz of the insertion of 30.

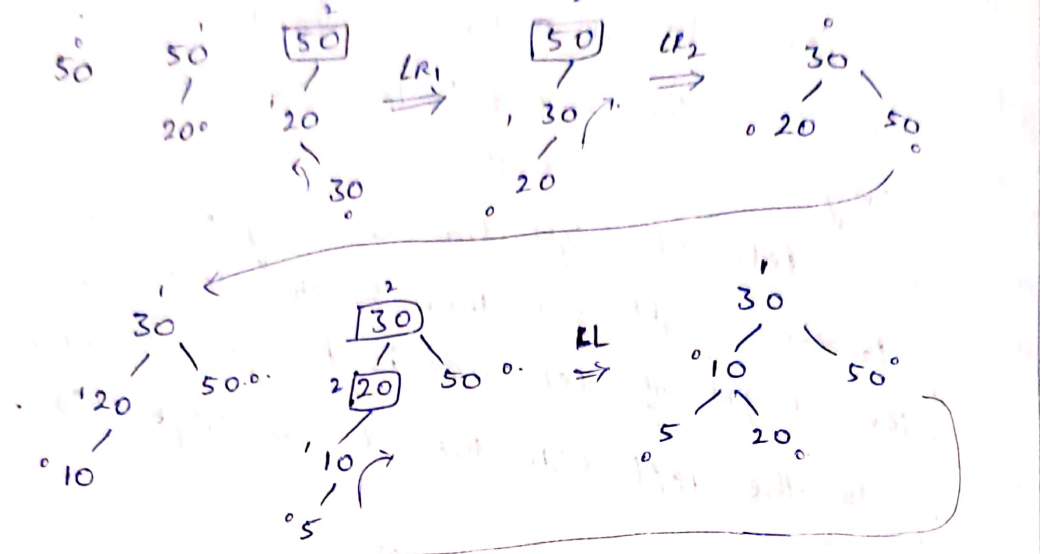
* RL rotation:-

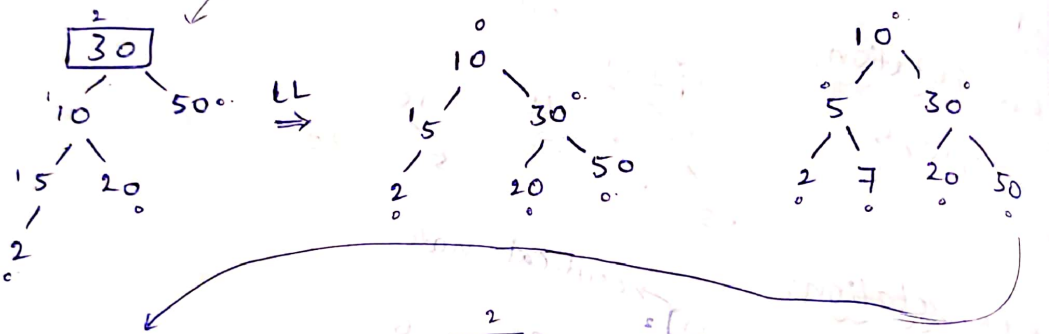
40, 70, 50

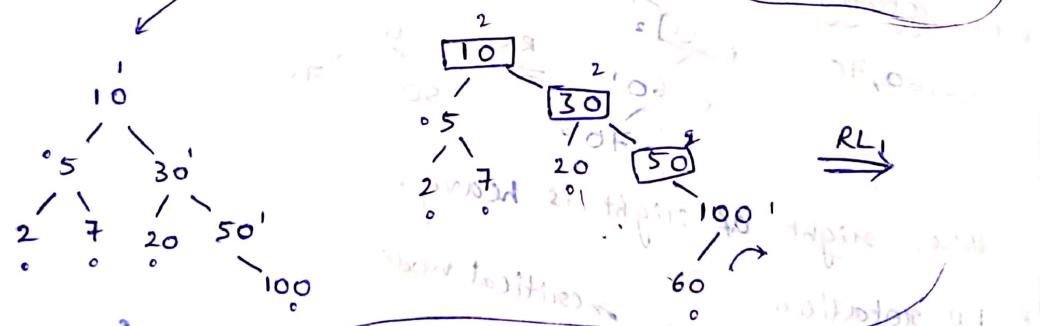


* construct an AVL tree

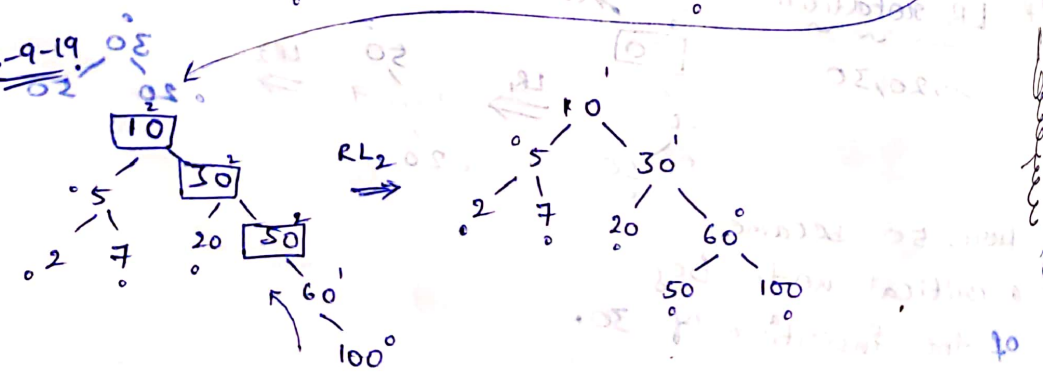
50, 20, 30, 10, 5, 2, 7, 100, 60, 19

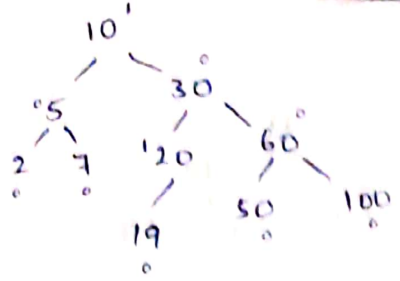






13-9-19

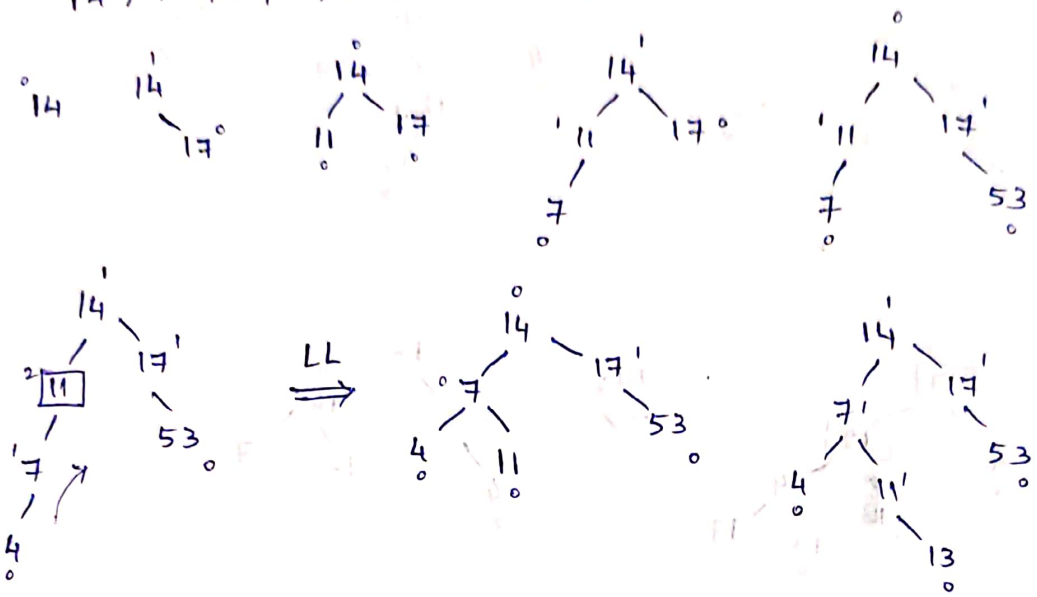


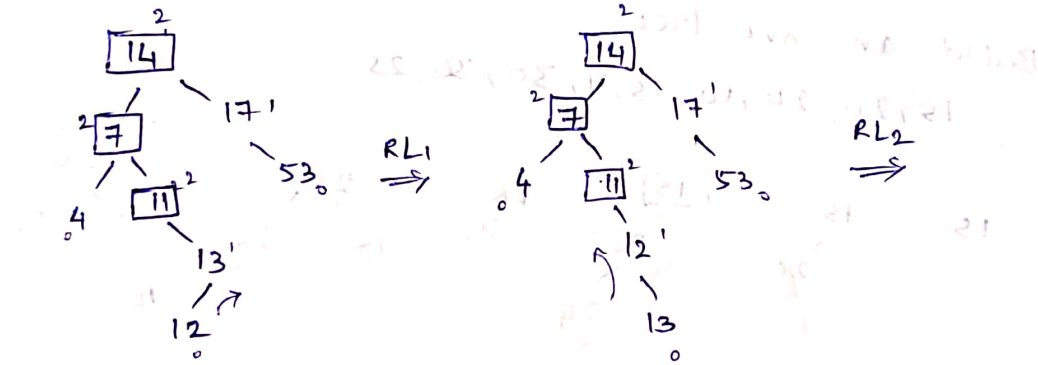


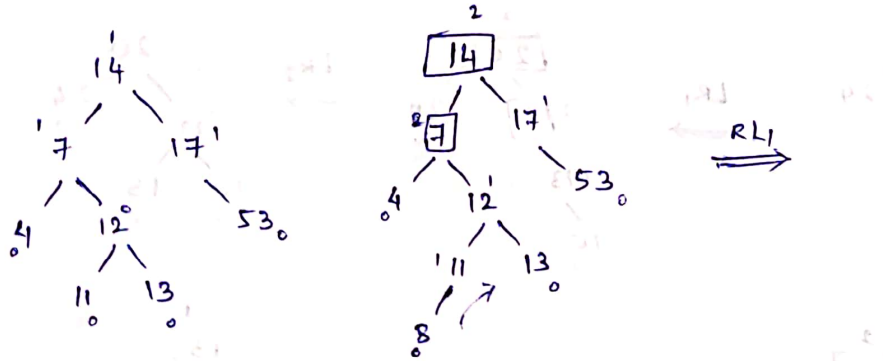
* Adilson Velskin Landis tree (AVL tree)

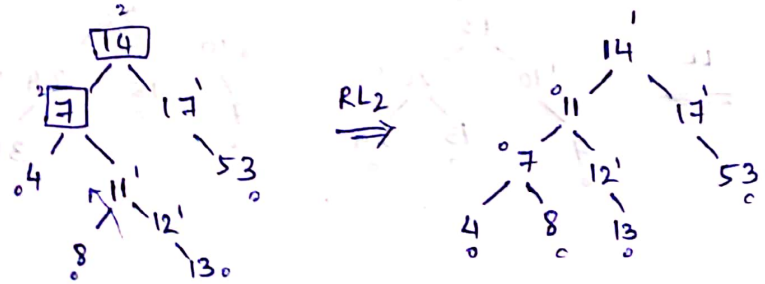
* construct an AVL tree

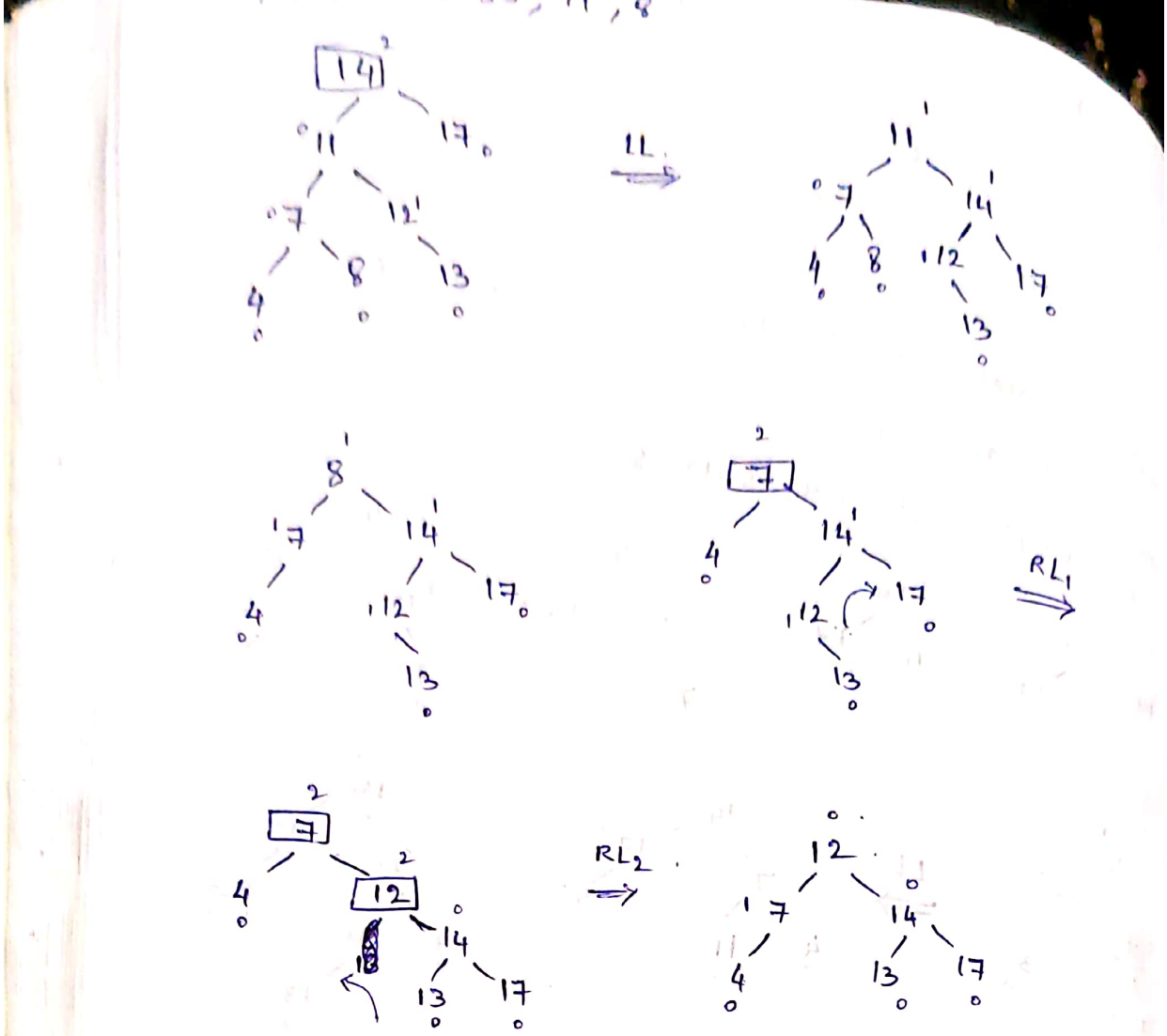
14, 17, 11, 7, 53, 4, 13, 12, 8







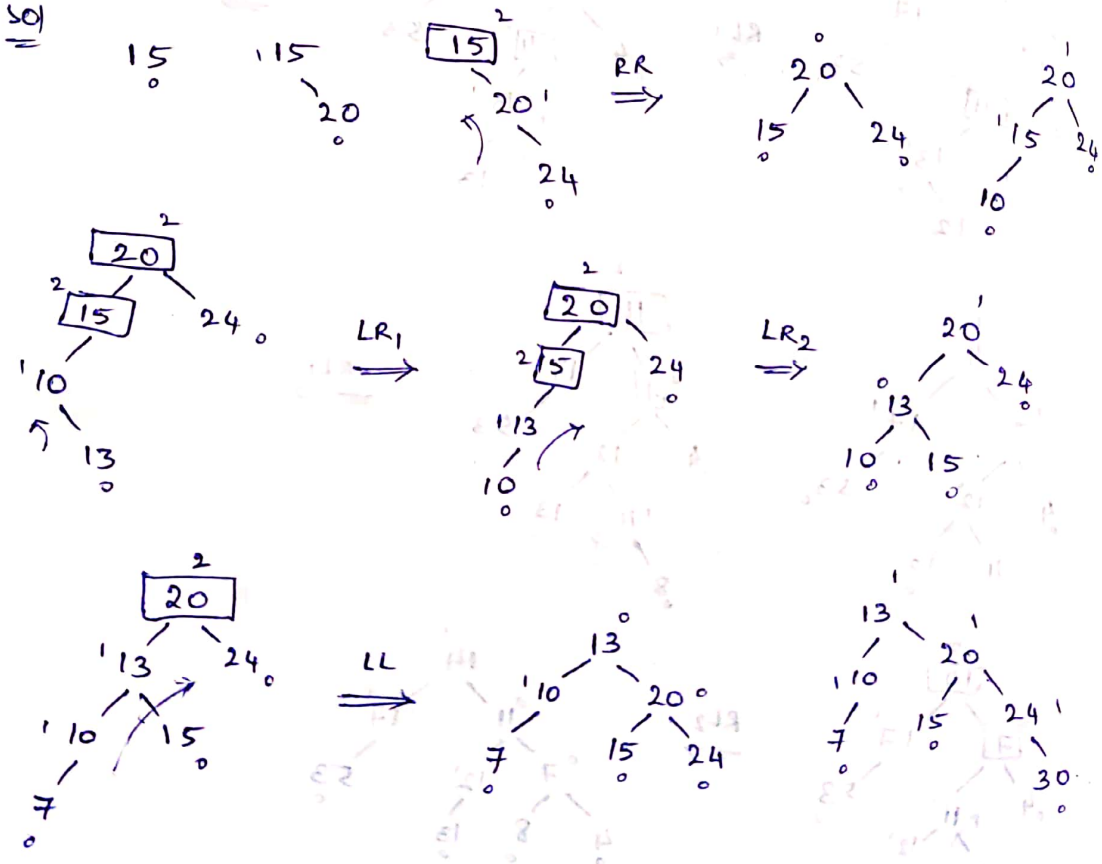


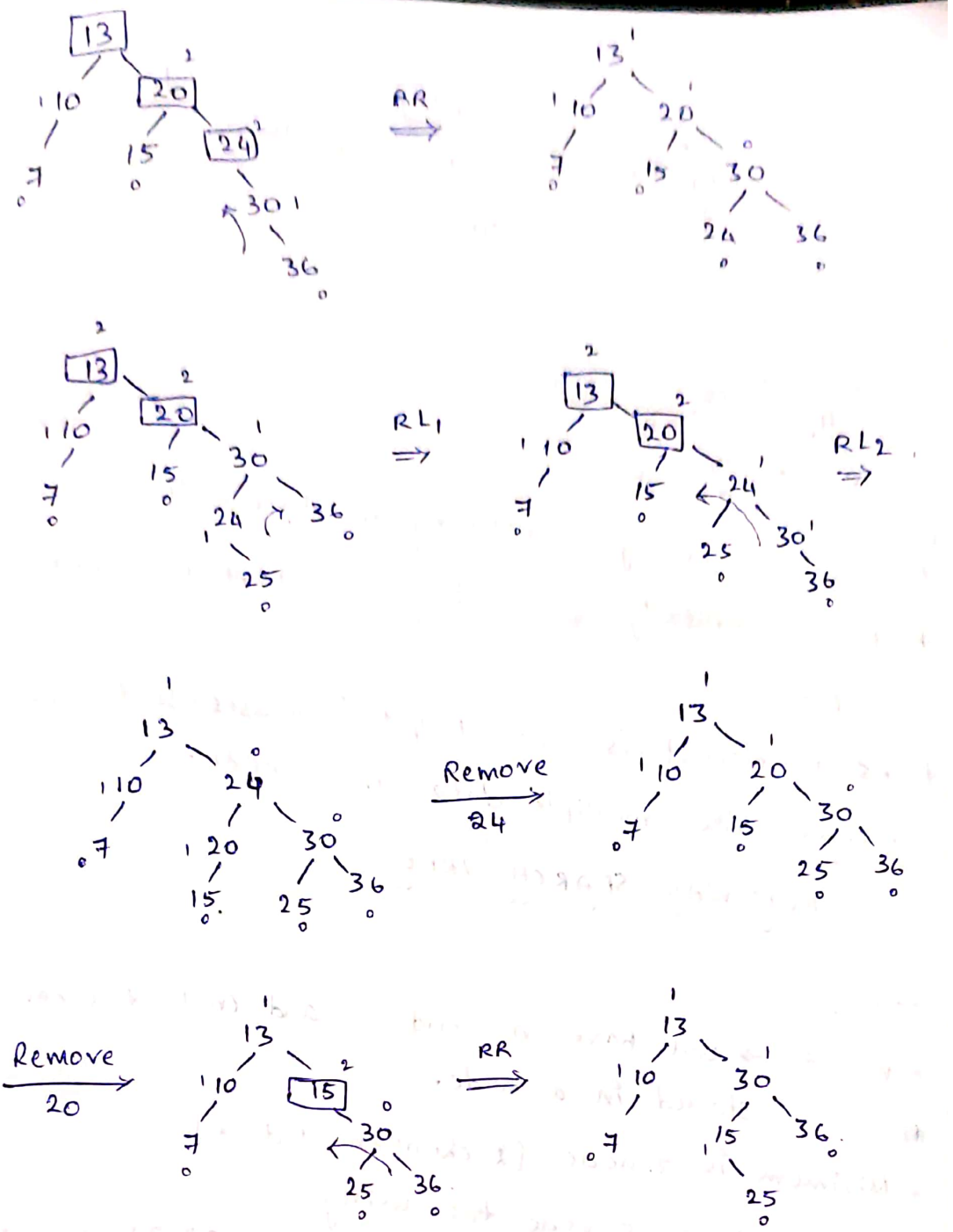


* Build an AVL tree

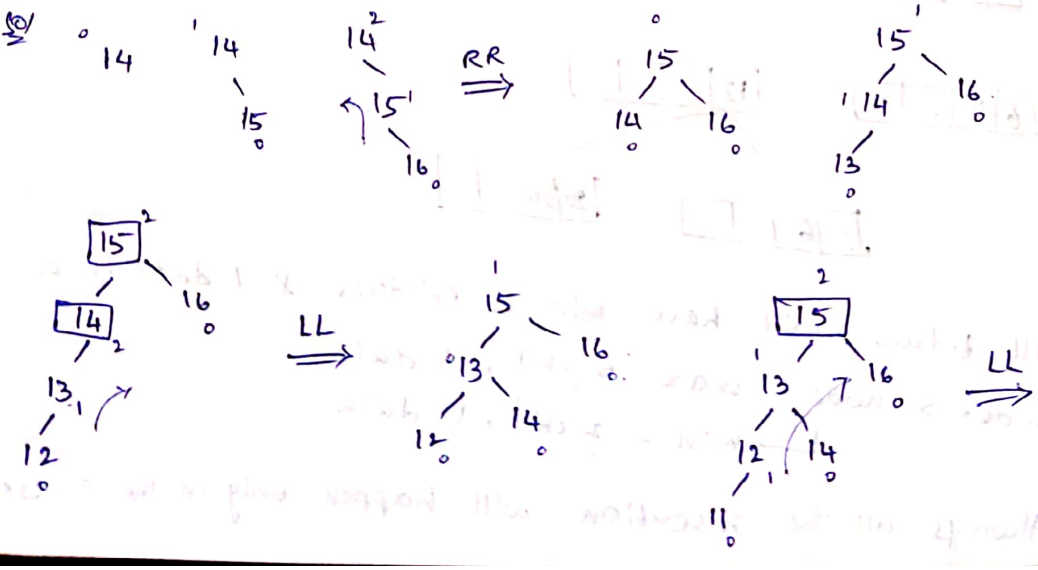
15, 20, 24, 10, 13, 7, 30, 36, 25

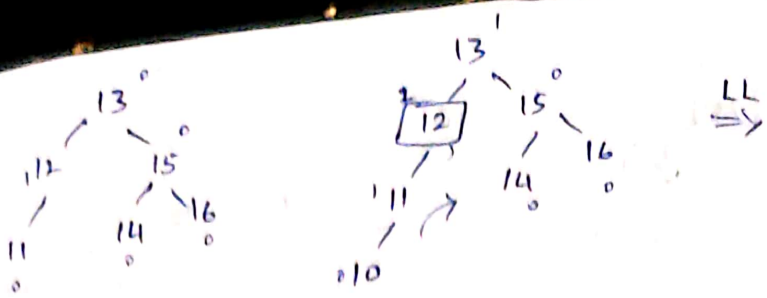
Sol

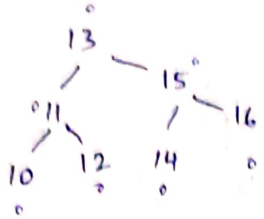




* → Insert 14, 15, 16, 13, 12, 11, 10 in an empty AVL tree.







17-9-19

* Time complexity of inserting an element is $O(n)$.
* Time complexity of searching an element is $O(\log_2 n)$ in

AVL tree

* As an element is added, height increases. So, to improve this, we store multiple data in a node.

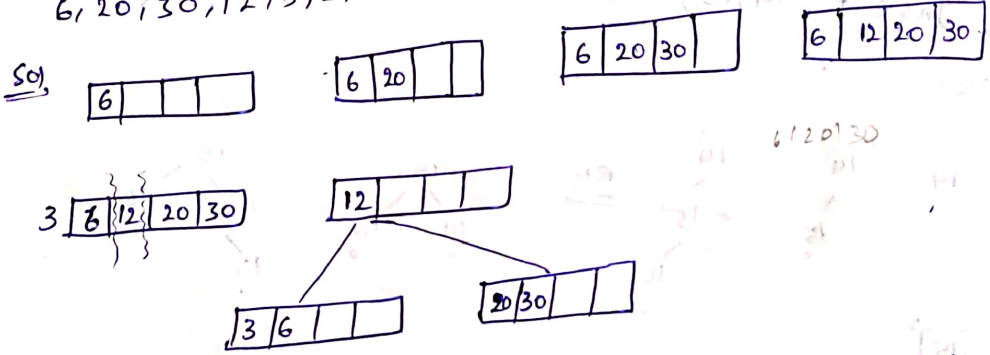
MULTIWAY SEARCH TREE

B-Tree:-

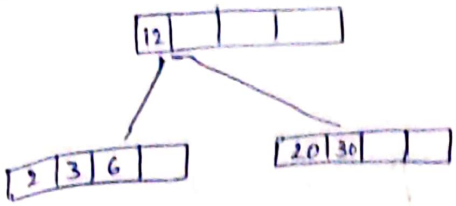
* n-node → will have n children and (n-1) data can be stored in a node.
* Minimum is 2-node [2 children, 1 data]
* construct a 5-node tree using

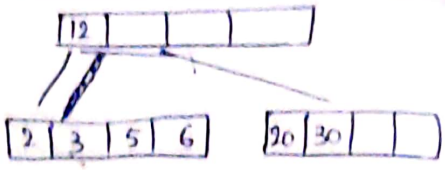
6, 20, 30, 12, 3, 2, 5, 4, 11, 10, 8, 7, 21, 28, 23, 25, 34, 33, 31, 35

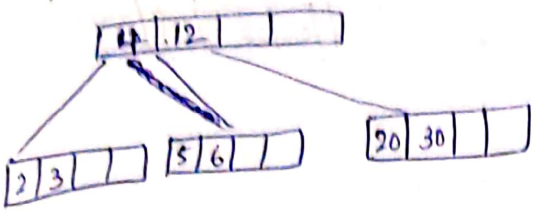
So,

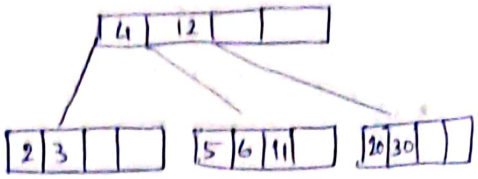


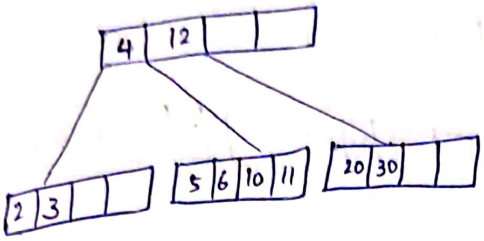
* All B-tree's will have min 2 children & 1 data in a node. 5 node → max - 5 chil, 4 data
  → min - 2 chil, 1 data
* Always all the insertion will happen only on the leaves.

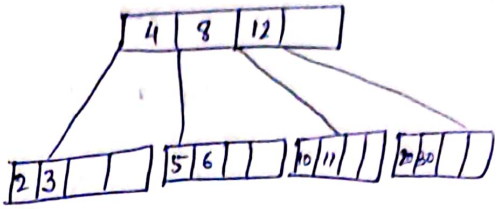


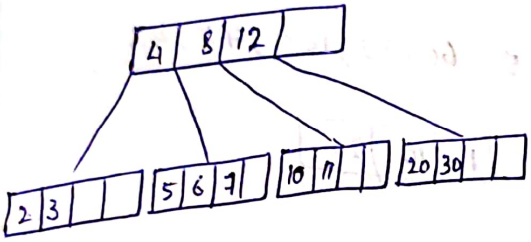


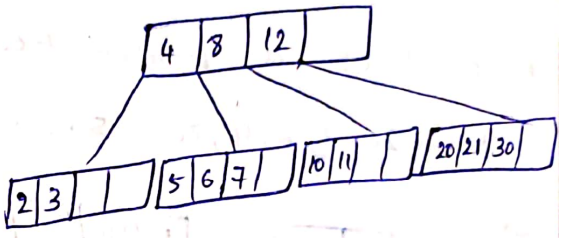


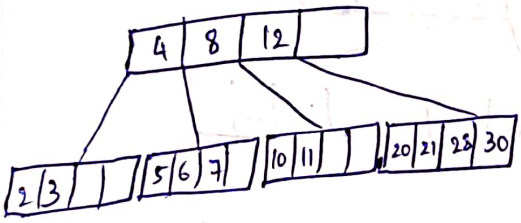


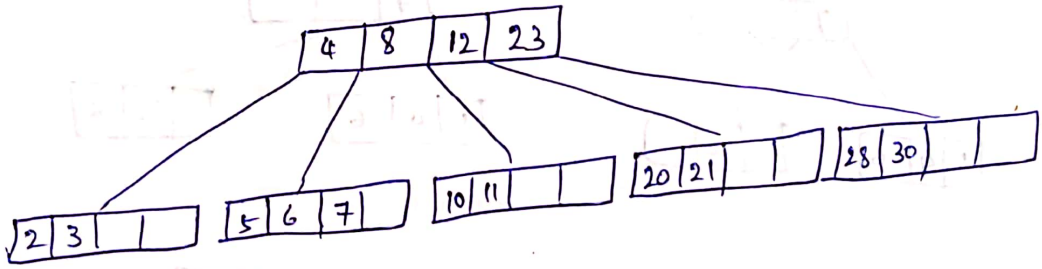


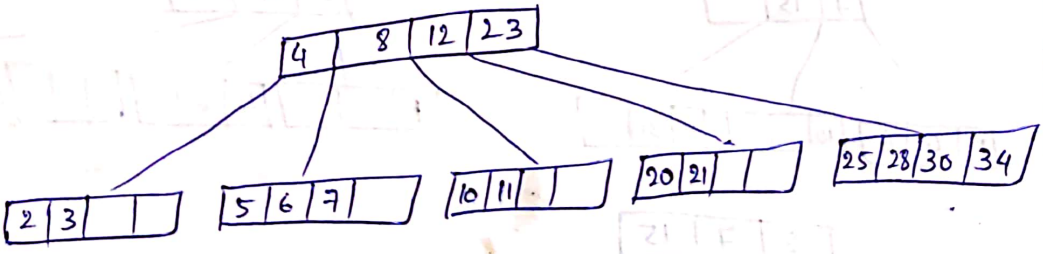




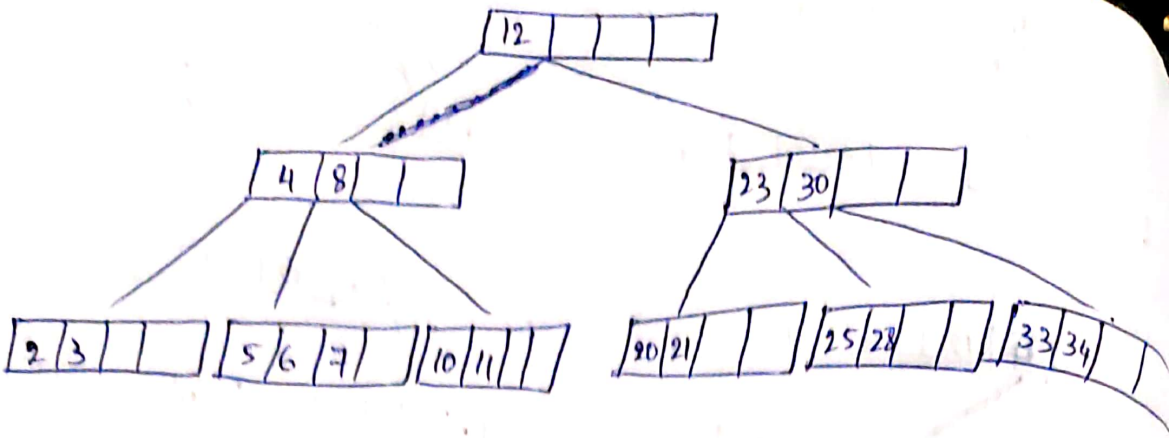


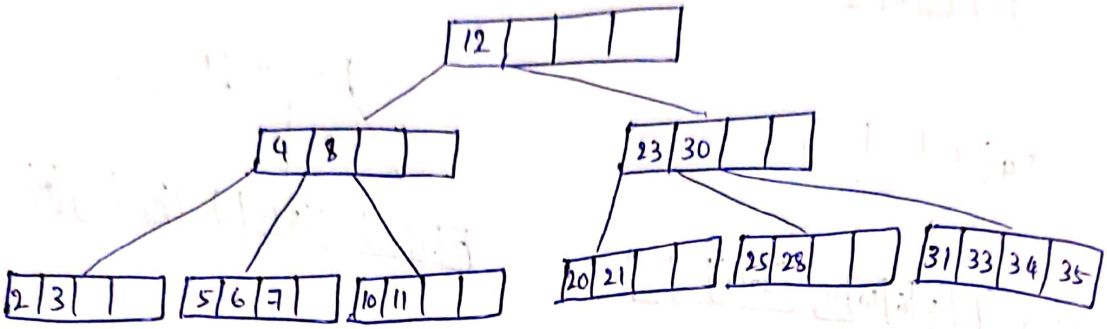






* Always, all the ~~leaves~~ leaves will be at same level in B-tree.

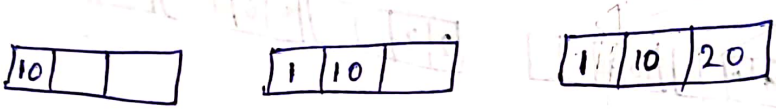


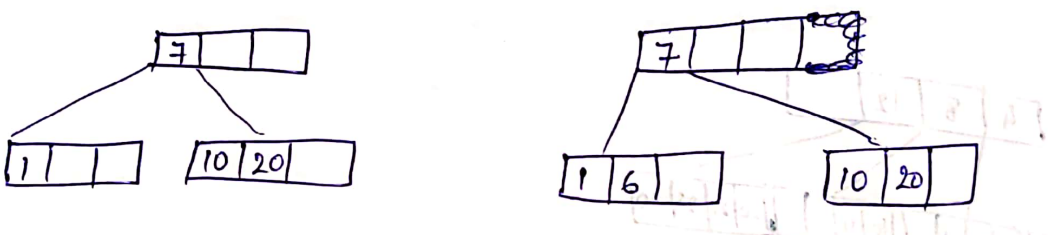


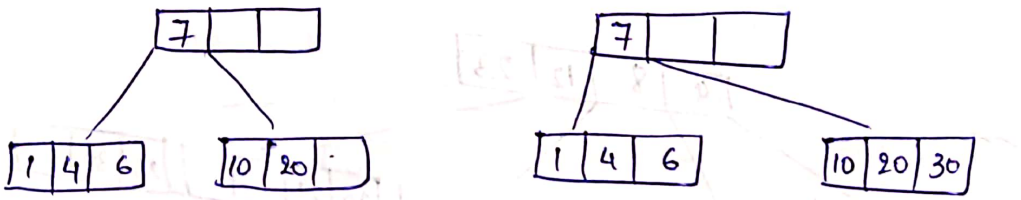
* Construct a 4-node tree with the values.

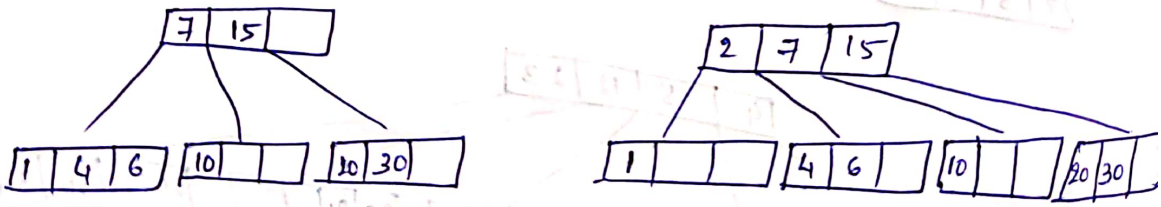
10, 1, 20, 7, 6, 4, 30, 15, 2, 3, 5, 60, 52, 100, 75, 43.

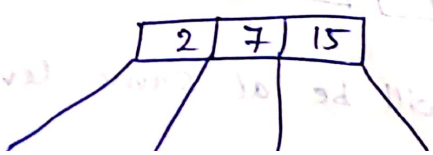
Sol.

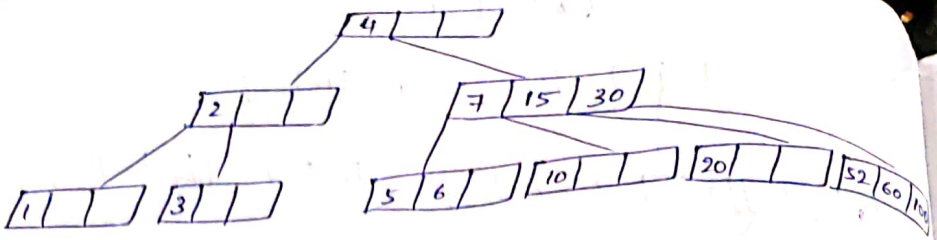


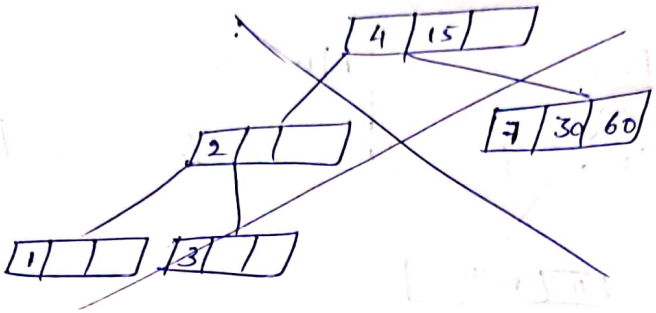


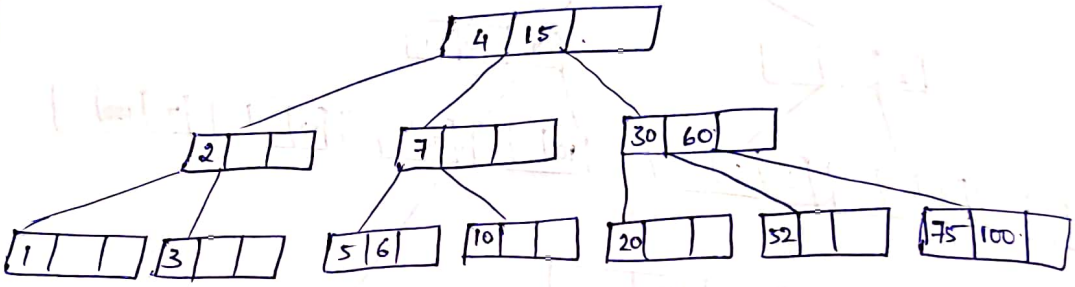


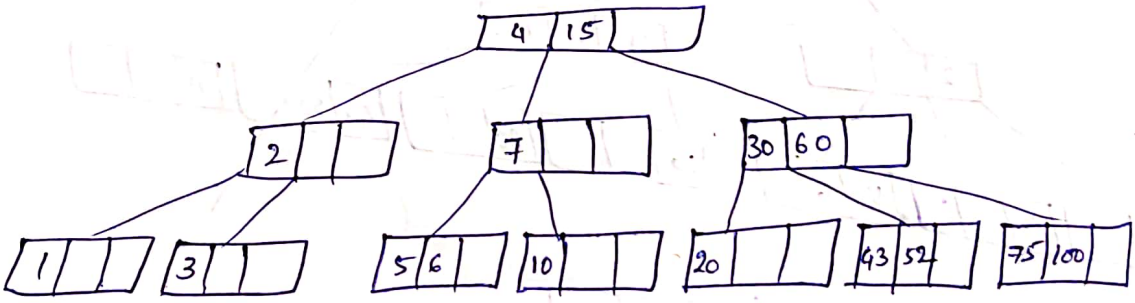












18-9-19

```
class node
{
    int x[4];
    node next[5];
}
```

Deleting an element from B-tree:-

* Deleting an element from leaf node:

If the element is in the leaf node, check for the min. condition, if there is no exception, simply delete it, else if underflow occurs [Ex:- [5 | |]. If we delete 5, underflow occurs]

if there is enough items in the siblings, go for transfer function or redistribution

If there is no enough items in the sibling, go for fusion (or) merge.

* If you are deleting an internal node, swap with inorder predecessor.

Now the ~~do~~ node is a leaf node. So, follow the steps for deleting element from leaf node.

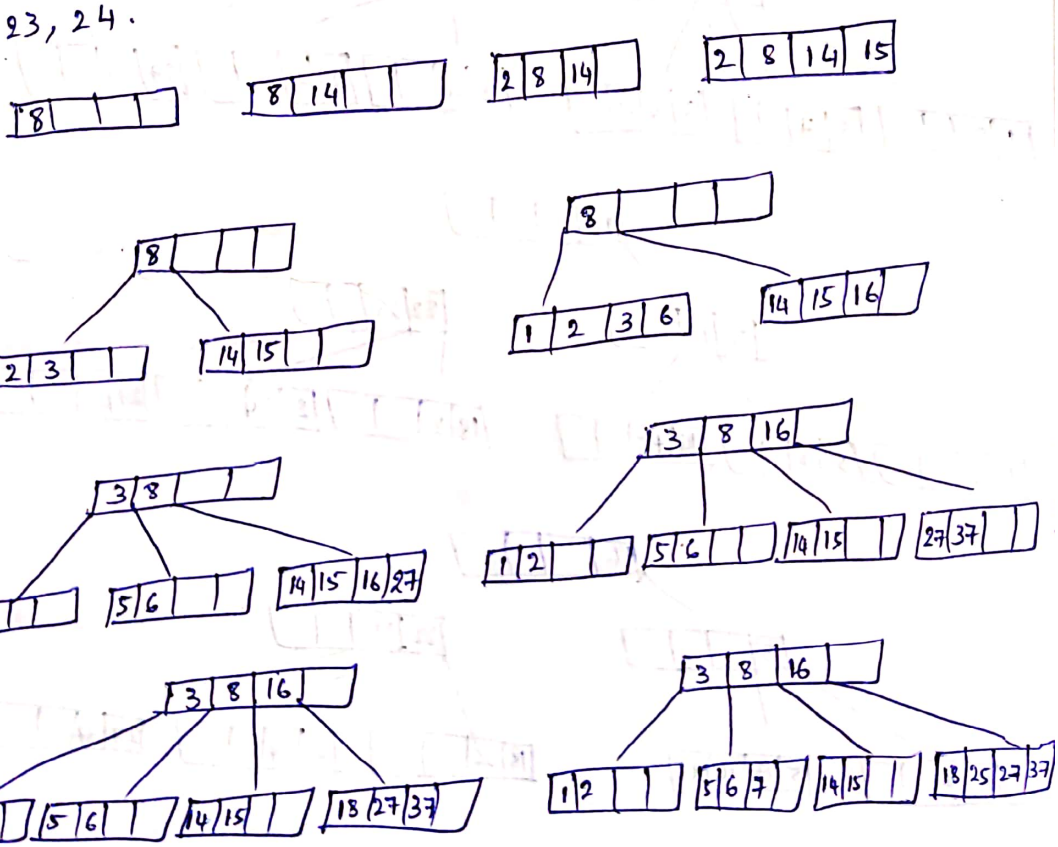
checking underflow:-

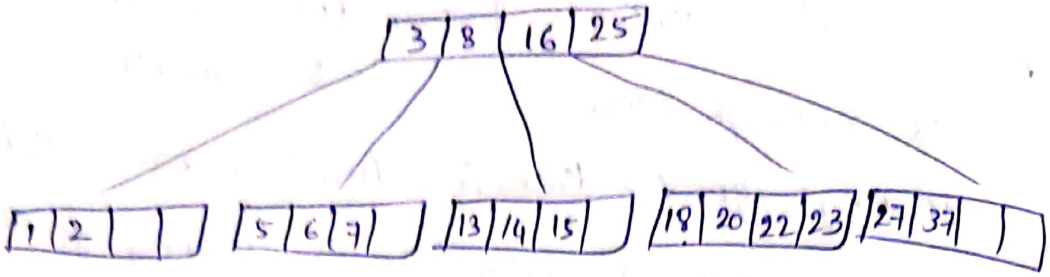
For n-node, minimum $\lceil \frac{n}{2} \rceil - 1$.

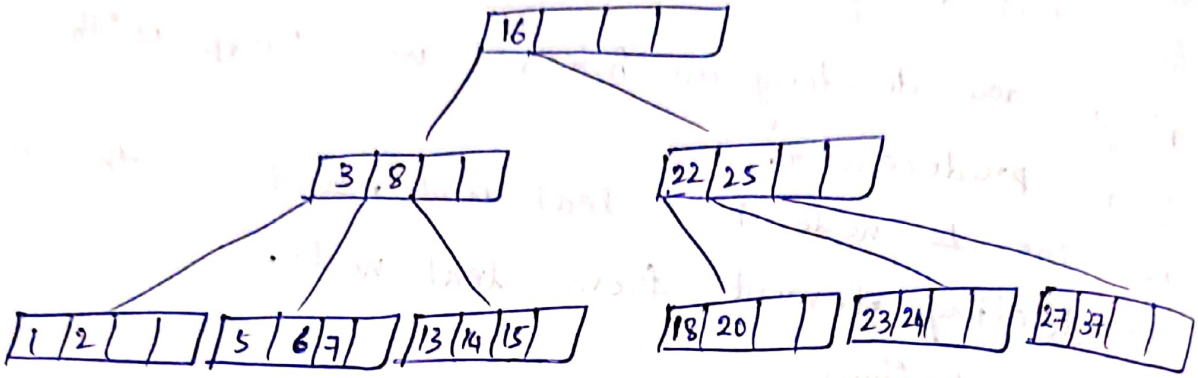
* construct a B-tree (5-node) [order-5]

8, 14, 12, 15, 3, 1, 16, 6, 5, 27, 37, 18, 25, 7, 13, 20, 22, 23, 24.

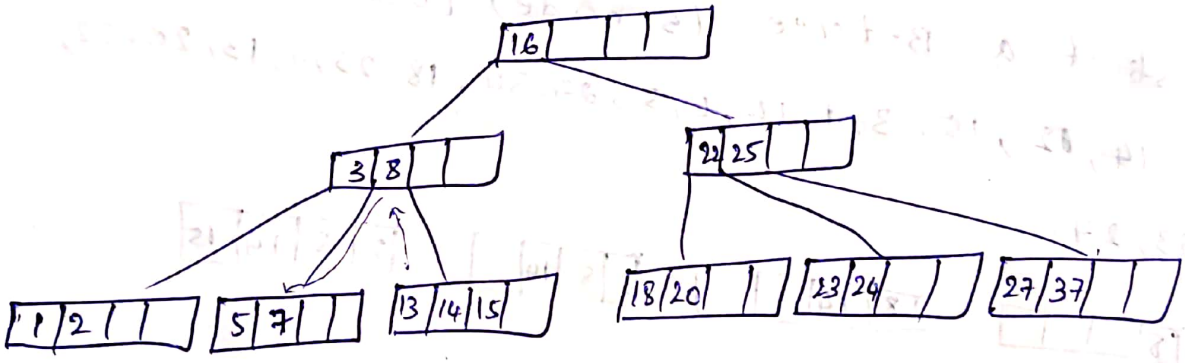
Sol.

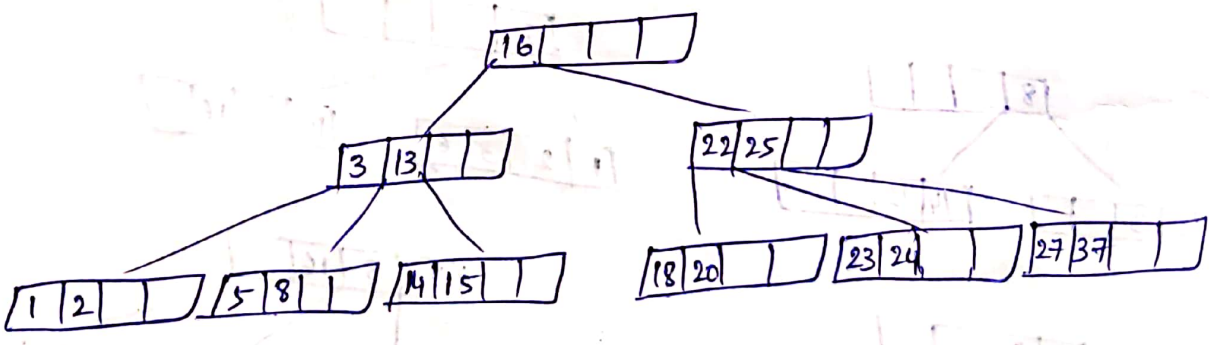


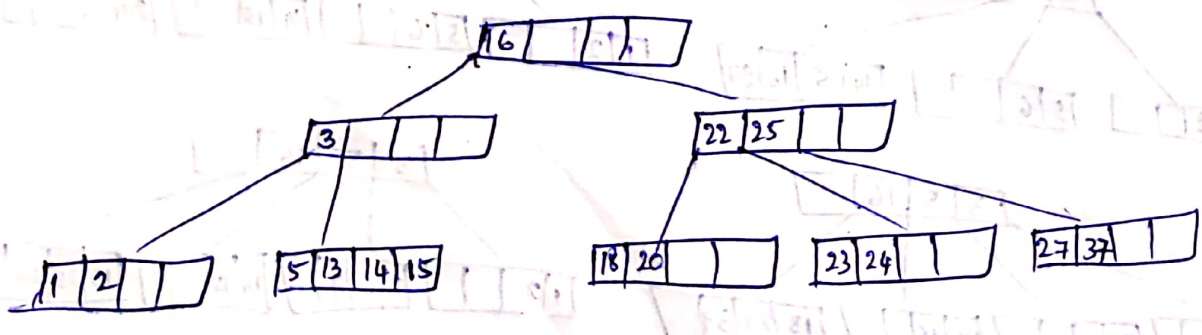


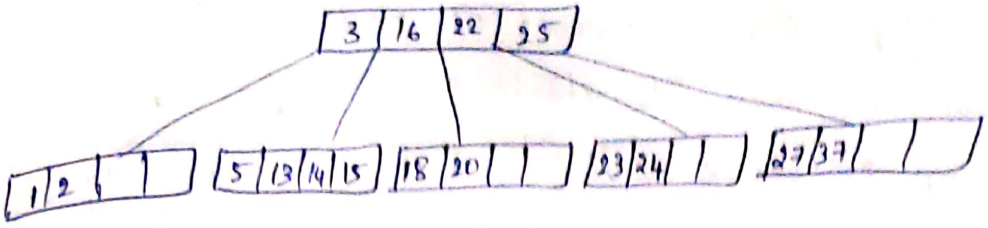


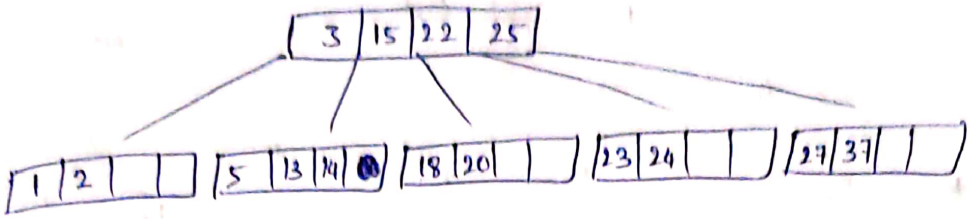
Delete 6, 7, 8, 16



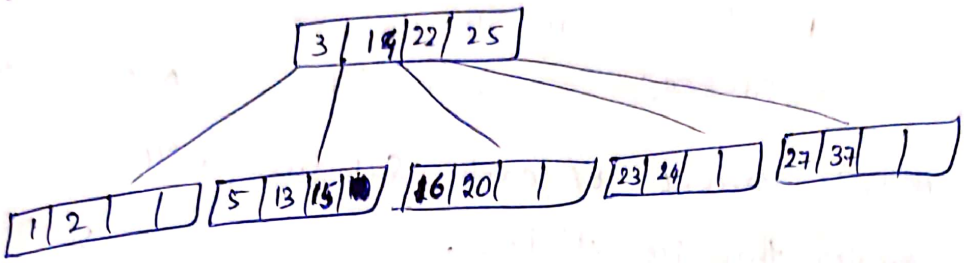








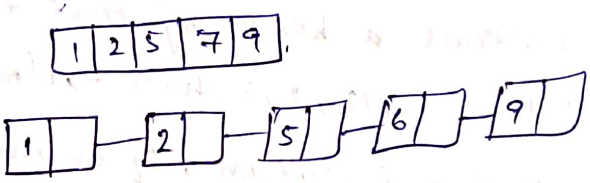
Delete 18.



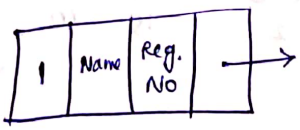
19-9-19

Priority Queue implementation:-

* key, value
* 1) Sorted sequence



* In the worst case, time complexity for insertion is O(n).
* Time complexity for deletion is O(1)
* 2) unsorted sequence.



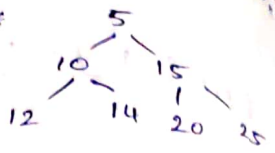
* Time complexity for insertion is O(1).
* For deletion, search for the high priority and delete. So, Time complexity is O(n).
* But as a whole, both sorted & unsorted sequence have a time-complexity of O(n).

Heap:-

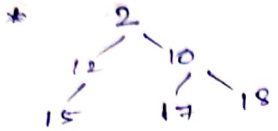
* Heap has less time complexity than sorted & unsorted sequence.

Min Heap:- Always parent of each sub-tree will have value less than its children.

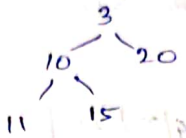
* Heap is not a BST.
* In Heap, always elements are filled from left to right and from top to bottom.



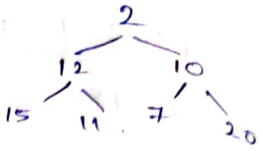
Heap



Not a valid heap



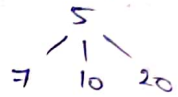
valid Heap



Not a Heap

Max Heap:- Parent of each sub-tree will have value greater than its children.

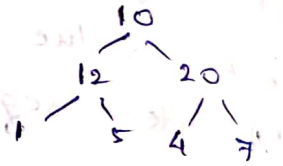
* Heap will have maximum 2 children
* Heap is a Binary tree, but not a BST.



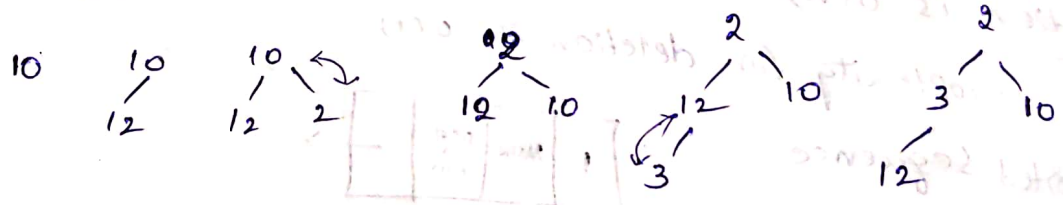
Invalid Heap

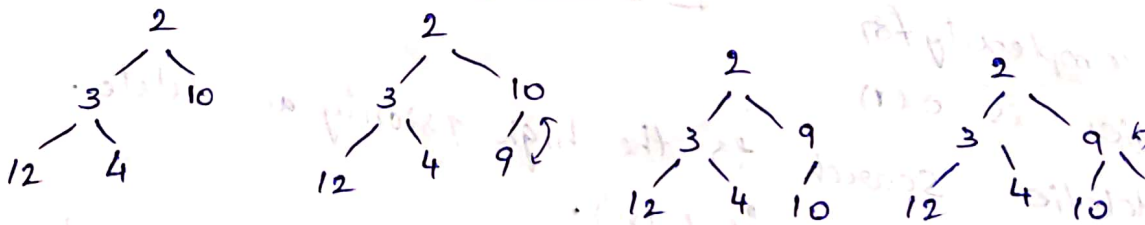
* construct a heap by inserting the values 10, 12, 2, 3, 4, 9, 6, 1 (minheap).

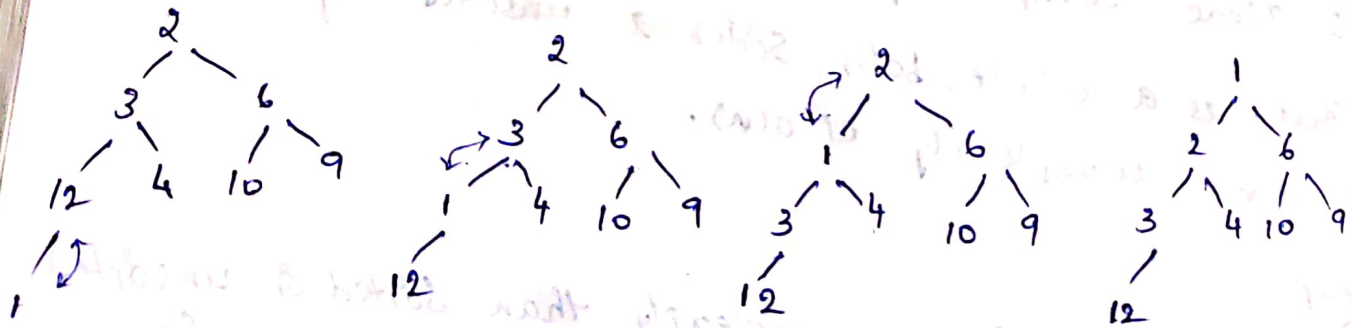
So during insertion, if an invalid condition occurs, we swap them, known as "upheap."



Valid Max Heap







* compared to arrays, linked lists can consume memory when we are doing trees. Because arrays (i.e; implementing trees using array) have unfilled fields.

* But, to construct a heap, we make use of arrays, since elements are filled from left to right & top to bottom, there is no change of unfilled fields.

* In worst case, time complexity to insert (it will travel entire tree's height) one element is $O(\log n)$.

* Time complexity to insert n elements is $O(n \log n)$.

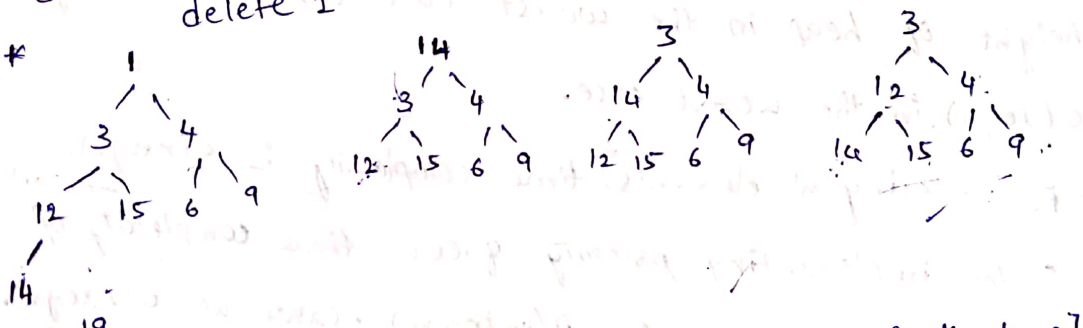
* Since it performs upheap operation till the root of the heap, insertion of an element depends on the height of the heap. Since the height of the heap is $O(\log n)$, in the worst case, insertion of an element will also take the time complexity as $O(\log n)$.

* For inserting 'n' elements in a heap, the time complexity is $n \log n$.

* Downheap is the process of swapping at the time of deletion.

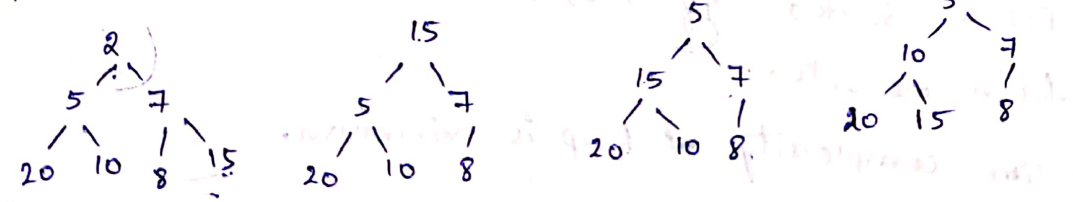
delete 1'

*



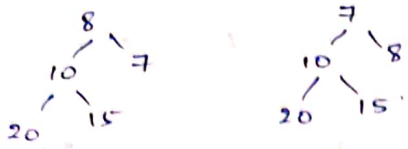
20-9-19

* To delete '2'. [replace with last element in the tree]

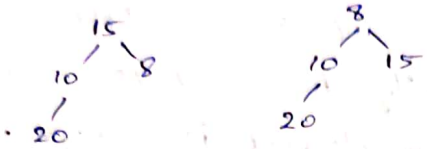


[swap with the smallest child].

* To delete 's'



Now, we delete 7.



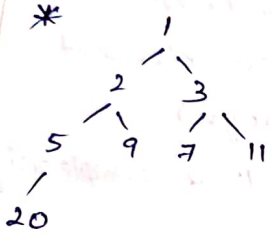
Now, we delete 8

Now, we delete 10.

Now, we delete 15.

20

*



Array representation of the heap is

1	2	3	5	9	7	11	20

* At the time of remove-minimum operation in a heap, the down-heap operation will go till the leaf node. So, the time-complexity of deletion will depend on the height of heap in the worst case. So, the complexity is O(logn) in the worst case.

* For deleting 'n' elements, time complexity is O(nlogn).

* So, for implementing priority queue (heap) - time complexity of insertion + deletion is O(2n log n), same as O(nlogn).

* For a sorted sequence, time complexity is $O(n^2)$ for 'n' elements.

* Time complexity of heap is minimum.

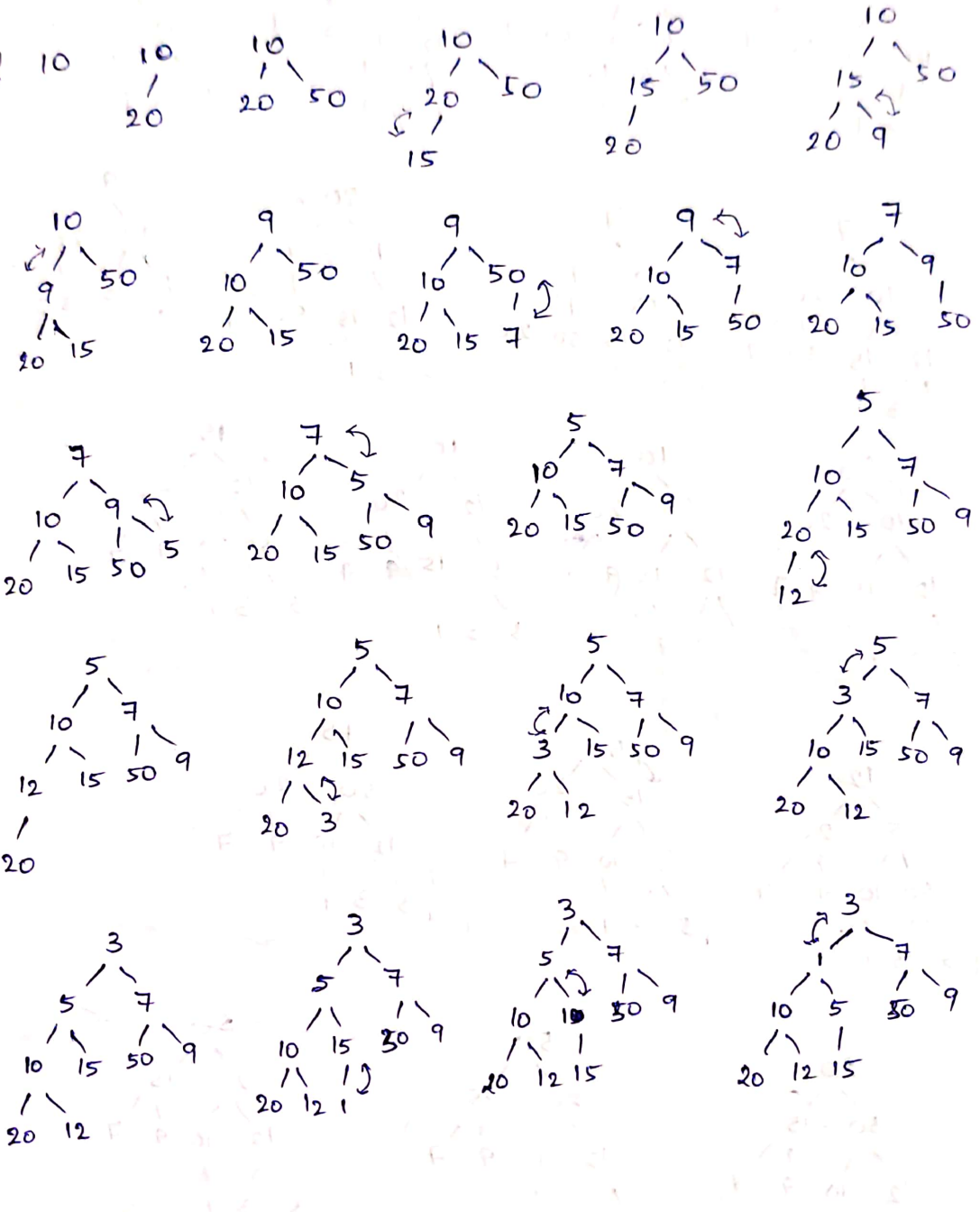
Priority Queue

Sorted Sequence	unsorted Sequence	Heap
Insert - O(n)	Insert - O(1)	Insert - O(log n)
Delete - O(1)	Delete - O(n)	Delete - O(log n)

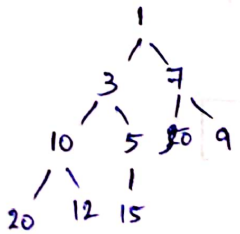
* construct and delete elements in the heap (minheap)

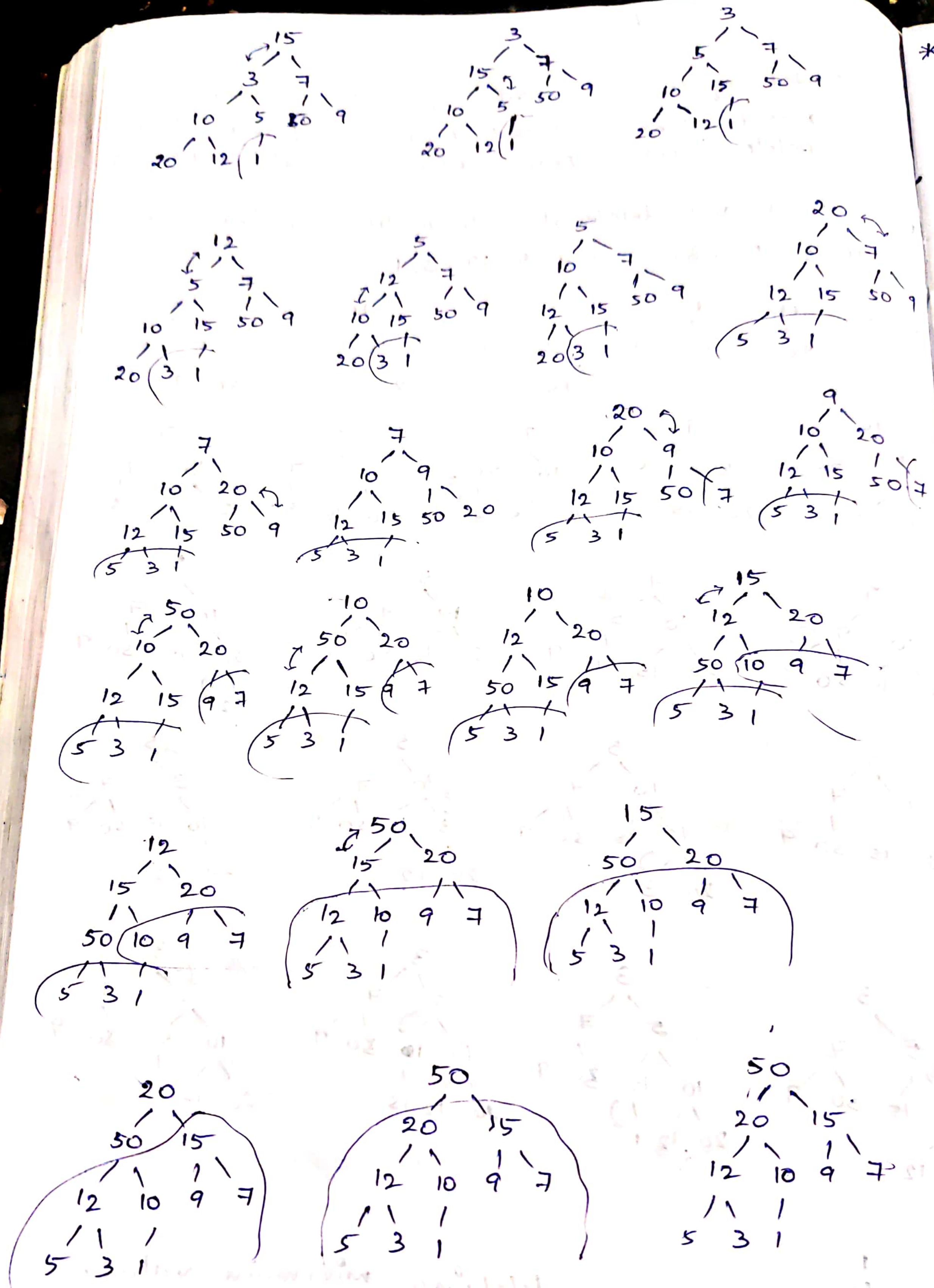
10, 20, 50, 15, 9, 7, 5, 12, 3, 1

Sol:



Now, deleting minimum value.





50	20	15	12	10	9	7	5	3	1

Descending order.

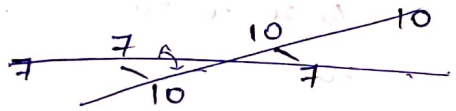
* To perform heap sort,
  (i) In ascending order, construct a max heap and perform remove max.
  (ii) To sort in descending order, construct minheap and perform remove min operation.

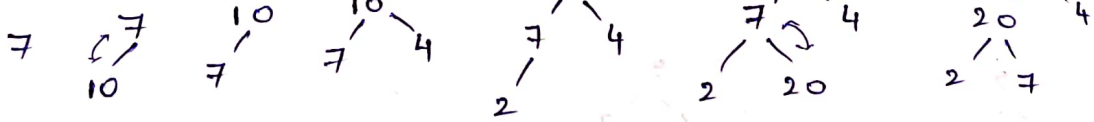
* Time complexity of heap sort
  = Time complexity of insertion + deletion.
  = $o(n \log n) + o(n \log n)$
  = $o(2n \log n)$
  = $o(n \log n)$.

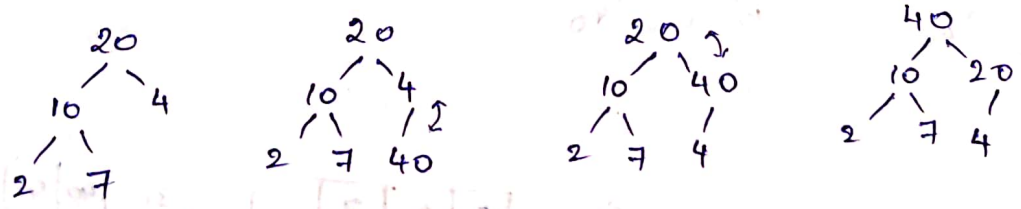
21-9-19

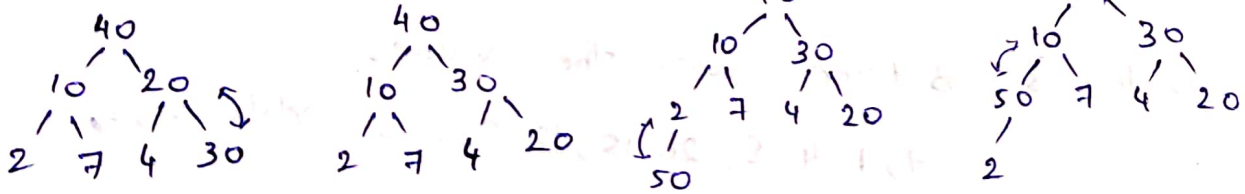
* Do the heap sort in the ascending order for the following values: ~~(max heap)~~
  7, 10, 4, 2, 20, 40, 30, 50

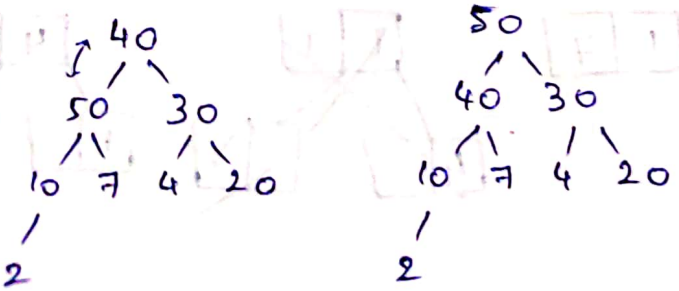
Sol

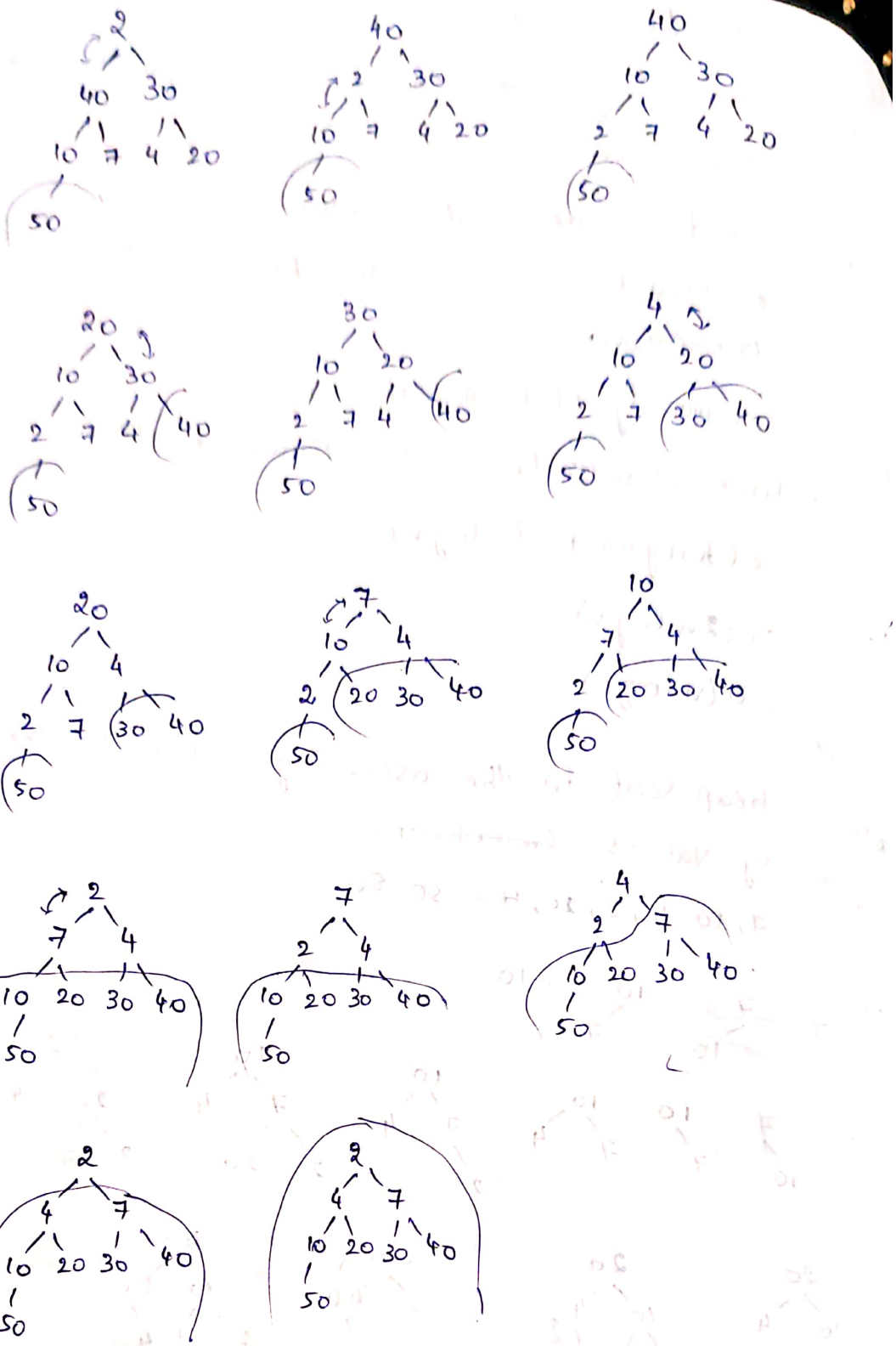












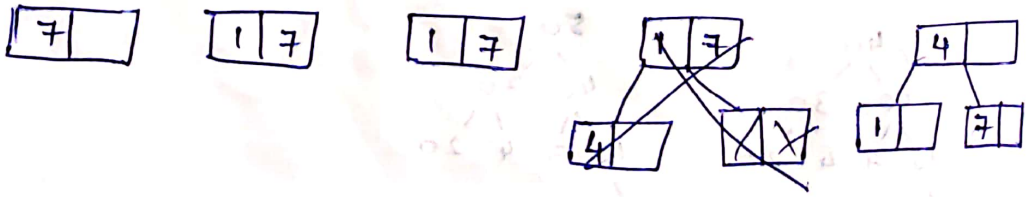
Array representation is

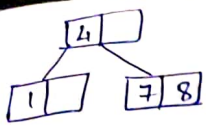
2	4	7	10	20	30	40	50

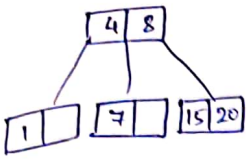
* order-3 B-tree with the values

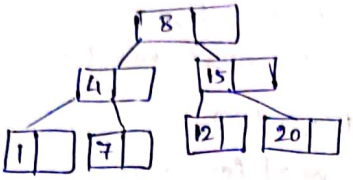
7, 1, 4, 8, 20, 15, 12, 9, 6, 3, Delete 20, 15,

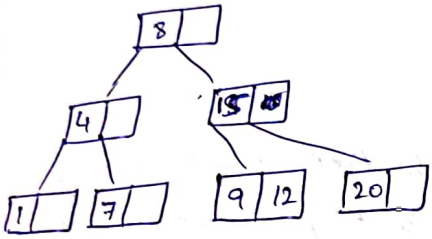
Sol:



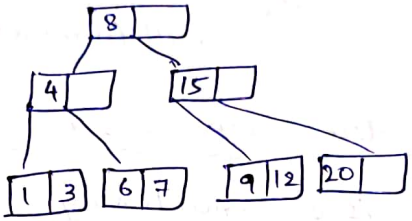




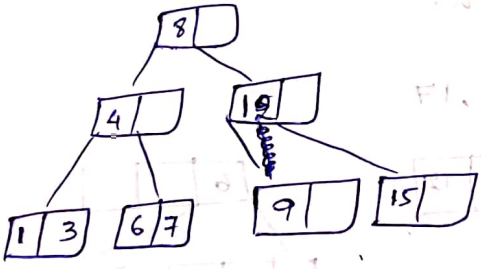


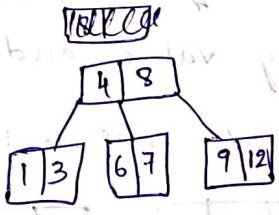


Now, delete 20



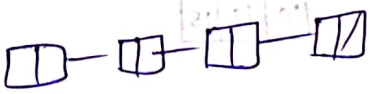
Now, delete 15



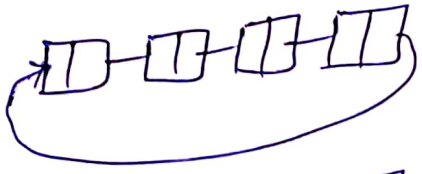


* check whether the linked list is a well formed linked list (or) mal-formed linked list.

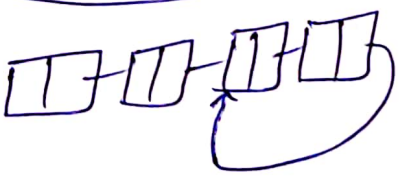
So



→ well formed



→ well formed



→ mal formed

→ mal formed.

```
t1 = head;
t2 = head.next;
while (t1 != t2 && t1 != null && t2 != null)
{
    t1 = t1.next;
    t2 = t2.next.next;
}
if (t1 == t2)
    malformed
else
    well formed.
```

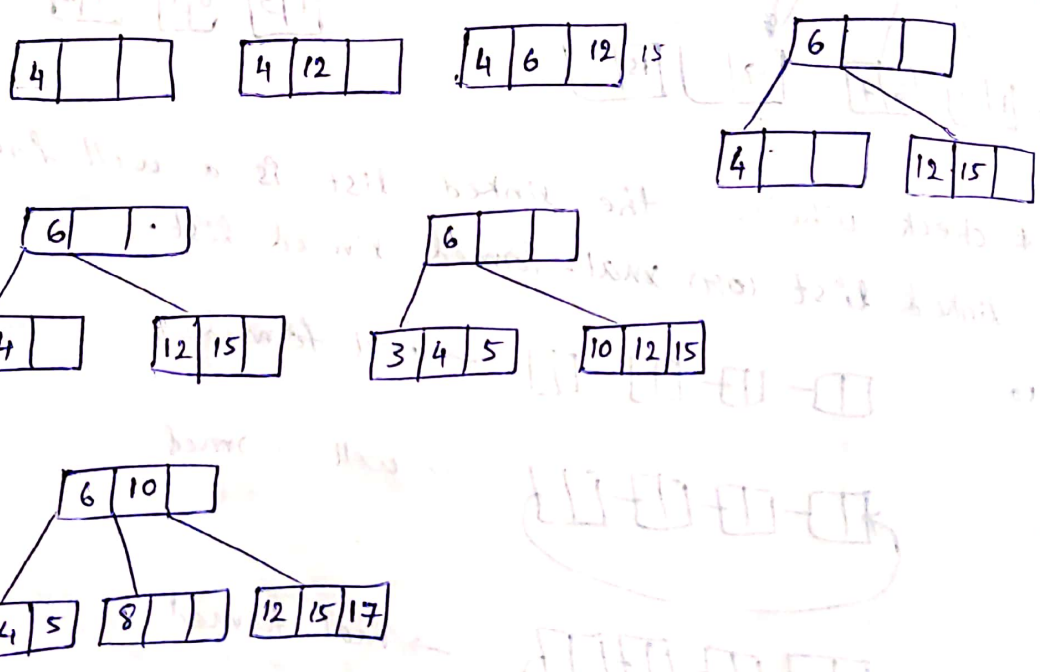
24-9-19

(2,4) tree (or) 2-3-4 tree:-

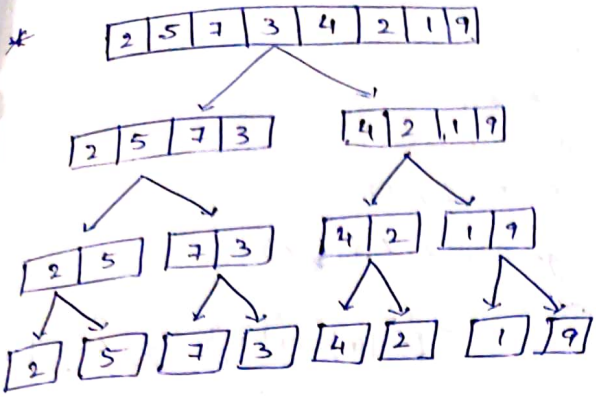
It is a multiway search tree where every internal node has atmost 3 key values (data) and 4 children. Every internal node has minimum 1 key value and 2 children.

* Insert 4, 12, 6, 15, 3, 5, 10, 8, 17

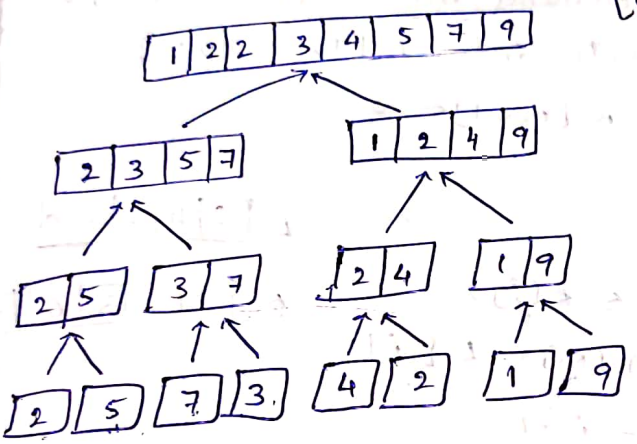
Sol.



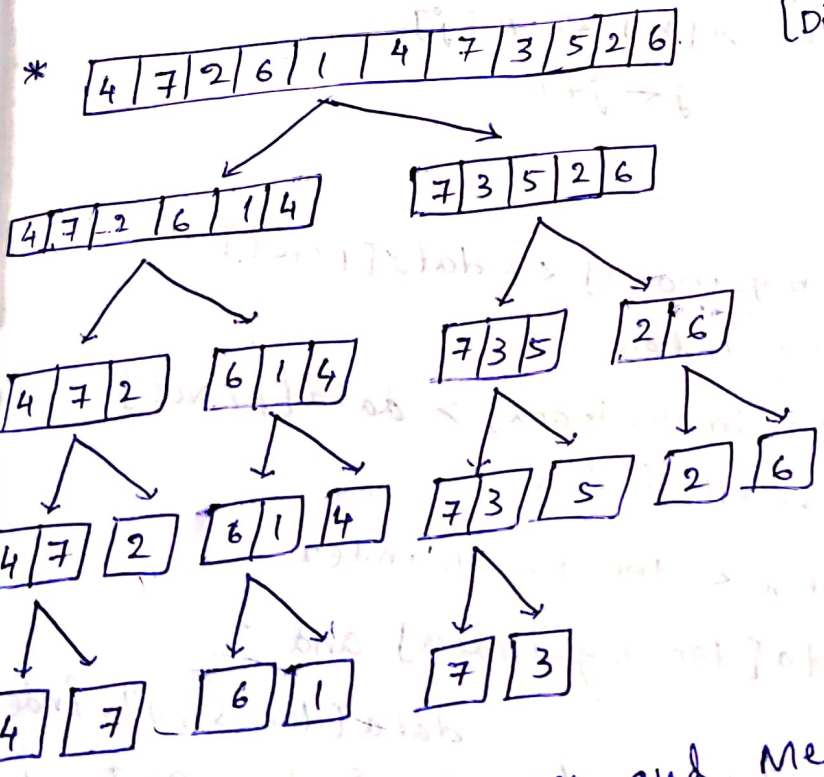
Merge Sort :- [Divide and conquer :-]



[conquer and Merge]



[Divide and conquer]



Then ~~Atera~~ conquer and Merge.

* steps are

1) check for base case 2) Divide 3) conquer 4) conquer 5) combine

Merge-sort (A, p, r)

if (p < r)

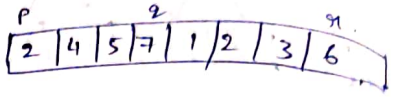
q = ⌊(p+r)/2⌋

Merge-sort (A, p, q)

Merge-sort (A, q+1, r)

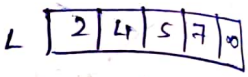
Merge (A, p, q, r)

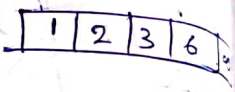
Initial call :- Merge-sort (A, 1, n)



Merge (A, p, q, r)

1. compute n_1 & n_2.
2. copy the 1st n_1 elements into $L[1, \ldots, n_1+1]$ and the next n_2 elements into $R[1, \ldots, n_2+1]$
3. $L[n_1+1] \leftarrow \infty$; $R[n_2+1] \leftarrow \infty$
4. $i \leftarrow 1$; $j \leftarrow 1$
5. for $k \leftarrow p$ to r





do if ($L[i] \leq R[j]$)

then $A[k] \leftarrow L[i]$

$i \leftarrow i+1$

else $A[k] \leftarrow R[j]$

$j \leftarrow j+1$

15-10-19

Quick sort:-

1. while data[too_big_index] <= data[pivot]
   ++ too_big_index
2. while data[too_small_index] > data[pivot]
   -- too_small_index.
3. if too_big_index < too_small_index
   swap data[too_big_index] and data[too_small_index]
4. while too_small_index > too_big_index, go to 1.
5. swap data[too_small_index] and data[pivot_index]

* In best
* worst
* In worst
→ Quick
{
}

* In best case, time complexity is $O(n \log_2 n)$.
* worst case is when it is already sorted.
* In worst case, time complexity is $O(n^2)$.

→ Quick Sort (start, end)

```
{  if (start < end)
   {
      p = partition (start, end);
      Quicksort (start, p);
      Quicksort ( p+1, end);
   }
Partition (start,end)
 {
      pivot = A[start];
      i = start;
      j = end;
      while (i<j)
      {
         do
         {
            i++;
          } while ( A[i] < pivot);
         do
          { j--;
          } while ( A[j] > pivot);
          if (i<j)
          { swap (A[i], A[j]);
           }
       }
       swap (A[start], A[j]);
       return j;
 }
```

Graph:-

* Traversals → BFS - Breadth First Search
  → DFS - Depth First Search.

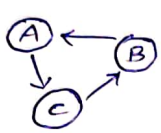
16-10-19

~~* A Graph G is a set V of vertices o~~

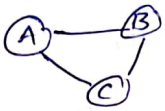
* Graph is a set 'V' of vertices and a collection 'E' of pairs of vertices from 'V' called edges.

*Types of graph
- Directed - Ex:- inheritance
- undirected - Ex:- Railway lines, water & current flow
- Mixed - Ex:- city map.

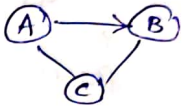
*



Directed graph

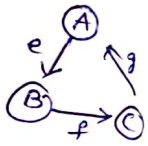


undirected graph.

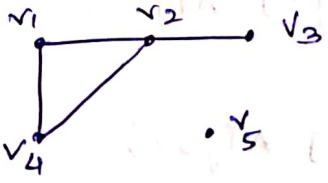


Mixed graph.

* End vertices/end points
* In Directed graph - origin, destination
* Adjacent vertices ~~[illegible]~~
* Incident [Ex:- edge e is incident on B]
* outgoing edge, incoming edge.
* deg(v) → No. of incoming edges + outgoing edges.
* Indeg(v), Outdeg(v)
* deg(v) = Indeg(v) + Outdeg(v)

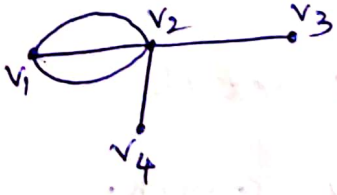


Ex:-



$deg(v_1) = 2$
$deg(v_2) = 3$
$deg(v_5) = 0$

* Parallel edges



$deg(v_1) = 3$
$deg(v_2) = 5$
$deg(v_3) = 1$
$deg(v_4) = 1$.

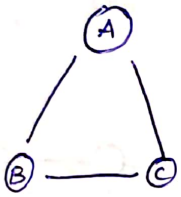
* Parallel edge / Multiple edge
  ↳ 2 directed edges to have same origin & same destination.
  Ex:- Flight from same origin and destination but at different times.

* self loop
* count no of self loop = 2
* graph having self loop is pseudograph
* Simple graph — does not have parallel edges & self loops.

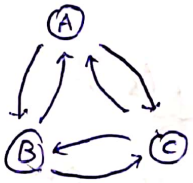
* If there are n vertices,
  Max No. of edges in undirected graph = $\frac{n(n-1)}{2}$
  Max no. of edges in directed graph = $n(n-1)$

* Minimum of (n-1) edges should be present to have a connected graph.

Ex:-



$\frac{3(3-1)}{2} = 3$



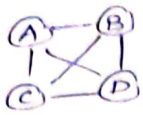
$3(3-1) = 6$

* path
* cycle → A path that starts & ends at same vertex.
* A path is simple if each vertex is distinct.
* A cycle is simple if each vertex is distinct and each edge is distinct except the starting and ending points
* Directed path
* Directed cycle.
* simple graph.

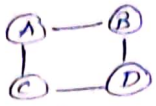
Subgraphs:-
* A subgraph S of a graph G is a graph such that
  ↳ vertices of S are subset of vertices of G
  ↳ edges of S are subset of vertices of G

* A spanning subgraph of G is a subgraph that contains all the vertices of G.

Ex:-



G

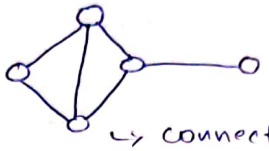


↳ spanning subgraph of G

Connectivity:-

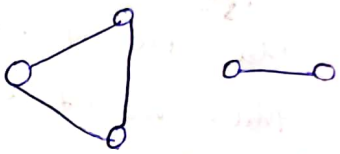
* A graph is connected if there is a path b/w every pair of vertices.

Ex:-



↳ connected graph

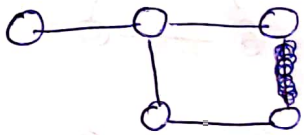
* A connected component of a graph G is a maximal connected subgraph of G.



↳ Not a connected graph

Trees and Forests:-

* A (free) tree is an undirected graph T such that
  - T is connected
  - T has no cycle

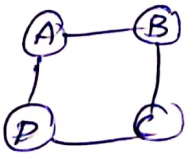


tree

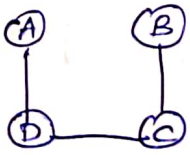
* The connected components of a forest are trees.
* A forest is an undirected graph without cycles.

Forest.

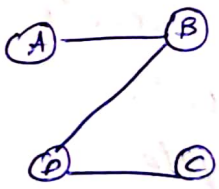
* Spanning Trees & forests:-



Not a Spanning tree



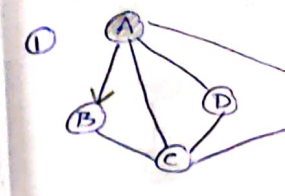
spanning tree

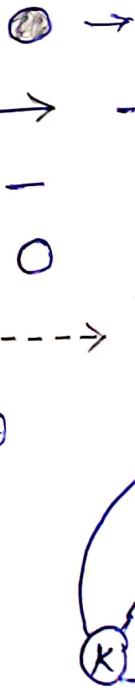


Spanning tree.

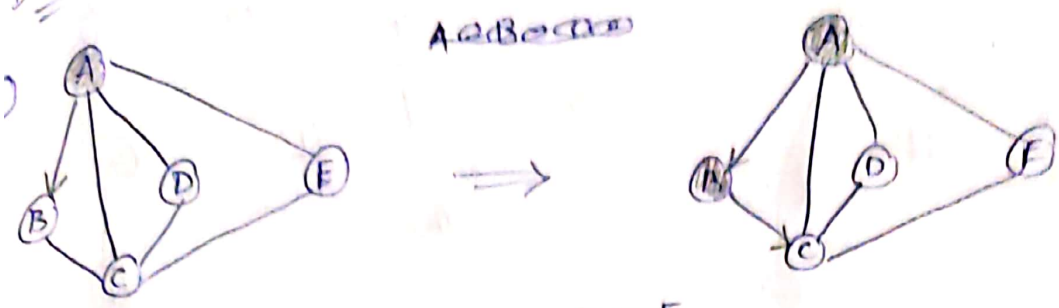
* spanning forest is a combination of spanning trees.

DFS:-

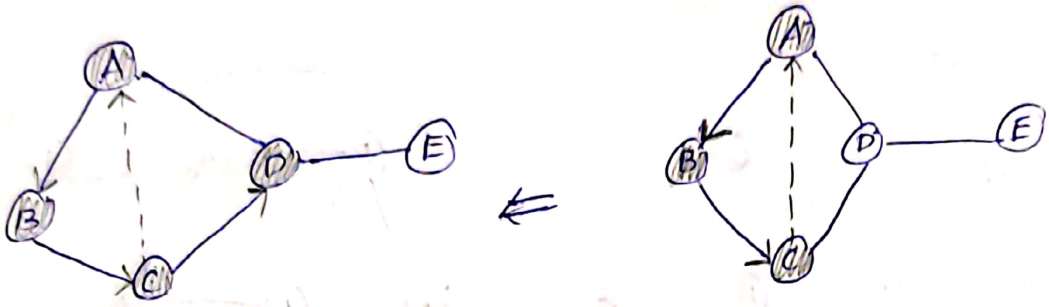


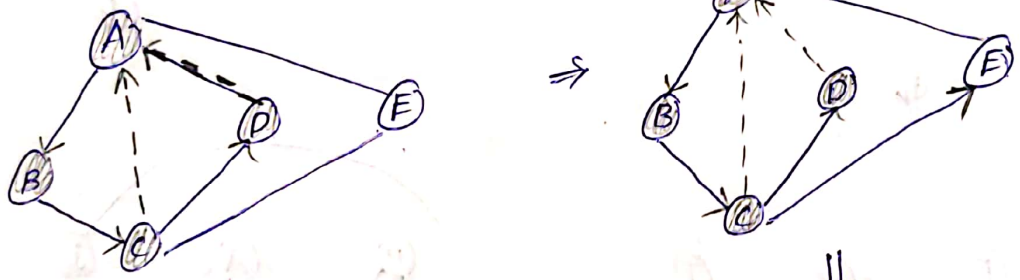


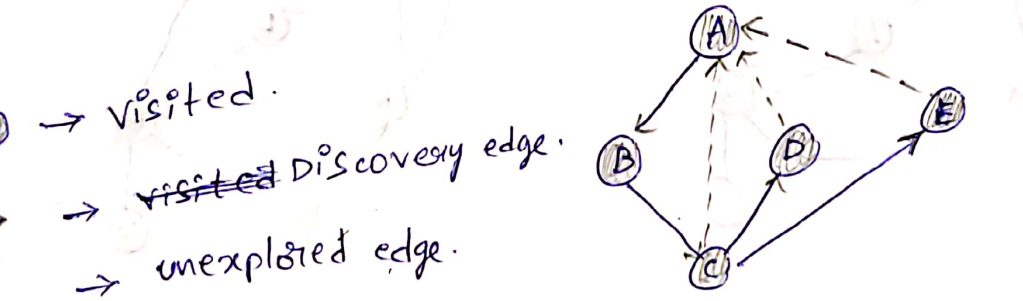
DFS:-



order is A B C D F E







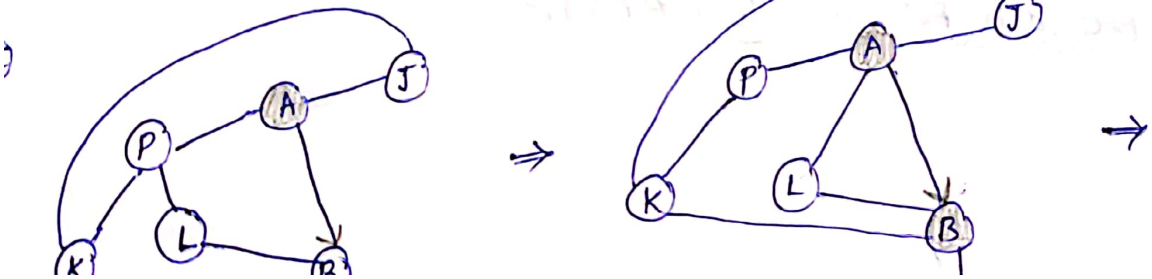
● → visited.

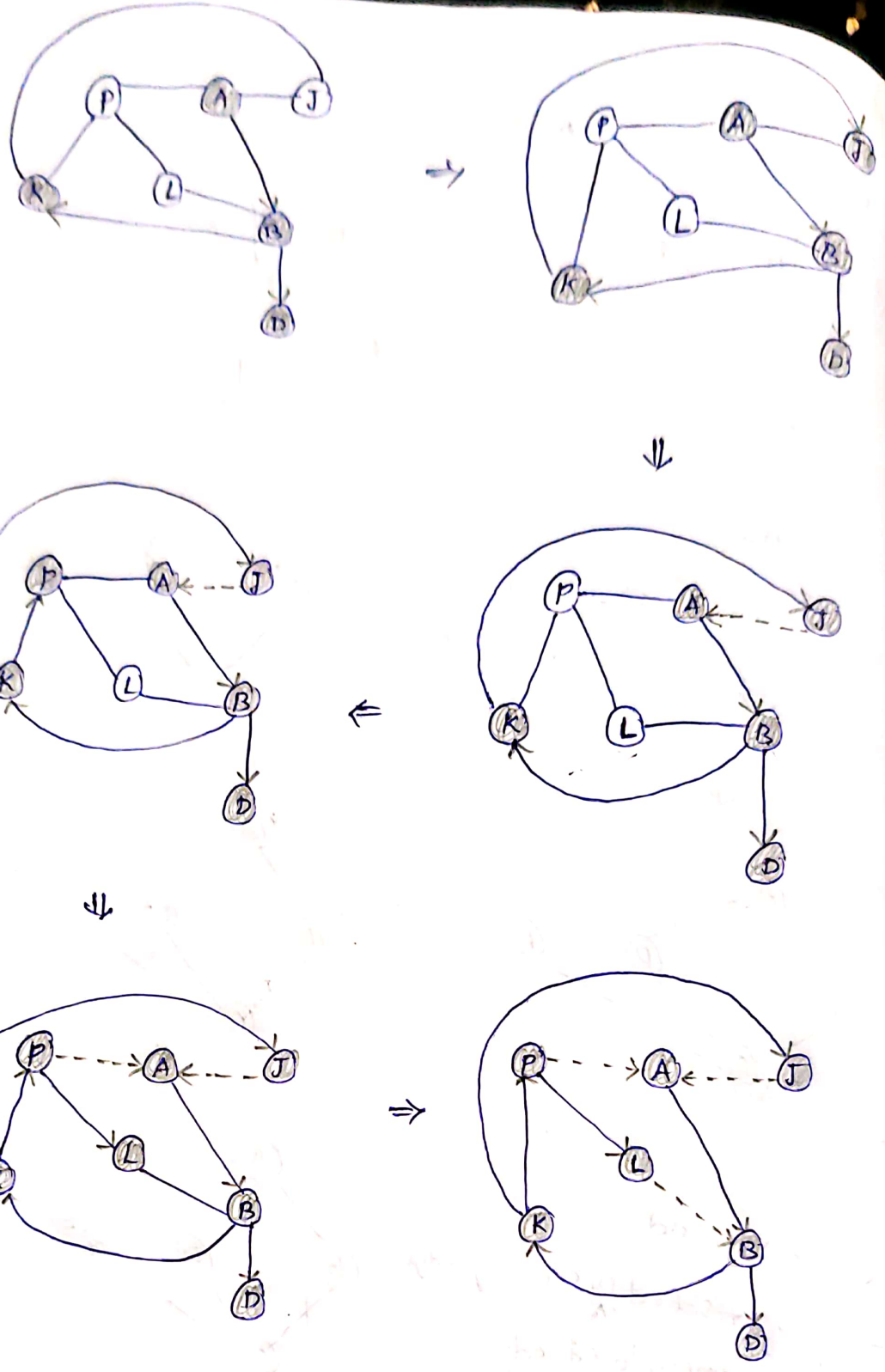
→ → ~~visited~~ Discovery edge.

— → unexplored edge.

○ → unexplored vertex.

----> → Back edge.

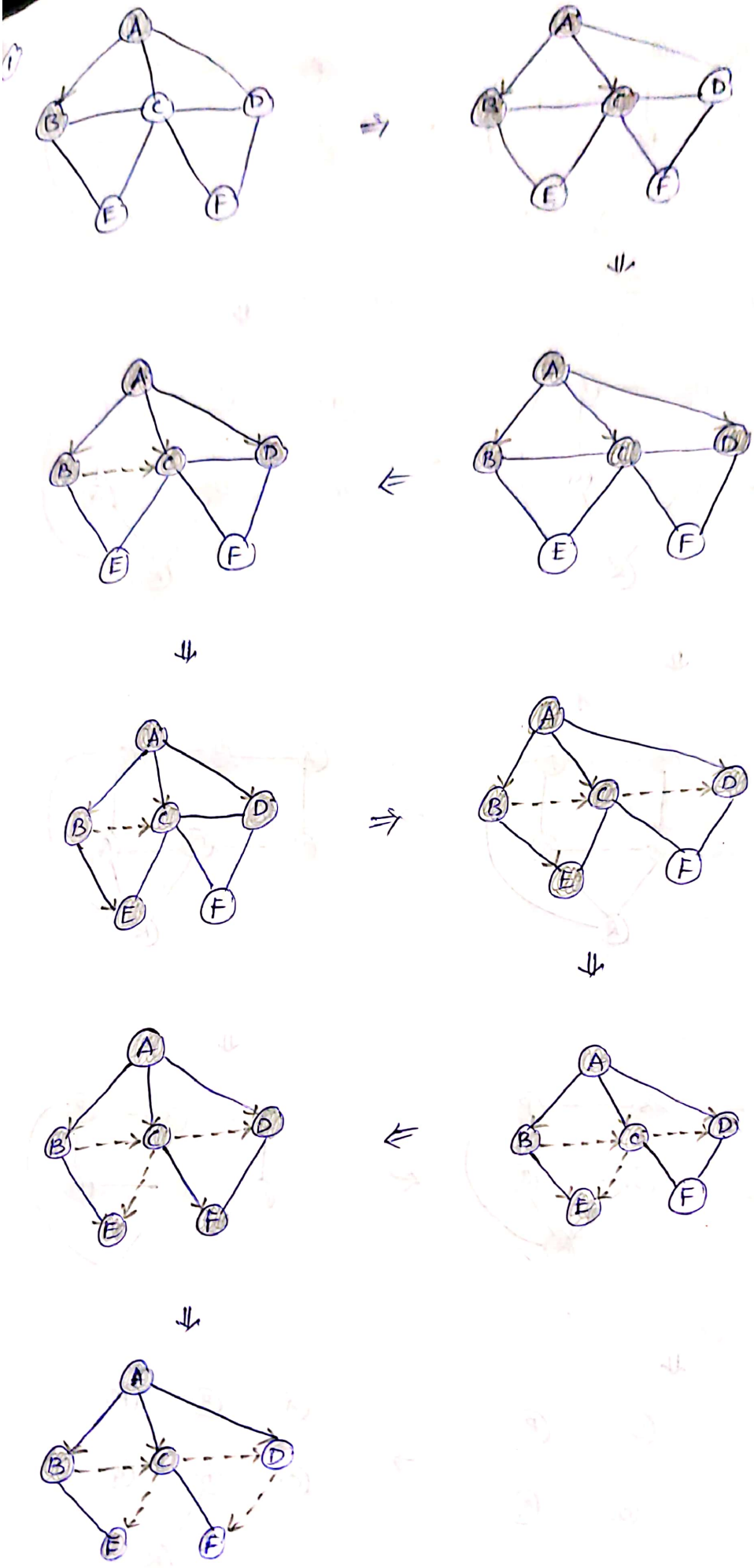


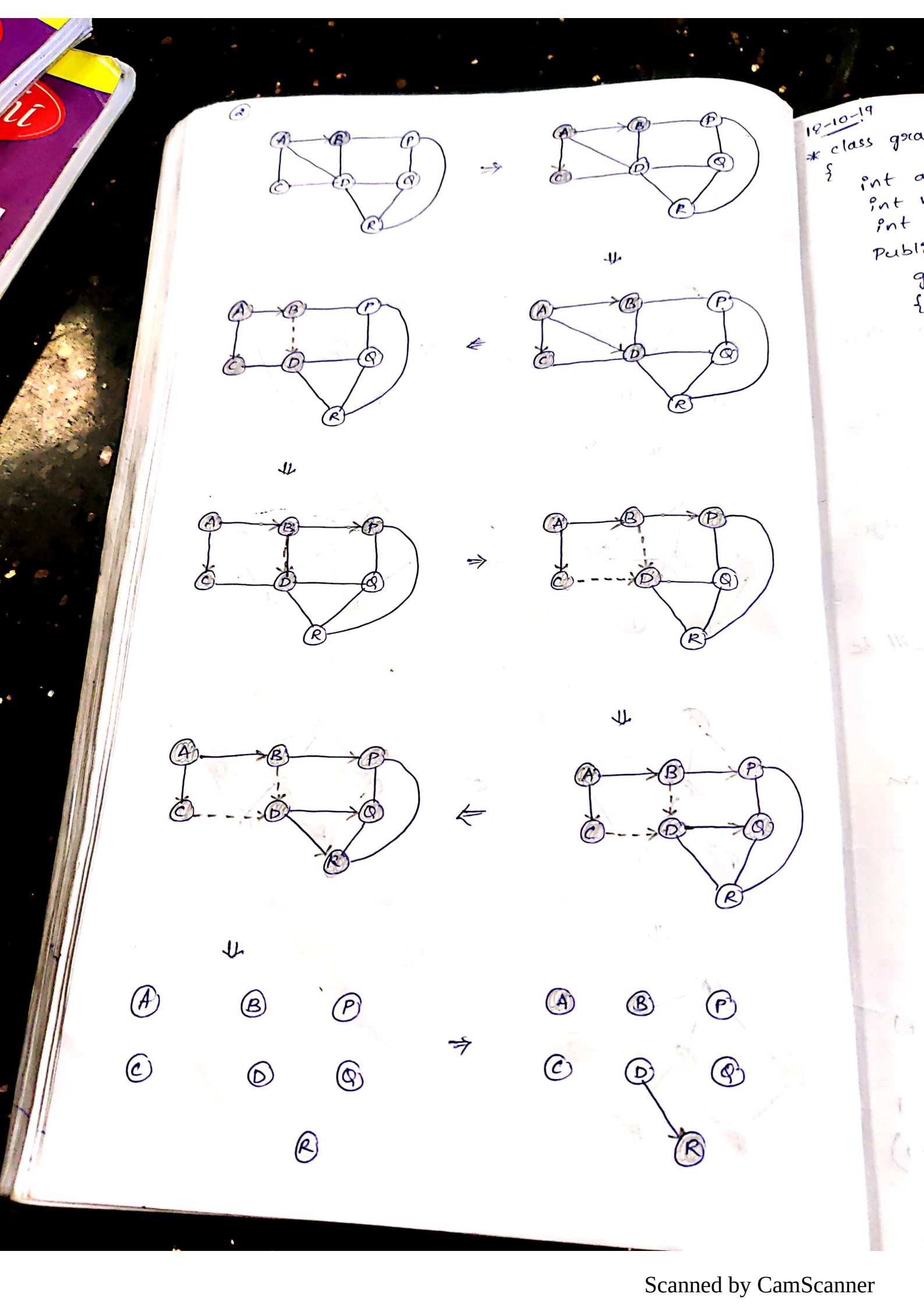


17-10-19 Order is A, B, D, K, J, P, L.

* Maze is an example of DFS.

Breadth For Search (BFS):-





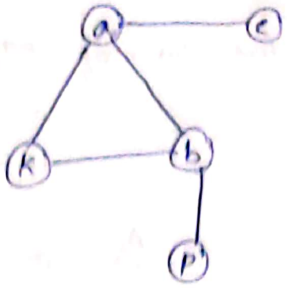
Scanned by CamScanner

18-10-19

* class graph

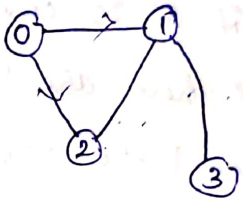
```
{
  int a[10][10];
  int n;
  int visited[10];
  Public:
    graph(int n1)
    {
      n = n1;
      for(int i=0; i<n;
                  i++)
      {
        visited[i]=0;
      }
    }
  void read()
  {
    for(int i=0; i<n;
                i++)
    {
      for(int j=0; j<n;
                  j++)
        read a[i][j];
    }
  }
void dfst(int node)
{
    visited[node]=1;
    simple(node);
    for(int j=0; j<n;
                j++)
    {
      if(visited(j)!=1
            &&
        a[node][j]==1)
        dfst(j);
    }
}
```

* Adjacency matrix



		0	1	2	3	4
		a	b	c	k	p
0	a	0	1	1	1	0
1	b	1	0	0	1	1
2	c	1	0	0	0	0
3	k	1	1	0	0	0
4	p	0	1	0	0	0

Ex:-



	0	1	2	3
0	0	1	1	0
1	1	0	1	1
2	1	1	0	0
3	0	1	0	0

O/P:- 0 1 2 3

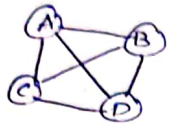
1	1	1	1

visited.

* If G is an undirected graph with 'm' edges, then sum of the degree of all vertices is equal to twice the no. of edges.

$$\sum_{v \in G} deg(v) = 2m$$

Ex:-



Here, $m = 6$

$\Rightarrow$

* If G is a directed graph with 'm' edges, then

$$\sum_{v \in G} indeg(v) = \sum_{v \in G} outdeg(v) = m.$$

* Let 'G' be an undirected graph with 'n' vertices and 'm' edges, then

1) if 'G' is connected, then $m >= n-1$.

i.e; if there are 'n' vertices, then minimum (n-1) edges should be there to make it connected.

2) if 'G' is a tree, then $m = n-1$;

i.e; In a tree, always the no. of edges will be equal to no. of vertices -1.

3) if 'G' is a forest, then $m <= n-1$

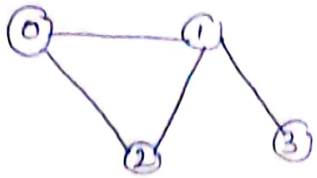
i.e; if there are 'n' vertices, then maximum. there can be (n-1) edges.

19-10-19

```
void bfst(int node)
{
    queue q=new queue();
    initgraph();
    visited[node]=1;
    S.O.println(node);
    q.enqueue(node);
    while(!q.empty())
    {
        q.dequeue();
        for(int j=0; j<n; j++)
        {
            if(visited[j]!=1
                && a[x][j]==1)
            {
                visited[j]=1;
```

```
S.o.println(j);
q.enqueue(j);
}
}
}
}
```

Ex:-



Op:- 0 1 2 3

* Perform quick sort, consider 1st element as pivot element. 10, 17, 8, 4, 15, 20, 1, 25, 2

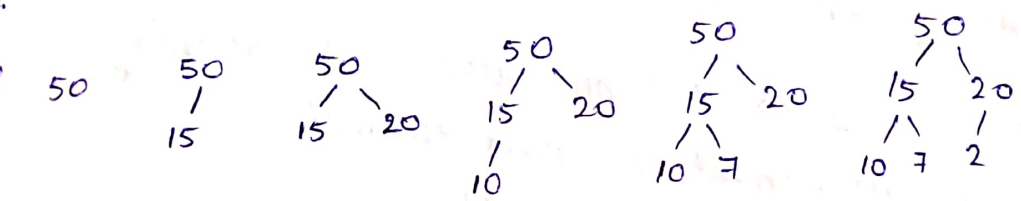
Ans 10, 17, 8, 4, 15, 20, 1, 25, 2.

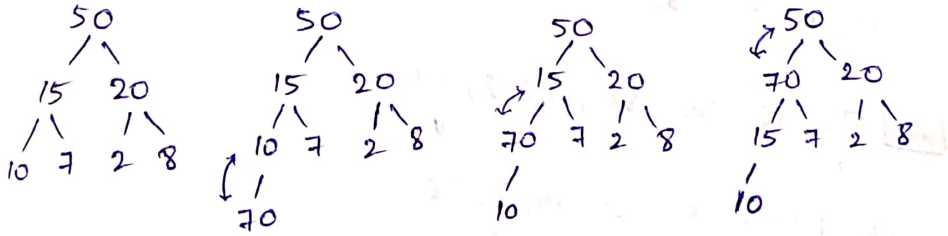
Heap sort Question:-

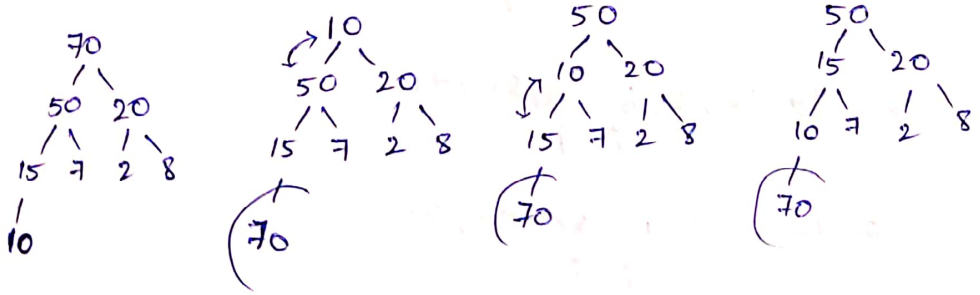
* Perform heap sort and arrange the values in ascending order.

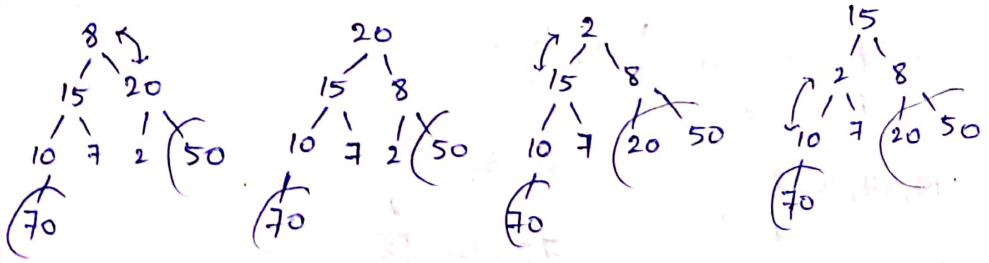
Ex. 50, 15, 20, 10, 7, 2, 8, 70.

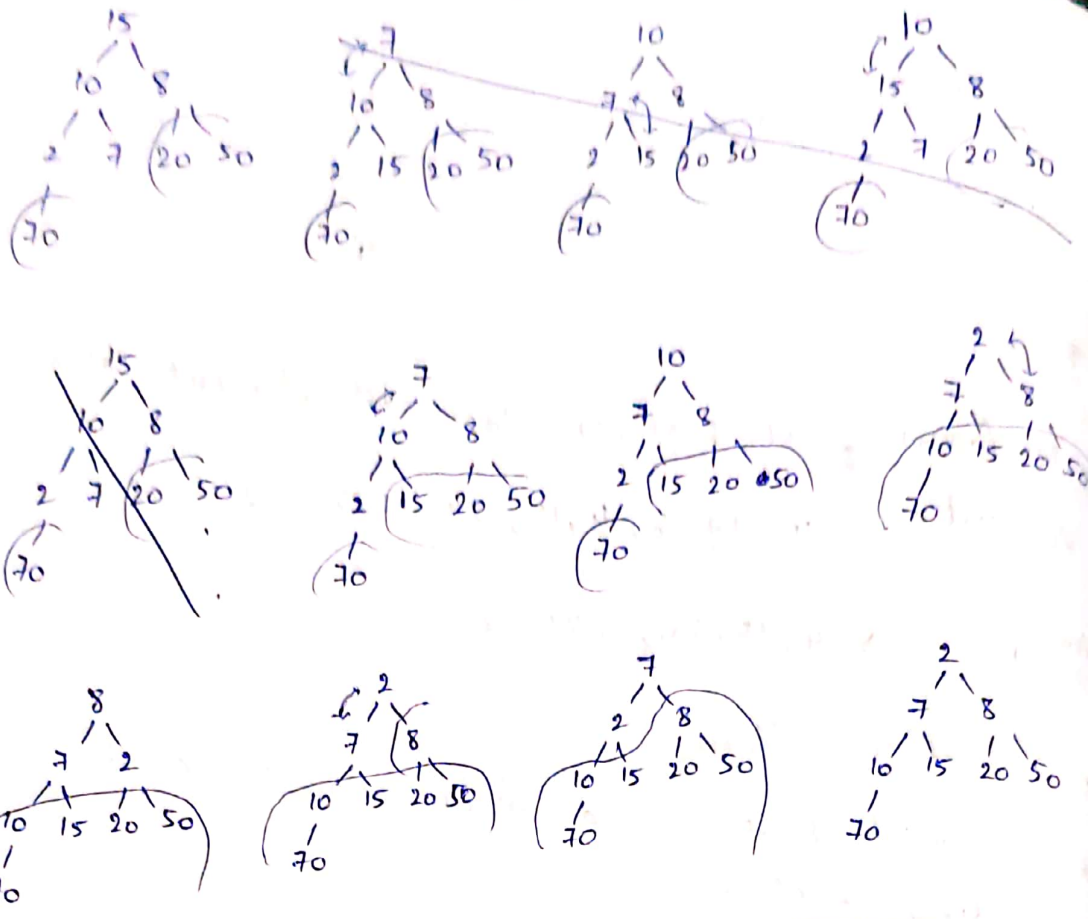
Sol











Array representation is

2	7	8	10	15	20	50	70	

* perform quick sort and arrange in ascending elemen

consider 1st element as pivot element.

20, 70, 10, 50, 80, 19, 2, 17.

Sol

[20], 70, 10, 50, 80, 19, 2, 17.
 i j

20, 17, 10, 50, 80, 19, 2, 70
 i j

20, 17, 10, 2, 80, 19, 50, 70.
 i j

20, 17, 10, 2, 19, 80, 50, 70
 j i

[19], 17, 10, 2, | 20 |, [80], 50, 70
 i j i j

19, 17, 10, 2
 j i

[2], 17, 10, [19]
 i i

80, 50, 70.
 j i

70, 50, [80].

2, 10, 17
i j

~~2, 10, 17~~

2, 17, 10
i i

~~2 10 17~~

~~10, 17~~

2 17 10
i j

17 10
j i

10 17. (∵ j > i)

⇒ 2, 10, 17, 19, 20.

∴ 2, 10, 17, 19, 20, 50, 70, 80.

70, 50
i j

70, 50
j i

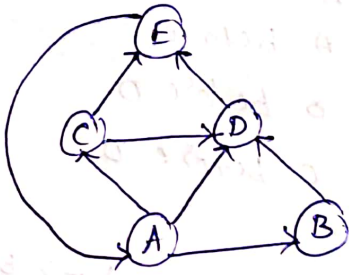
50, 70

⇒ 50, 70, 80.

28-11-19

Digraphs :- [Directed graph]

* A graph whose edges are all directed.



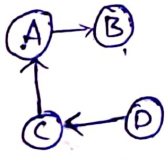
* Applications :-
  One-way streets
  Flights
  Task scheduling.

* Reachability

* Strong connectivity :- Each vertex can reach all other vertices.

Strongly connected components :-

* can be done in O(n+m) using DFS.



Not a strongly connected graph.

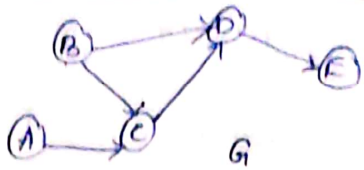
Transitive closure :-

* The transitive closure provides reachability information about a digraph.

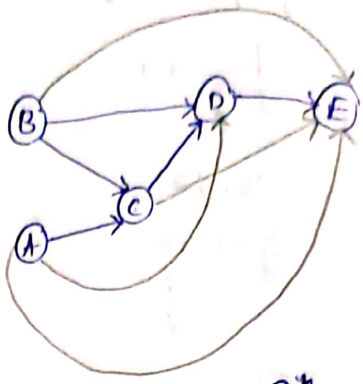
* Given a digraph G,

→ G* has same vertices as G

→ If G has a directed path from u to v ($u \neq v$), G* has a directed edge from u to v.



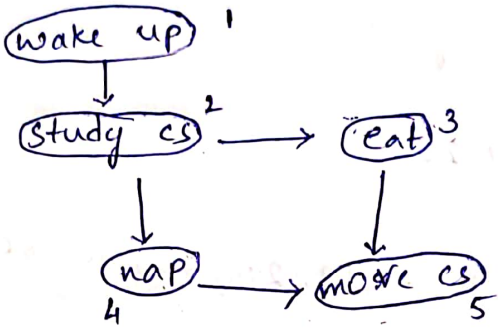
G



G*

Topological Sorting:

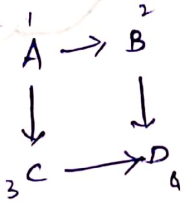
* Number vertices, so that (u, v) in E implies u<v.



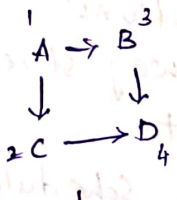
→ A before B
A before C
B before D
C before D.

A → waking up
B → Getting ready.
C → Eating
D → Go to college.

=>



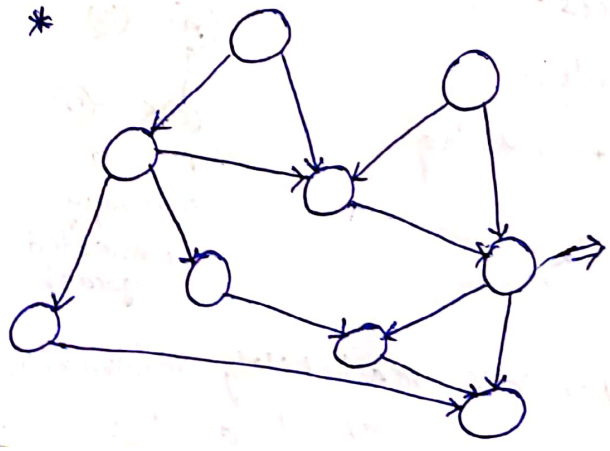
(or)

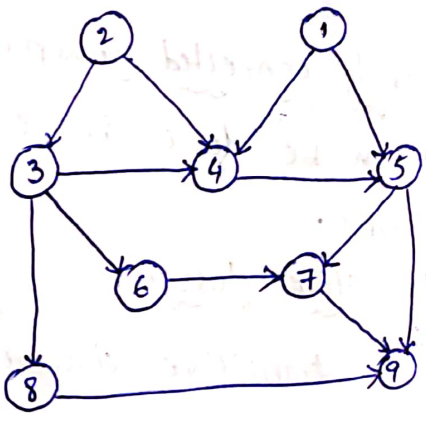


Performing DFS

[A → B → D → B → A
A ← C ← D ← C ←]

*

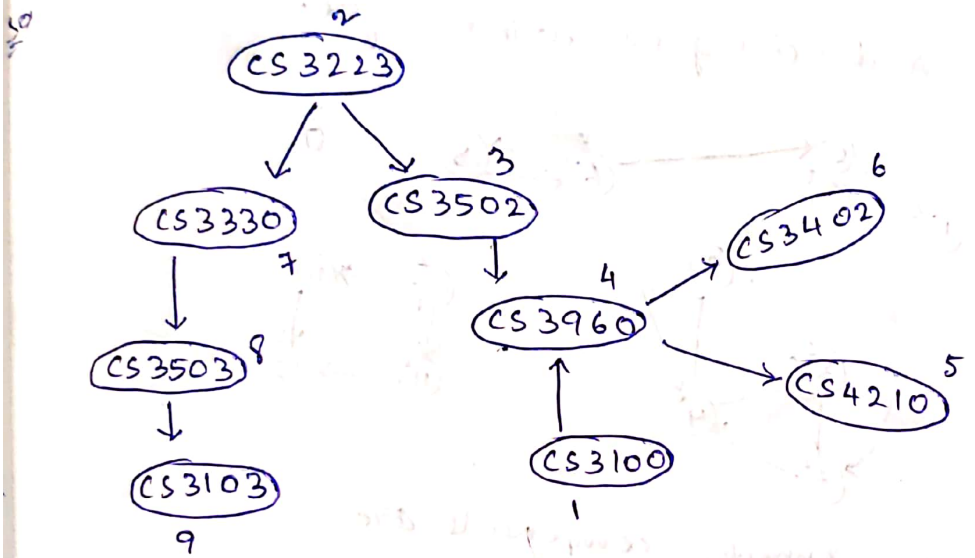


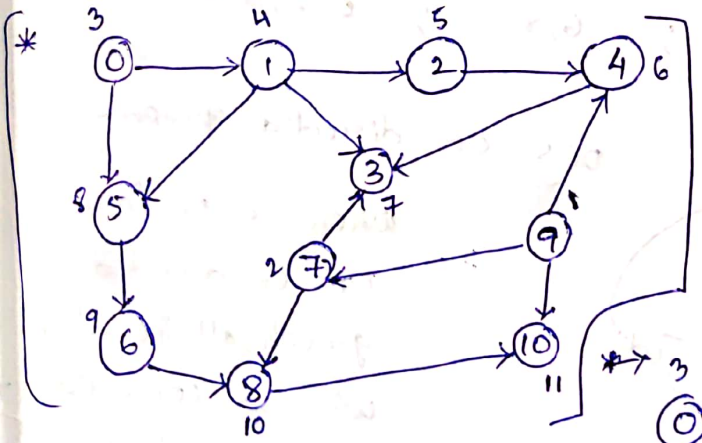


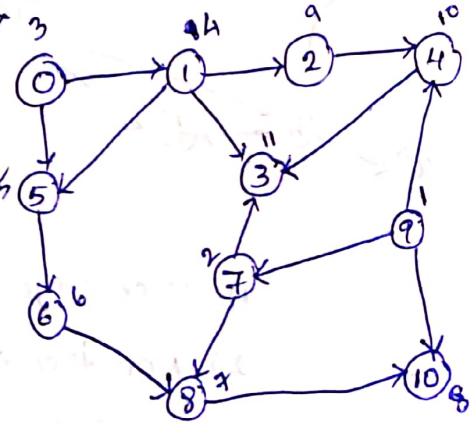
* Consider the following subjects and conditions and based on the given condition, draw the graph and find the topological orders.

cs 3223
cs 3502
cs 3330
cs 3100
cs 3503
cs 3103
cs 3960
cs 3402
cs 4210

cs 3223 before cs 3330 & cs 3502
cs 3330 before cs 3503
cs 3502 before cs 3960
cs 3503 before cs 3103
cs 3100 before cs 3960
cs 3960 before cs 3402 & cs 4210



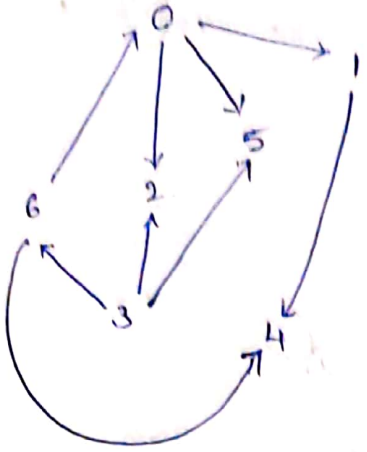


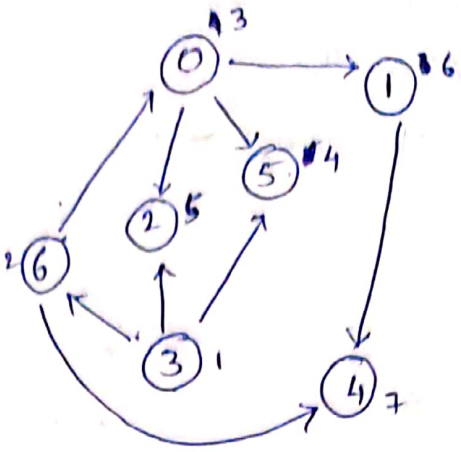


This is alphabetical order.

prefer, this one →

*



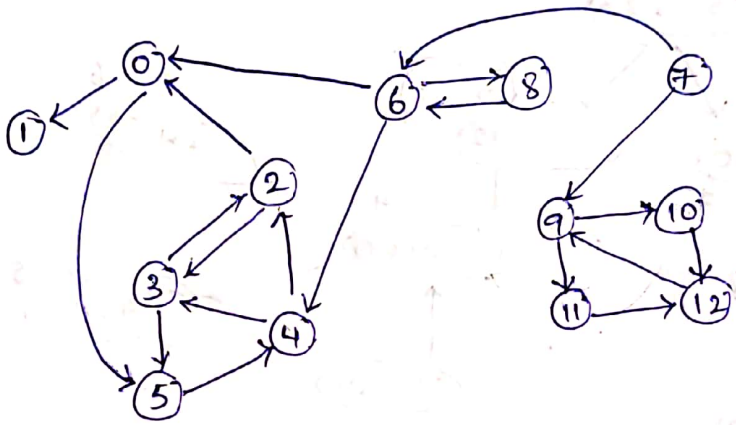


Topological ordering is 3, 6, 0, 5, 2, 1, 4.

* Topological ordering is possible only in DAG (Directed acyclic graph).

* DAG :- A directed graph with no cycle.

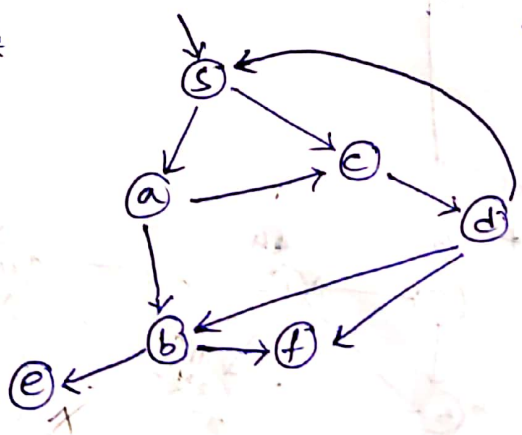
*



Strongly connected components are

9, 10, 11, 12 ; 6, 8 ; 0, 2, 3, 4, 5

*



DFS on directed graph :-

When we perform DFS on directed graph, all the edges will be converted to tree edges / discovery edges & Non-tree edges.

1) Tree edges / Discovery edges.

2) Non-tree edges → back edge
cross edge ← forward edge

* If ↘ is indi present, it indicates the starting point.

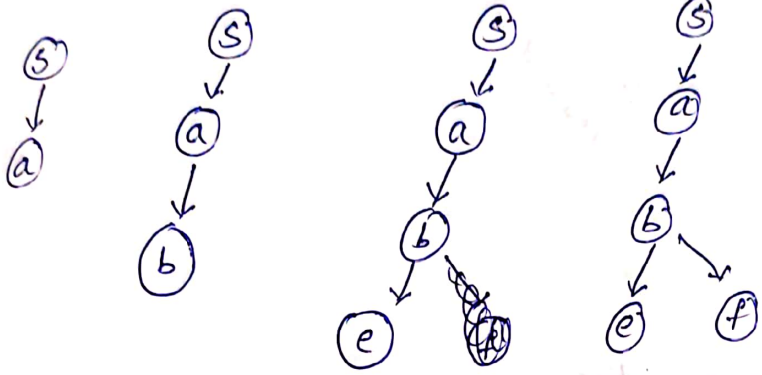
~~If it is~~ connecting to a

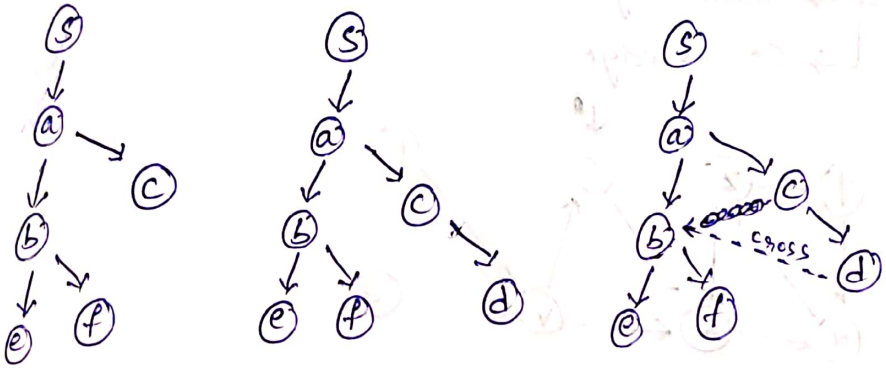
Back edge :- connects the vertex to its ancestor.

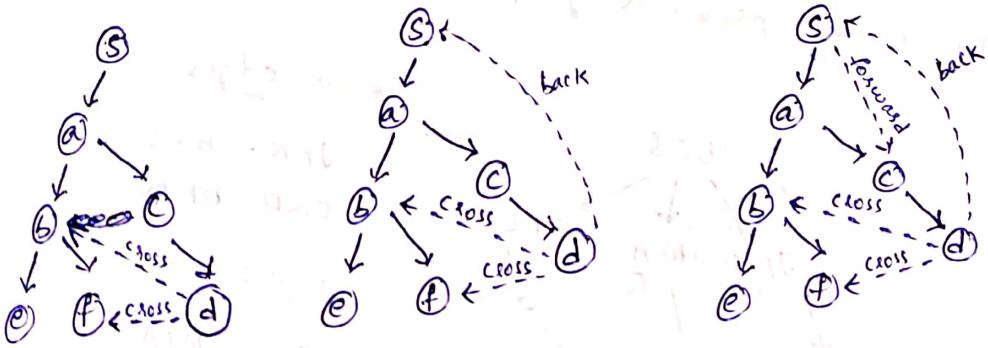
forward edge :- connects the vertex to its descendant.

cross edge :- connects the vertex neither to its ancestor nor its descendant.

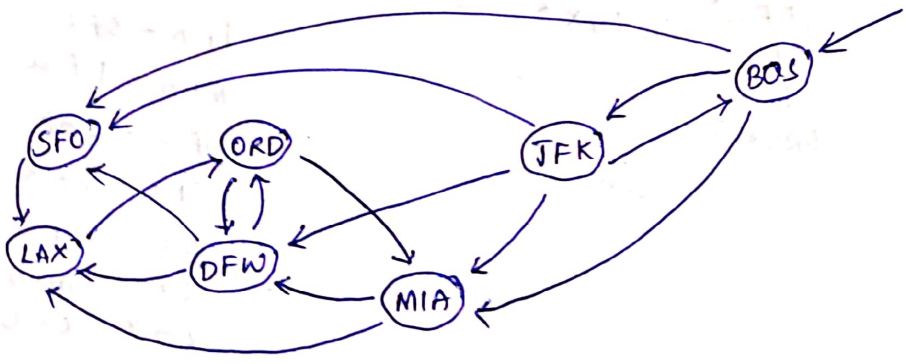
* For the above graph,

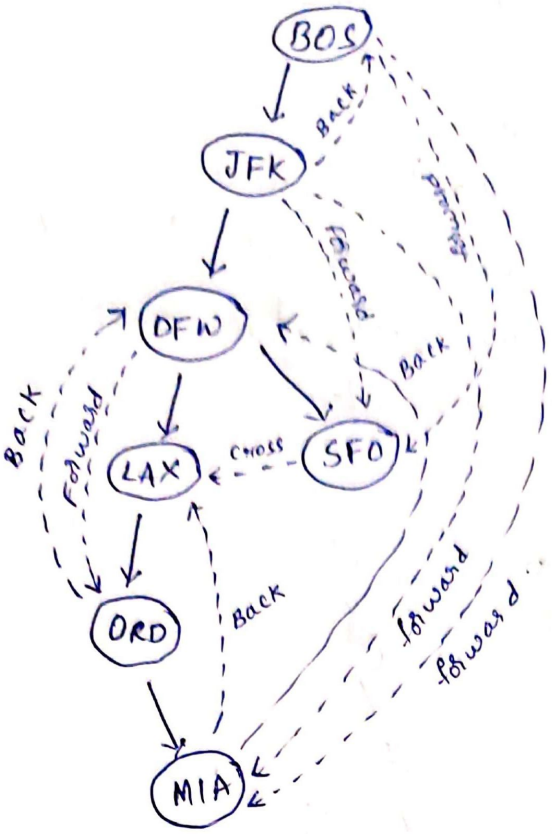






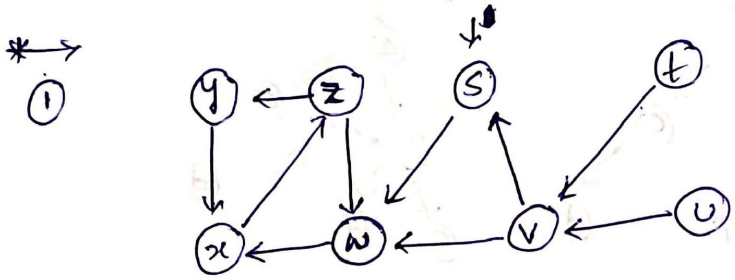
*



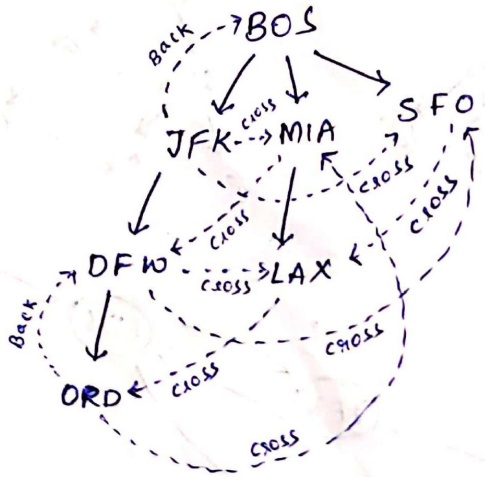


30-11-19

BFS :- → Back edge
→ Cross edge



* → For ~~abc~~ previous graph,



Back edges :-

JFK – BOS
ORD – DFW

cross edges :-

JFK – MIA
JFK – SFO
MIA – DFW
DFW – LAX
SFO – LAX
DFW – SFO
ORD – MIA
LAX – ORD

Undirected :-

DFS	BFS
1. All unexplored edges will be converted to discovery edges and Back edges.	1. All unexplored edges will be converted to discovery edges & cross edge

Directed :-

DFS	BFS
1. All unexplored edges will be converted to discovery edge, back edge, cross edge and tree forward edge.	1. All unexplored edges will be converted to discovery edge, back edge and cross edge.

Applications of DFS & BFS in directed graph :-

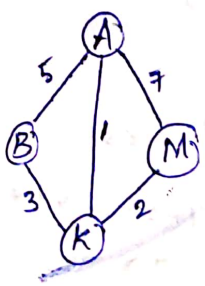
1. Can check reachability.
2. Can find the strong connectivity in the graph.
3. Topological ordering.
4. DAG or cycle detection. [DAG - Directed Acyclic graph]
5. Path finding.

* Time Complexity = $O(n+m)$, $n \rightarrow$ No. of vertices
$m \rightarrow$ No. of edges.

This is for both DFS & BFS.

Weighted Graph :-

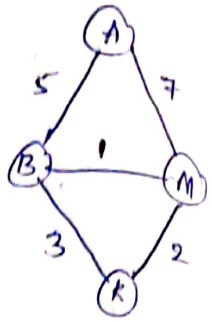
* Based on traffic, distance etc.



* There are 2 algorithms to obtain minimum spanning tree.
[i.e; with minimum cost, we need to cover all the vertices].

1. Prim's Algorithm 2. Kruskal's Algorithm

Minimal Spanning Tree :- with the minimum cost, visit all the vertices.

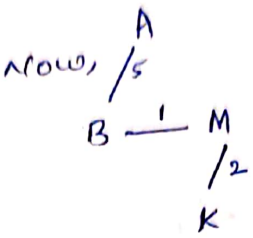


Kruskal's Algorithm:-

1. write in ascending order of weights.

edge	cost
✓ BM	1
✓ MK	2
✗ BK	3
✓ BA	5
✗ AM	7

Now,

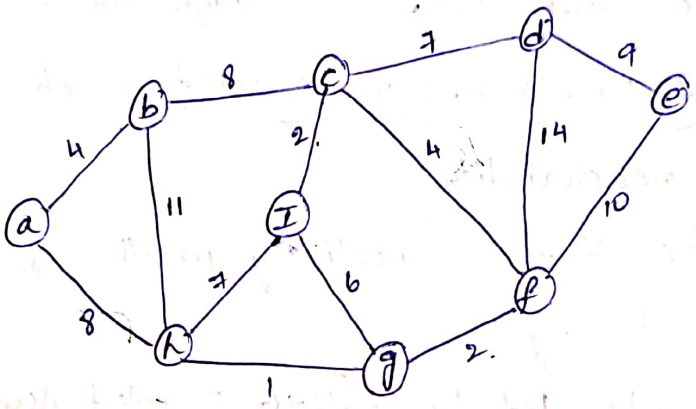


we need to avoid the formation of cycle.

⇒ Total cost = 5+1+2 = 8.

3-12-19

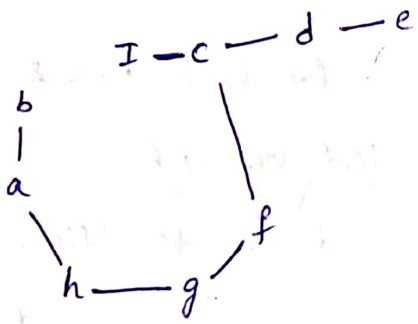
*→



Solve using Kruskal's Algorithm.

Sol.

edge	cost
✓ hg	1
✓ gf	2
✓ cI	2
✓ ab	4
✓ cf	4
✗ Ig	6
✓ cd	7 → 7
✗ Ih	
✗ bc	8
✓ ah	8
✓ de	9
✗ ef	10
✗ bh	11
✗ df	14

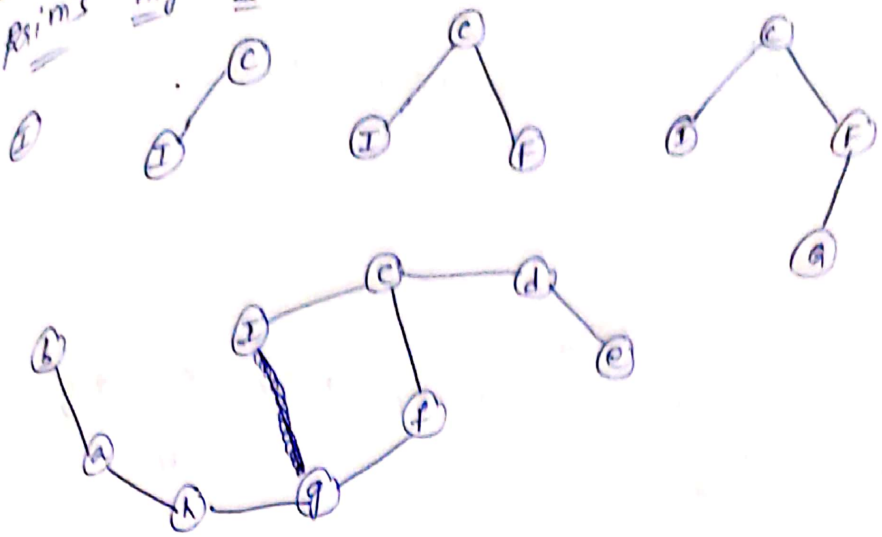


⇒ Total cost

= 1+2+2+4+4+7 +8+9

= 37

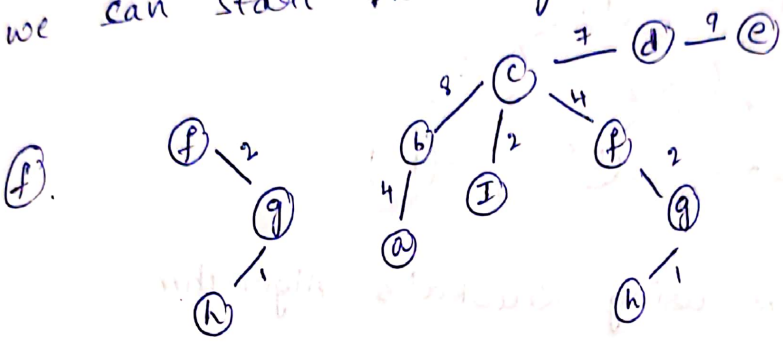
* Prim's Algorithm:-



=> cost = 37.

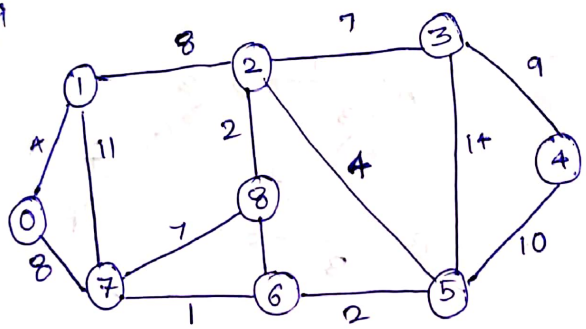
Here, we can start from any vertex.

So,

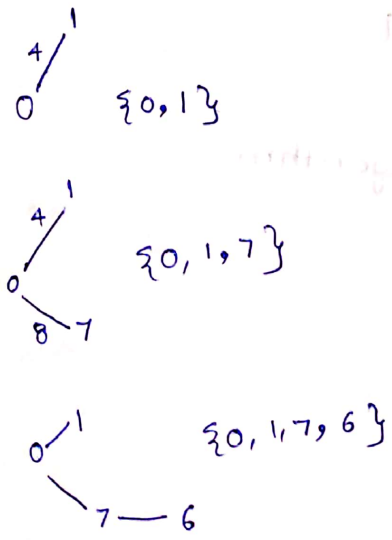


4-12-19

*



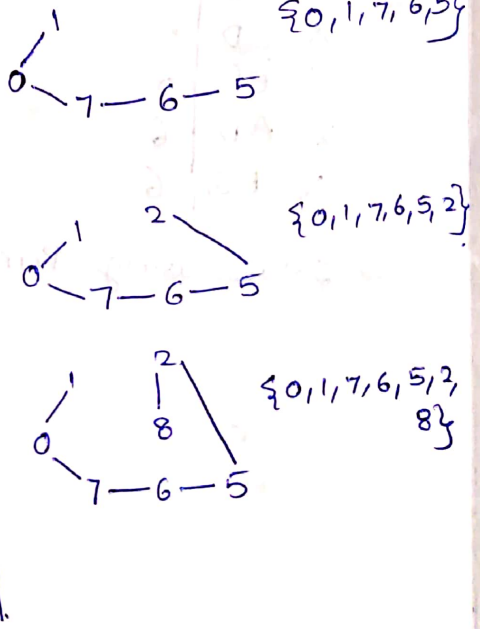
Prim's algorithm.



{0, 1}

{0, 1, 7}

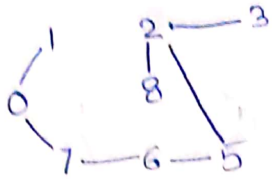
{0, 1, 7, 6}



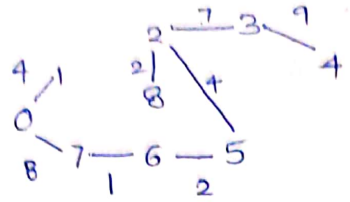
{0, 1, 7, 6, 5}

{0, 1, 7, 6, 5, 2}

{0, 1, 7, 6, 5, 2, 8}



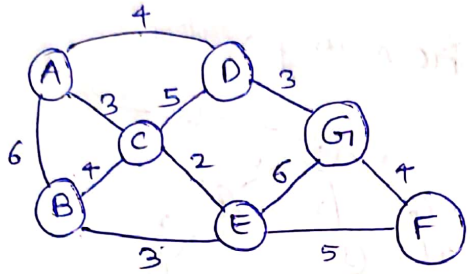
{0, 1, 7, 6, 5, 2, 8, 3}.



{0, 1, 7, 6, 5, 2, 8, 3, 4}

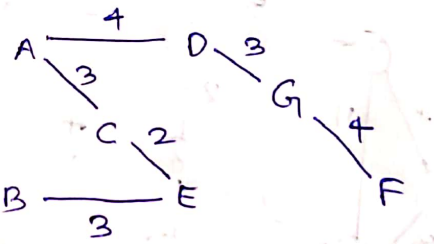
⇒ cost = 37

*



→ Solve using Kruskal's Algorithm

edge	cost
✓ CE	2
✓ AC	3
✓ BE	3
✓ DG	3
X BC	4
✓ AD	4
✓ GF	4
X CD	5
X EF	5
X AB	6
X EG	6



⇒ Total cost

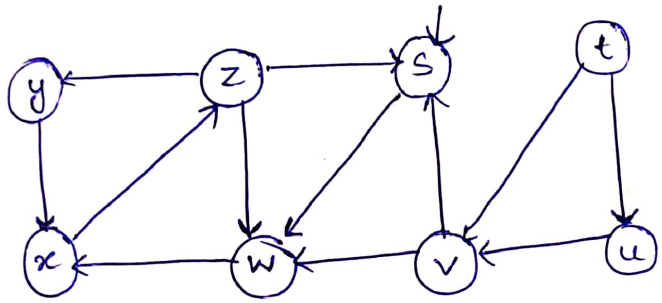
= 3 + 2 + 3 + 4 + 3 + 4

= 19

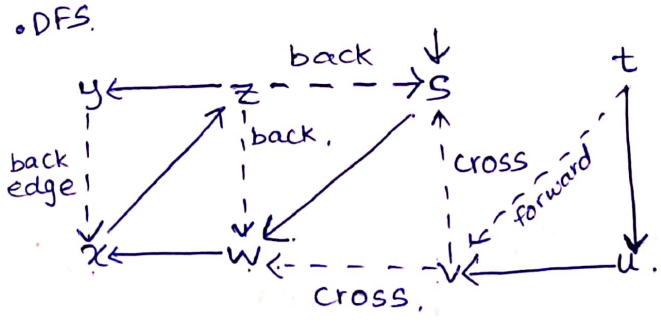
→ Solve using Prim's algorithm.

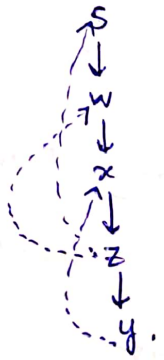
C

Directed DFS & BFS

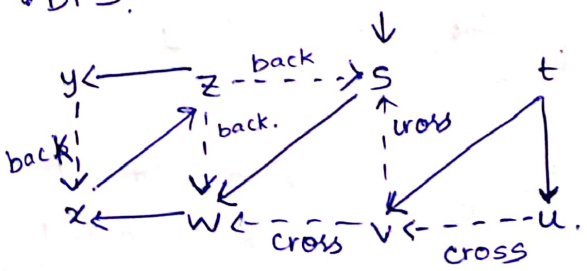


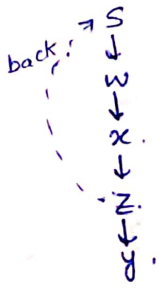
- DFS.



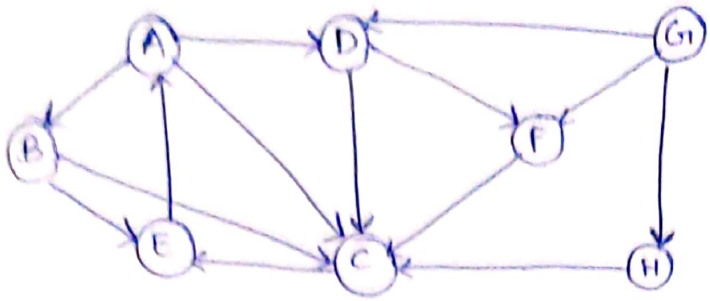


- BFS.





*



5-12-19

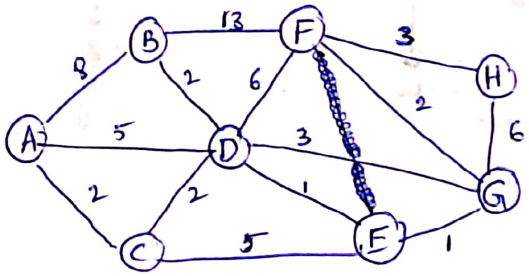
Shortest Path:-

* Dijksla's shortest path Algorithm:-

* In case we have multiple edges, then consider the least cost one and then perform kruskal's or P.sim's algorithm.

[A ⇉ B consider 2]

*(1)



Find shortest path from A to F and A to H.

	A	B	C	D	E	F	G	H
	[0]	∞	∞	∞	∞	∞	∞	∞
A		8_A	[2_A]	5_A	∞	∞	∞	∞
C		8_A		[4_C]	7_C	∞	∞	∞
D		6_D			[5_D]	10_D	7_D	∞
E		[6_D]				10_D	6_E	∞
B						10_D	[6_E]	∞
G						[8_G]		12_G
F								[11_F]
H								

A - C = {A, C} = 2

A - D = {A, C, D} = 4

A - E = {A, C, D, E} = 6

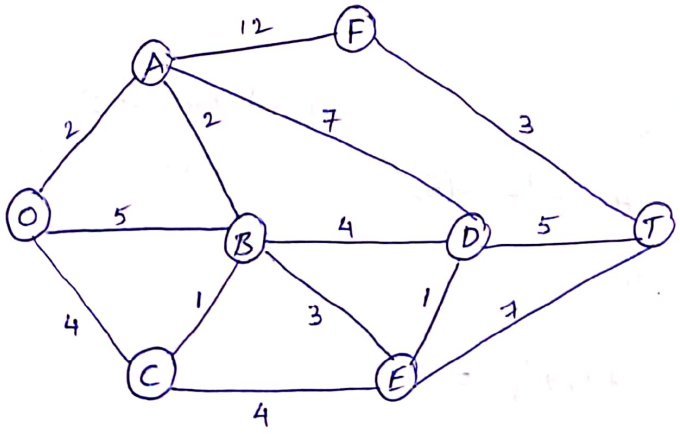
4) A - B = {A, C, D, B} = 6

5) A - G = {A, C, D, E, G} = 6

6) A - F = {A, C, D, E, G, F} = 8

7) A - H = {A, C, D, E, G, F, H} = 11

2)



Find the shortest path from O to T.

	O	A	B	C	D	E	F	T
	0	∞	∞	∞	∞	∞	∞	∞
O		2₀	5₀	4₀	∞	∞	∞	∞
A			4₀	4₀	9₀	∞	14₀	∞
B				4₀	8₀	7₀	14₀	∞
C					8₀	7₀	14₀	∞
E					8₀		14₀	14₀
D							14₀	13₀
T							14₀	
F								

O − A = {O, A} = 2

O − B = {O, A, B} = 4

O − C = {O, C} = 4

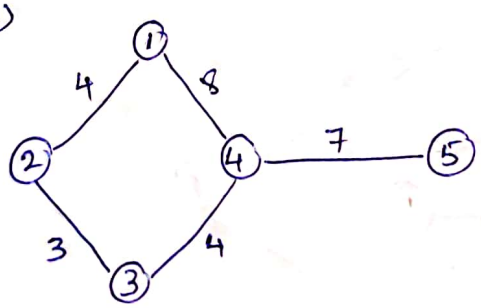
O − E = {O, A, B, E} = 7

O − D = {O, A, B, D} = 8

O − T = {O, A, B, D, T} = 13

O − F = {O, A, F} = 14

*(3)



Find the shortest path from ① to ⑤.

Sol

1 − 2 = {1, 2} = 4

1 − 3 = {1, 2, 3} = 7

1 − 4 = {1, 4} = 8

1 − 5 = {1, 4, 5} = 15

	1	2	3	4	5
	0	∞	∞	∞	∞
1		4_1	∞	8_1	∞
2			7_2	8_1	∞
3				8_0	∞
4					15_4
5					

6-12-19

* ~~Time~~ space complexity to write adjacency matrix is $O(n^2)$.

* 1) Edge List
  2) Adjacency list
  3) Adjacency matrix.

Edge List:- It contains vertex objects & edge objects.

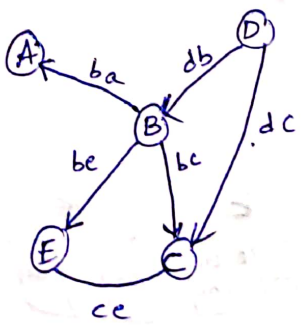
* The vertex object for a vertex 'B' has data members:
  i.e; i) count of incident undirected edges
  ii) count of incoming directed edges
  iii) count of outgoing directed edges

(a) * A boolean indicator of whether the edge is directed or not. ~~is edge object~~

(b) * Reference to the vertex object associated with the end-point vertices of E.

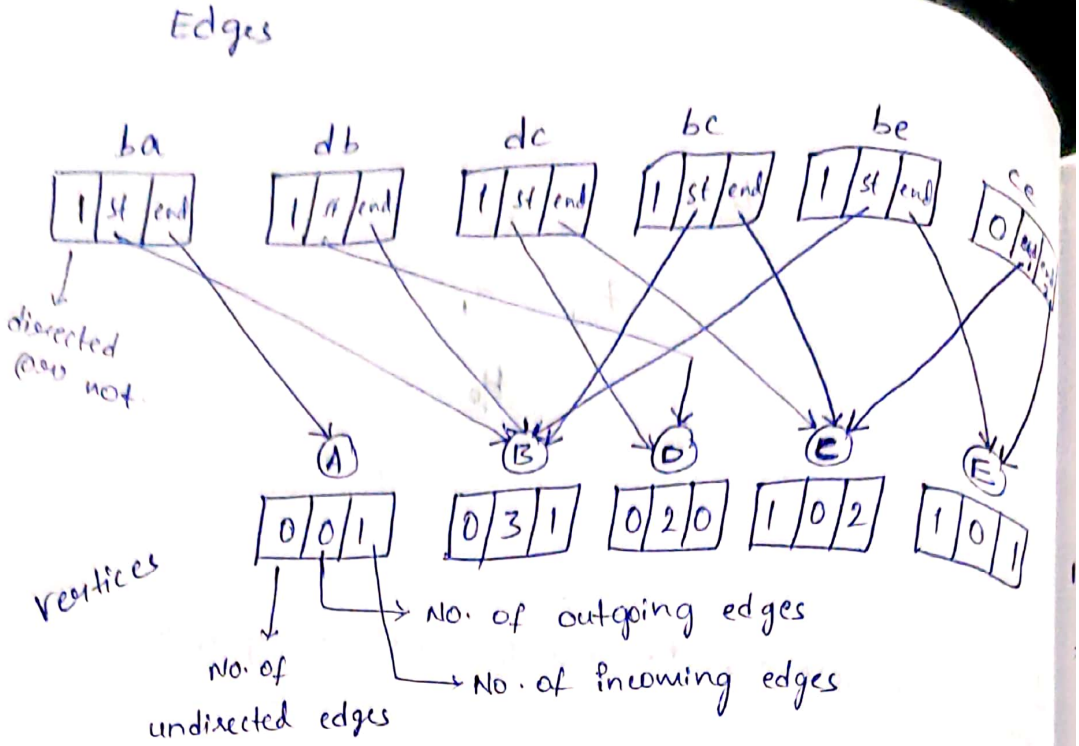
* (a) & (b) are present in edge objects.

* →



No. of edges = m

No. of Vertices = n



*Time complexity :

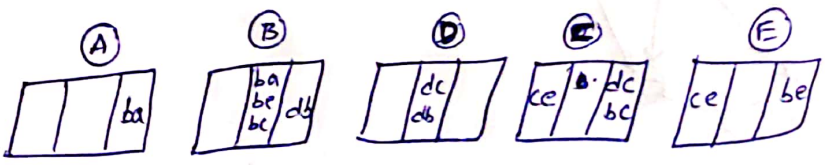
→endvertices()	— O(1)	incidentedges(v)
→incidentEdges()	— O(m)	areadjacent(v,E)
→adjacentvertices()	— O(m)	
→areAdjacent()	— O(m)	
→removevertex()	— O(m)	

* space complexity is O(m+n) whereas for Adjacency list, it is O(n²)

Adjacency List:- It has the edge objects with the same details as edge list and vertex ~~vertex~~ list with the members like

1) which are the undirected edges incident on that vertex
2) which are the outgoing edges
3) which are the incoming edges.

*→



Time complexity :

→ endvertices() — O(1)

→ incidentEdges() — O(deg(v))

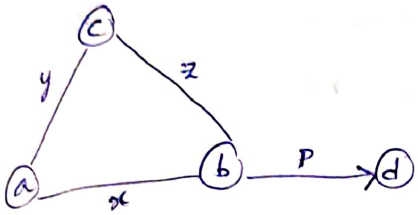
→ adjacentvertices() — O(deg(v))

→ areAdjacent() — O(mindeg(u,v))

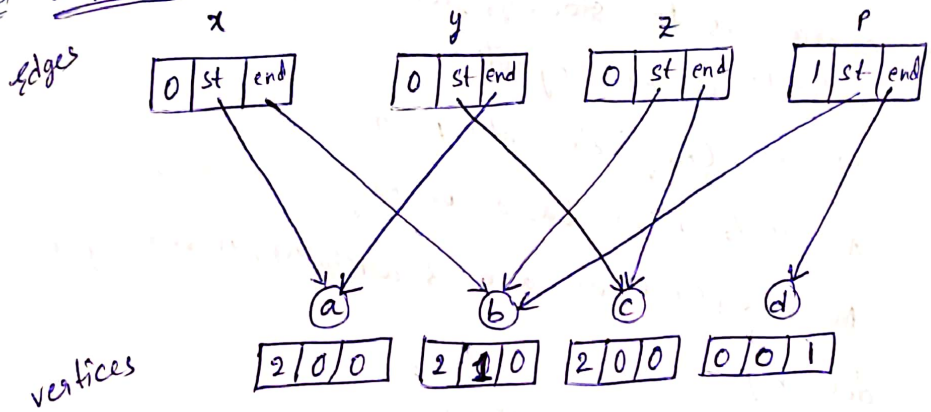
→ removevertex() — O(deg(v))

10-12-19

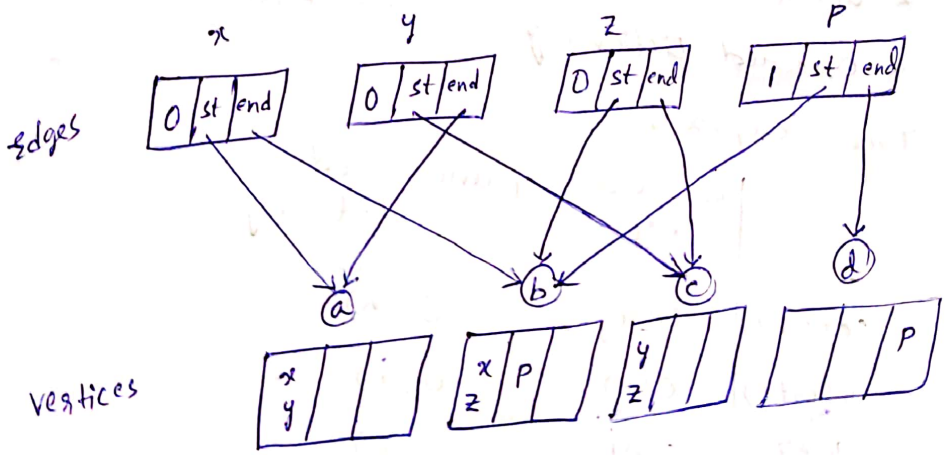
* Draw the edge list, adjacency list & adjacency matrix for the following graph:-



Sol Edge List :-



Adjacency List :-



Adjacency matrix is

	a	b	c	d
a	0	1	1	0
b	1	0	1	1
c	1	1	0	0
d	0	0	0	0

* using adjacency matrix,
  To find out
  1) incidentEdges(v) → check column of v → O(n)
  2) areAdjacent(v1, v2) → O(1)
  3) endVertices(E) → O(1)
  4) removeVertex(v) → O(n²)
  5) adjacentVertices(v) → O(n)

11-12-19

Dictionaries:-
→ containers for storing, retreiving, modifying collections of objects.
→ Items = (key, element) pair
→ Access to item based on key -- O(k,e)
→ Main operations are searching, inserting & deleting.
→ Dictionary is an ADT.

* unordered Dictionary → log files (or audit trails), Hash tables
* ordered Dictionary

Time complexity:-

	ordered Dic	unordered Dictionary
search()	O(logn)	O(n)
insert()	O(n)	O(1)
delete()	O(n)	O(n)

Hash Table:-
* Time complexity for search(), insert() & delete is O(1).

This is what we are excepting.

* But during Hashing, collision might occur.

* But, when we have limited storage, we can't get time complexity as O(1). Here, collision occurs.

Hash functions:-

→ A hash function h maps keys of a given type of integers in a fixed interval [0, N-1]

Ex:- 1) $h(x) = x \bmod N$

2) Extract last digits

3) h(k) = k

Ex:- SSN (Social Security Number)

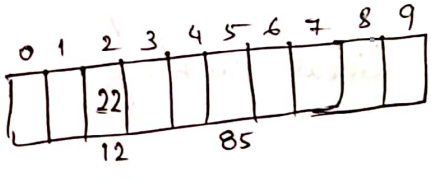
1) Hash code Map Ex:- 50 students to 50 locations

2) Compression Map Ex:- 50 students mapped to 10 locations.

Collision Handling

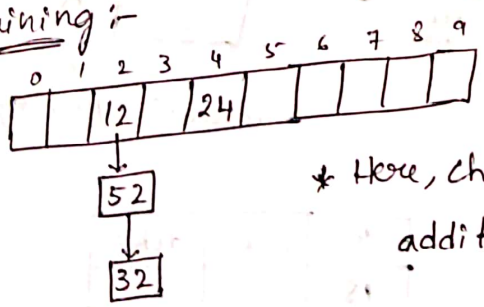
- chaining
- Open addressing
  - Linear Probing
  - Quadratic Probing

$h(k) = k \bmod 10$



Here, collision occured.

Chaining:-

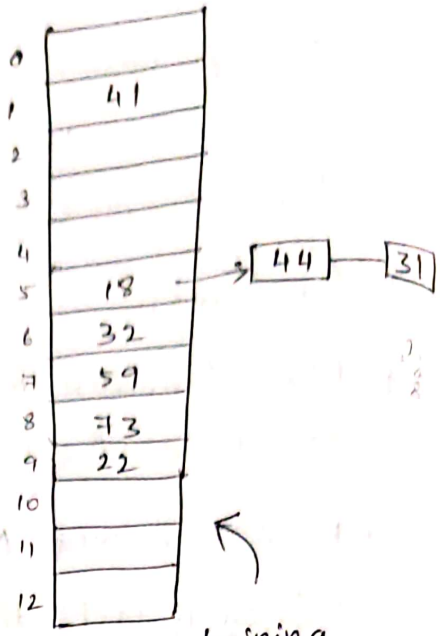


* Here, chaining is simple, but requires additional memory outside the table.

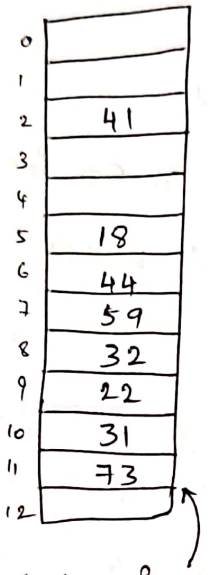
* consider the hashing function $h(x) = x \bmod 13$.

Insert the keys 18, 41, 22, 44, 59, 32, 31, 73.

key	h(key)
18	5
41	2
22	9
44	5
59	7
32	6
31	5
73	8

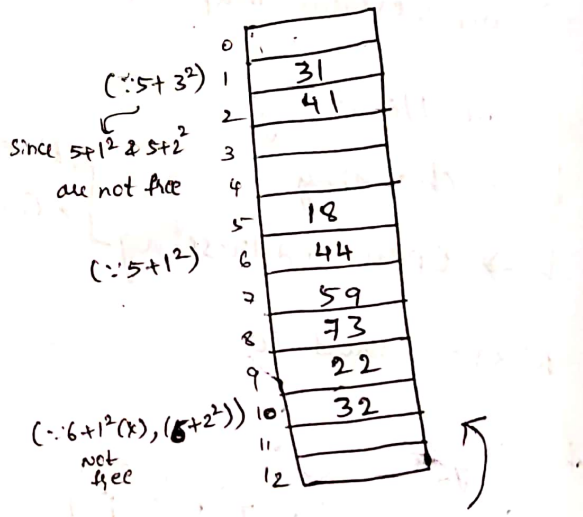


chaining



Linear probing

* Time complexity for searching in linear probing is O(n).



Quadratic probing.

13-12-19

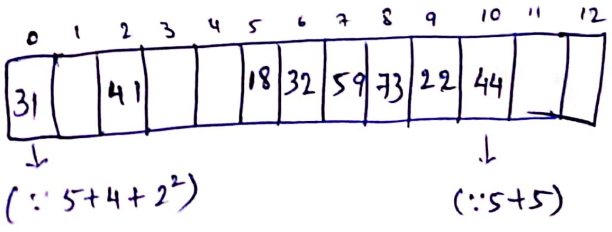
Double Hashing:-

N = 13,

h(k) = k mod 13

d(k) = 7 − k mod 7

k	h(k)	d(k)
18	5	3
41	2	1
22	9	6
44	5	5
59	7	4
32	6	3
31	5	4
73	8	4



(∵5+4+2²)

(∵5+5)

- Load factor:-

Load factor = $\frac{n}{N}$,

n = No. of elements we want to store in hash table

N = Total capacity of hash Table.

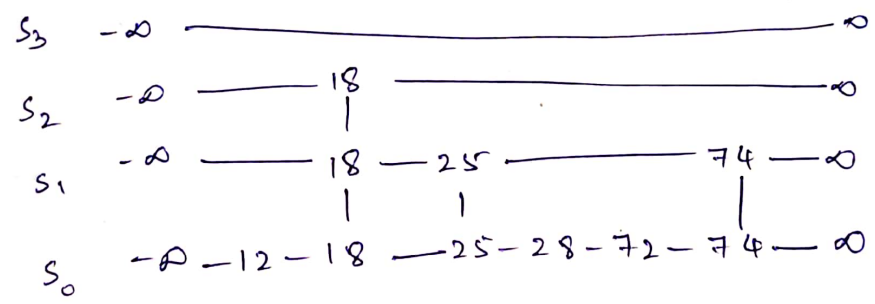
→ For the above problem,

Load factor = $\frac{8}{13} \times 100$.

It is recommened to have < 50%

→ Skip List:-

- Good implementation for a dictionary.
- A series of lists $\{S_0, S_1, \ldots, S_k\}$
- To reduce the time complexity to search an element in (linked list ~~to~~ O(log n), it is introduced.

→ 

→ S_0 has n elements.

→ S_1 has about n/2 elements.

→ S_2 has about n/4 elements.