DESIGN AND ANALYSIS OF ALGORITHMS.

UNIT-1

The word Algorithm comes from the name of Parsian Author. Abu, Ja'far Mohammed ibn Musa al Khowarizmi (825 A.D.) who wrote a book on Mathmatics.

Definition:

(i) An algorithm is a sequence of computational step that transform the input into output. (Thomas H. Cormen)

(ii) An algorithm is a finite set of instructions that, if followed, accomplishes a particular task. (S. Sahni)

Criteria for Algorithm: All algorithm must satisfy the following Criteria:

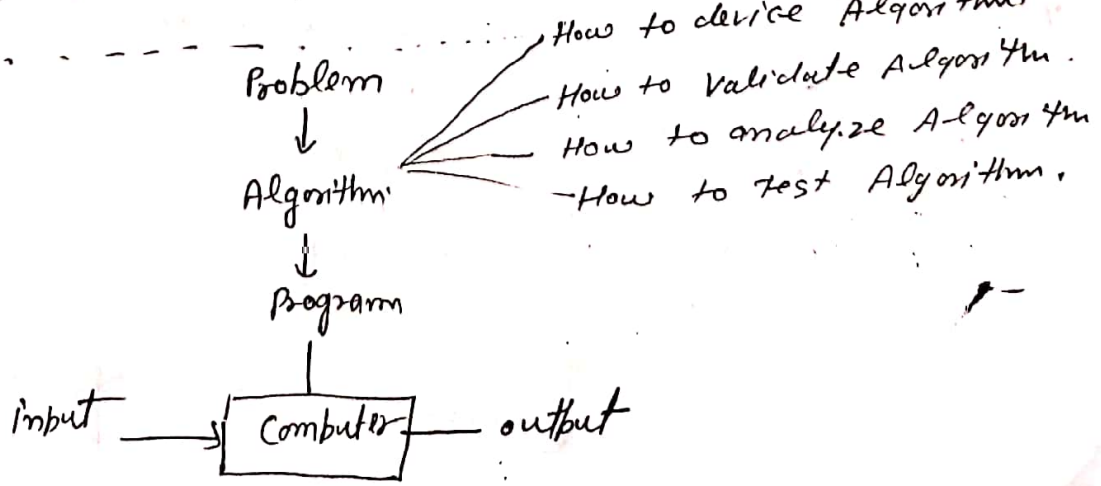
(i) Input: zero or more quantities are externally supplied.

(ii) Output: At least one quantity is produced.

(iii) Definiteness: Each Instruction is clear and unambiguous.
e.g. "5/0" is not permitted.

(IV) Finiteness: Algorithm terminates after finite number of steps for all test cases.

(V) Effectiveness: All operation to be performed must be sufficiently basic that they can be done exactly and in finite length.



Why Study Algorithms (Main Issue to Algorithm)

1. How to device Algorithm: Devide and Conquer, Greedy, Dynamic prog. Backtracking, Branch & Bound. etc.
2. How to validate Algorithm: Once an algorithm is planned it is necessary to show that it computes the correct answer for all legal input into a finite number of steps.
3. How to analyze Algorithm: Time and space complexity.
4. How to Test an Algorithm: Debugging and Profiling
   Performance measurment

Difference between Algorithm, Pseudocode, and Program.

Algorithm.	Pseudocode	Program.
Systematic logical approch to solve any problem. it can be expresed using Natural language.	It is simple version of Programing code in plain English which uses to write code before it implemented in spaitic P.L.	it is exact code written for problem in programming language.

e.g Linear Search:
1. start from left element of arr[] and one by one compare x with each element of arr[].
2. if x matches with an element return index.
3. if x doesn't match with any element return -1.
	e.g linear search.
	FUNCTION linearS(list, item)
	FOR index from 0 -> length(list)
	IF list[index] == item THEN
	RETURN index.
	END IF
	END LOOP
	RETURN -1
	END FUNCTION.
		e.g linear Search in C
		int search(int arr[], int n, int x)
		{
		int i;
		for(i=0; i<n; i++)
		if (arr[i]==x)
		return i;
		return -1;
		}.

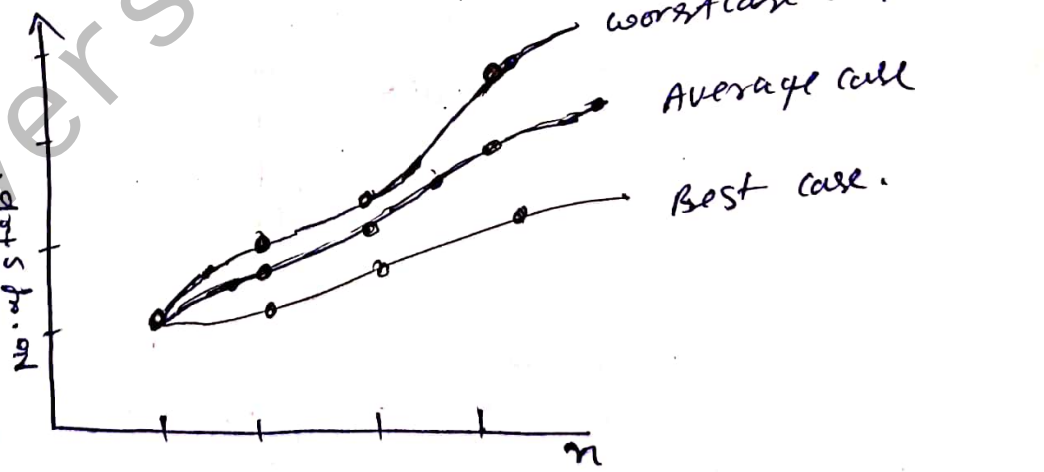
Analysis of Algorithm:

Analysis of an algorithm is requise to declute the correctness and measure the efficiency of Algorithm.

the efficiency of algorithm depends on two factor Amount of computer memory and time required for successful execution of the algorithm.

generally we perform following types of analysis.

1. Worst Case: maximum number of steps taken on any instance of size 'n'.
2. Best case: minimum number of step taken on any instance of size 'n'.
3. Average Case: An ~~avg~~ average number of steps taken on any instance of size 'n'.



example: Complexity of Bubble sort.

Best — Avg — worst.

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

Complexity of Algorithm:

1. Time Complexity: The time complexity of an algorithm is the amount of computer time it needs to run to completion.

Rules for Calculating time Complexity:-

1- Loop: for $i \leftarrow 1$ to n
$S \leftarrow S + A[i]$

computation time = $\sum_{i=1}^{n} (1) = \Theta(n)$

2. Nested Loop: for $i \leftarrow 1$ to m
for $j \leftarrow 1$ to m
$S \leftarrow S + A[i,j]$

$= \sum_{i=1}^{m} (m) = \Theta(m^2)$

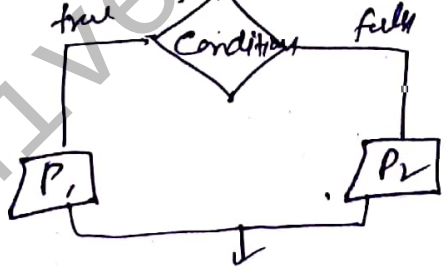
3. Consecutive Statement:

$t_1 = \Theta(n)$

$t_2 = \Theta(n^2)$

$t_1 + t_2 = \max(t_1, t_2)$
$= \Theta(n^2)$

4. if else.



$P_1 \rightarrow t_1 = \Theta(n)$

$P_2 \rightarrow t_2 = \Theta(n^2)$

$t = \max(t_1, t_2)$
$= \Theta(n^2)$

5. while Loops.

```
while (n>0)
{
  i ← i+1
  n ← n/2
}
```
$\Theta(\log n)$

```
while (n>0)
{
  i ← i+1
  n ← n-1
}
```
$\Theta(n)$

6- Recursion:

```
Factorial (n)
1. if n≤1
2     return 1
3 else
4  return n* factorial (n-1)
```

$T_c = O(n)$

```
Fib (n)
1 if n≤1
2     return 1
3 else
4  Return Fib(n-1) + Fib (n-2)
```

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

(Example (Solved))

```
Algo sum(a,n)                    — 0
{                                — 0
  s := 0.0                       — 1
  for i := 1 to n do             — n+1     No. of time execution.
    s = s + a[i]                 — n
  return s                       — 1
}                                — 0
                                 ------
                                 2n+3
```

$T_{sum} = O(n)$

```
Product (a[1...m, 1...n]), (b[1...n, 1....P])
1. for i ← 1 to m do                          — m+1
       for j ← 1 to P do                      — m(P+1)
           C[i,j] ← 0                         — mP
           for{ K ← 1 to n do                 — m.P(n+1)
               C[i,j] ← c[i,j] + a[i,k] * b[k,j]   — m.P.n
return C[1...m, 1....P]
```

~~$T(m,n,p) = O(mnp)$~~

$T(m,n,P) = O(m.n.P)$

Space Complexity:

Space complexity of an algorithm is the amount of memory needs to run to completion.

Space Complexity = Auxiliary space + Input space.

$S(P) = S_p + C$

S_p — instance characteristics.

e.g.

```
Algo abc(a, b, c)
{
  return a+b+b*c + (a+b-c)/a+b +4.0
}
```

$S(P) = 0 + 3$ (word)

eg.

```
Algo Sum(a, n)
{
  s := 0
  for i := 1 to n do
      s := s + a[i]
  return s;
}
```

$S_p = n + 3$

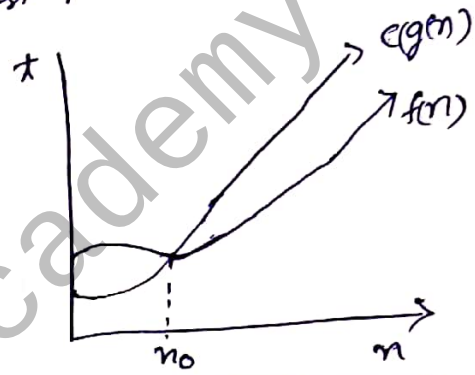
$a[\cdot]$, n, i, s

Asymptotic Notation:

Asymptotic Notation are mathmatical tool to represent complexity in term of notation for time and space. there are following asymptotic Notation used in Analysis of algo.

1. Big-oh (O) :- upper bound or worst case.

the function $f(n) = O(g(n))$ iff there exist positive constant c and n_0 such that $f(n) \leq cg(n)$ for all $n, n \geq n_0$.



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

$n \geq n_0$

$c > 0,\ n_0 \geq 1$

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

$f(n) = 3n+2 \qquad g(n) = n$

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

$f(n) \leq c\,g(n), \quad c > 0, \quad n_0 \geq 1$

$3n+2 \leq cn$

$c = 4$

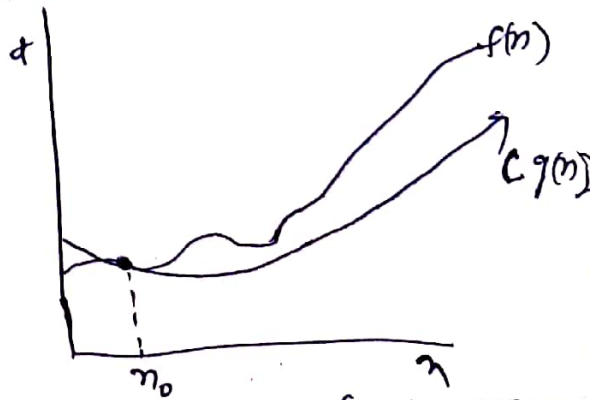
$3n+2 \leq 4n$

$n \geq 2$

$f(n) = 3n+2 \qquad g(n) = n$

$f(n) = O(g(n))$ ✓

2. Big Omega Ω : lower bound or best case.



$f(n) = 3n+2 \qquad g(n) = n$

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

$3n+2 \geq cn$

$c = 3$

$3n+2 \geq 3n$

$n \geq 1$

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

$n \geq n_0$

$c > 0 \quad n_0 \geq 1$

Average case or tight bound

Big theta Θ: the function $f(n) = \Theta(g(n))$ iff there exist positive constant c_1, c_2 and n_0 such that $c_1 g(n) \leq f(n) \leq c_2 g(n)$ for all n, $n \geq n_0$

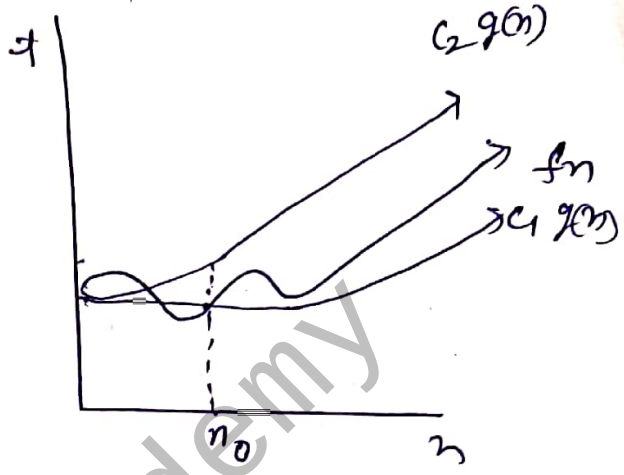
$f(n) = \Theta\, g(n)$

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

$c_1, c_2 > 0$

$n > n_0$

$n_0 \geq 1$



$f(n)$ $3n+2$ $\quad g(n) = n$

$f(n) \leq c g(n)$

$3n+2 \leq 4n.$ $\quad n \geq 1$

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

$3n+2 \geq 3n$ $\quad n \geq 1$

Recurrences when an algo contain a recursive call to itself if running time can be described by recurrence, equation. A Recurrence is an equation that describe a function in term of its value on smaller input.

To solve a recurrence relation means to obtain a function defined on the natural number that satisfy the recurrence:

eg. Recurrence Relation of MERGE SORT is

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

①

```
void fun(int n)        T(n)
{
  if (n>0)
  {
    print(n);   ---- 1
    fun(n-1)    ---- T(n-1)
  }
}
```

∴ $T(n) = T(n-1)+1$

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

2.

```
void A(n)
{
  if(n>0)
  {
    for(i=0; i<n; i++)
    {
      Print(n);
    }
    A(n-1);
  }
}
```

$T(n)$

1

$\rightarrow n+1$

$\rightarrow n$

$T(n-1)$

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

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

3. Fib (n)

```
if (n ≤ 1)
    return 1
else
    return Fib(n-1) + Fib(n-2)
```

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

$$= \begin{cases} 1 & \text{if } n \leq 1 \\ T(n-1) + T(n-2) + 2 & \text{if } n > 1 \end{cases}$$

There are mainly three way to solving Recurrences.

1- Substitution method
2- Recurrence Tree method
3- Master method.

1- Substitution method. (Back substitution method)

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

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

$T(n) = T(n-1) + 1$
$T(n-1) = T(n-2) + 1$
$T(n-2) = T(n-3) + 1$

substitute $T(n-1)$

$T(n) = T(n-2) + 1 + 1$
$T(n) = T(n-2) + 2$
$T(n) = T(n-3) + 3$
⋮ ⋮ ⋮ . k
$T(n) = T(n-k) + k$

Assumed $n - k = 0$, $n = k$

$T(n) = T(n-n) + n$
$= T(0) + n$
$= 1 + n = O(n)$.

2. $T(n) = \begin{cases} 1 & n=0 \\ T(n-1)+n & n>0 \end{cases}$

$T(n) = T(n-1) + n \quad \text{---(i)}$

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

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

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

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

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

$= [T(n-3) + (n-2)] + (n-1) + n \quad \text{---(ii)}$

$= T(n-3) + (n-2) + (n-1) + n \quad \text{---(iii)}$

$\vdots \qquad \vdots \qquad \vdots \quad k$

$= T(n-K) + (n-(k-1)) + (n-(k-2)) + \cdots (n-1) + n$

Assume $n-k=0$

$n=k$

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

$= T(0) + 1 + 2 + \cdots (n-1) + n$

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

3

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

$T(n)$

```
void fun(n)
{
  if (n>1)
  {
    for i ← 0 to n do
        statement;
    T(n/2)  fun(n/2);
    T(n/2)  fun(n/2);
  }
}
```

n

$$T(n) = 2T(n/2)+n \quad \text{---(1)}$$

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

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

$$T(n/2^2) = 2T(\frac{n}{2^3})+\frac{n}{2^2}$$

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

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

$$= 2^2\left[2T\left(\frac{n}{2^3}\right)+\frac{n}{2^2}\right]+2n \quad \text{---(2)}$$

$$= 2^3 T\left(\frac{n}{2^3}\right)+3n \quad \text{---(3)}$$

$$\vdots \qquad \vdots \qquad 1<$$

$$= 2^k T\left(\frac{n}{2^k}\right)+kn$$

Assume: $T\left(\frac{n}{2^k}\right) = T(1)$

$\frac{n}{2^k} = 1 \quad n = 2^k$

$k = \log n$

$$T(n) = 2^k T(1) + kn$$

$$= n \cdot 1 + kn$$

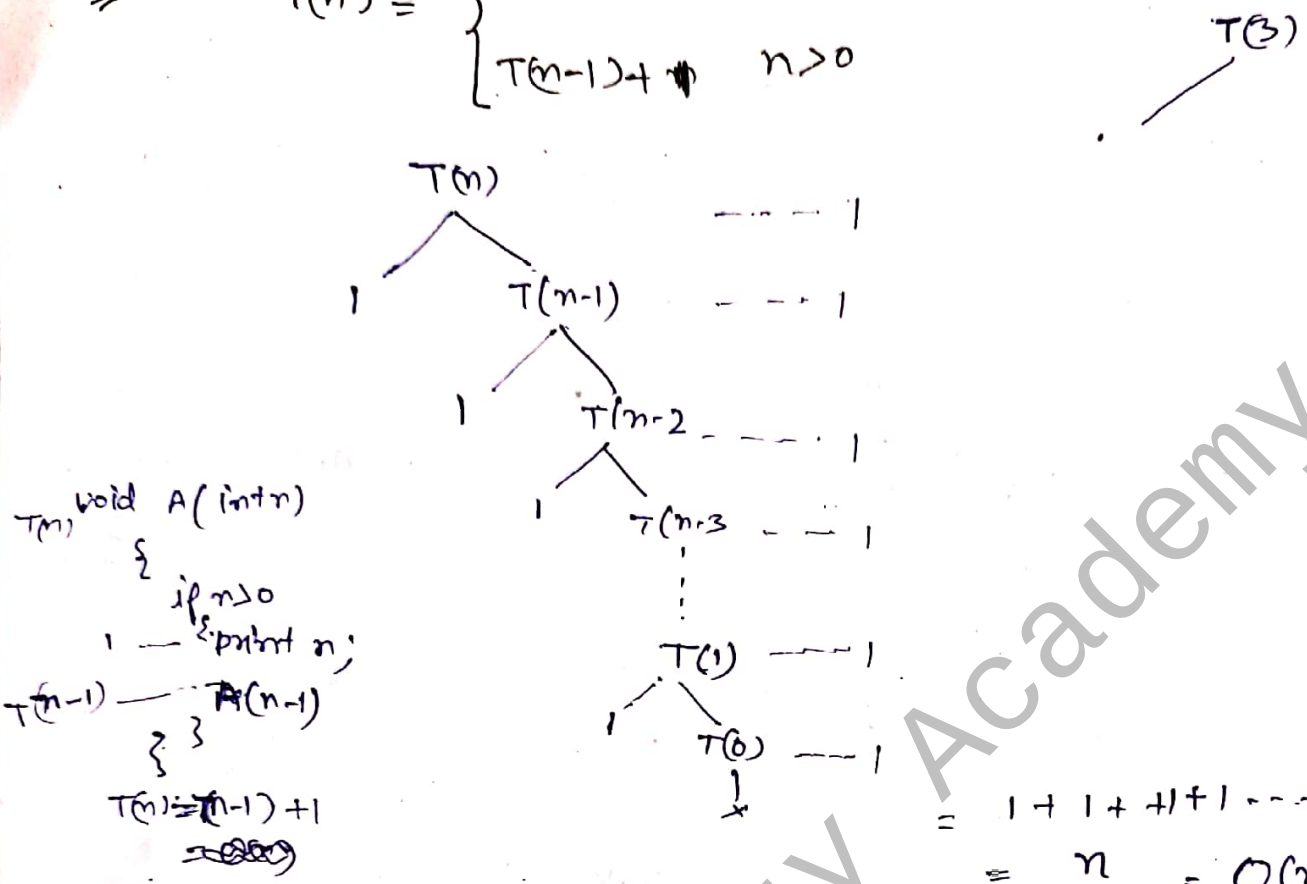
$$= n + n\log n$$

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

Recurrence Tree method.

1-

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

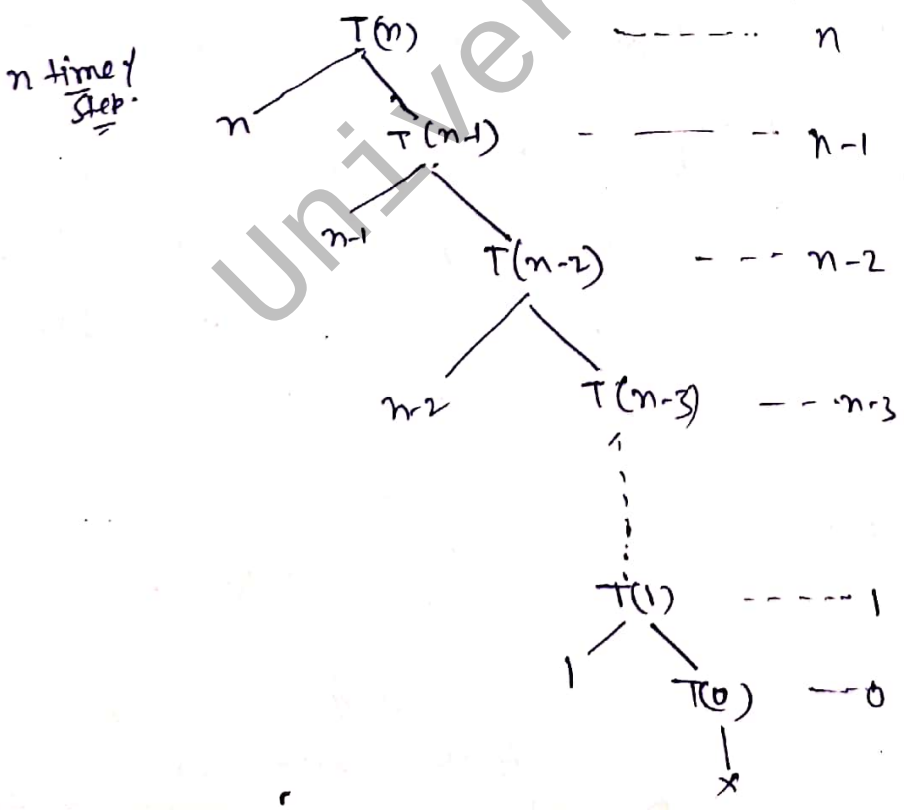


$= 1 + 1 + 1 + 1 \cdots n+1$

$= n = O(n)$.

II

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

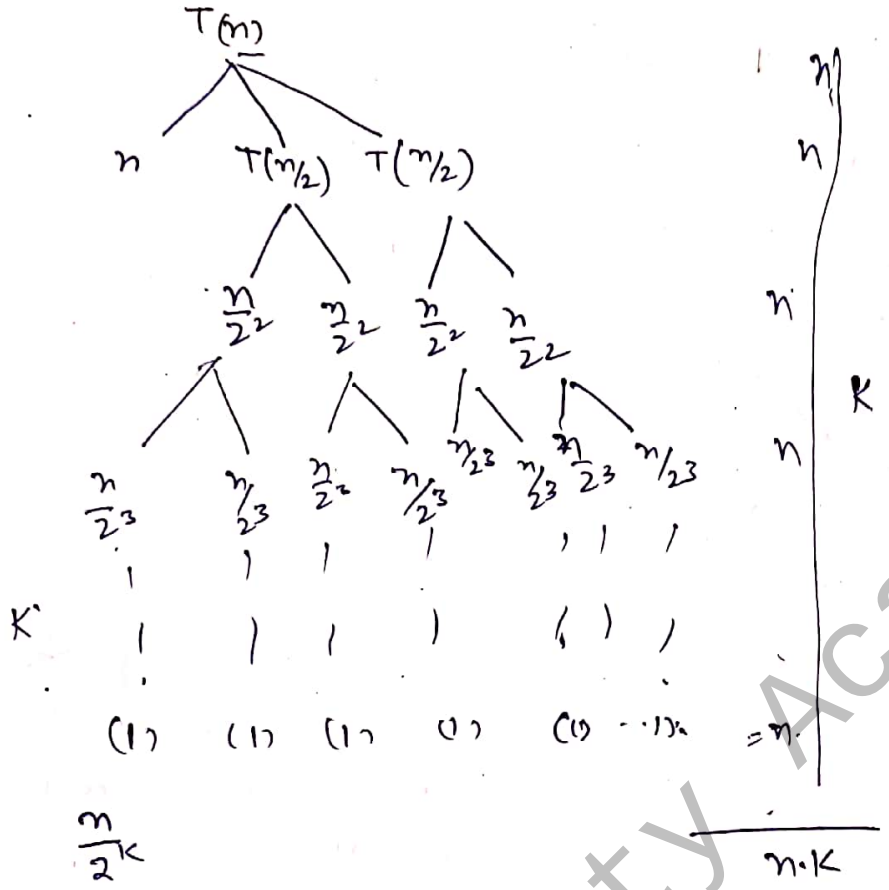


$0+1+2+3 \cdots n-1+n = \frac{n(n-1)}{2}$

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

3)

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



Assume $\frac{n}{2^K} = 1$

$n = 2^K$

$K = \log n$

$n \cdot K = n \log n$

$= O(n \log n)$

$=$

Master Method:

The master method for solving recurrences of the form $T(n) = aT(n/b) + f(n)$ where

$a \geq 1$, $b > 1$ are constant

$f(n)$ is asymptotically positive function.

$\frac{n}{b}$ means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$

$T(n)$ can be bounded asymptotically as follows.

1. if $f(n) = O(n^{\log_b a - \varepsilon})$ for some constant $\varepsilon > 0$ then $T(n) = \Theta(n^{\log_b a})$
2. if $f(n) = \Theta(n^{\log_b a})$ then $T(n) = \Theta(n^{\log_b a} \log n)$
3. if $f(n) = \Omega(n^{\log_b a + \varepsilon})$ for some constant $\varepsilon > 0$ and if $a f(n/b) \leq c f(n)$ for some constant $c < 1$ and all sufficiently large n, then $T(n) = \Theta(f(n))$

Q. 1:- $T(n) = 9T(n/3) + n$

$a = 9, \quad b = 3 \quad f(n) = n$

$n^{\log_b 9} = n^{\log_3 9} = \Theta(n^2)$

$f(n) < n^{\log_b 9}$

since $f(n) = O(n^{\log_3 9 - p})$ where $p = 1$

we can apply case 1 so selection is

$T(n) = \Theta(n^2)$

(2) $T(n) = T(2n/3) + 1$

$a = 1 \quad b = 3 \quad f(n) = 1$

$n^{\log_b 9} = n^{\log_3 1} = n^0 = 1$

$f(n) = n^{\log_b 9}$

~~since~~ $f(n) = \Theta(n^{\log_b 9}) = \Theta(1)$.

we can apply case 2:

$T(n) = n^{\log_b 9} \log n$

$= 1 \cdot \log n = \Theta(\log n)$

3:- $T(n) = 3T(n/4) + n \log n$

$a = 3, \quad b = 4 \quad f(n) = n \log n$

$n^{\log_b 9} = n^{\log_4 3} = O(n^{0.793})$.

$f(n) > n^{\log_b 9}$

since $f(n) = \Omega(n^{\log_4 3 + p}) = n \log n$

when $p = 0.2$

Case 3 Applied if we can show that

$a f(n/b) \leq c f(n)$ for $c < 1$

$3\left(\frac{n}{4}\right) \log\left(\frac{n}{4}\right) \leq \frac{3}{4} n \log n$ for $c = 3/4$

hence case 3 $T(n) = \Theta(n \log n)$

Q. Show that solution to the Binary Search Recurrence $T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$ is $T(n) = \Theta(\log n)$ by using master method.

Q.

Q Solved the following Recurrence by master method.

(i) $T(n) = 2T\left(\frac{n}{2}\right) + n^3$ (Case 3)

(ii) $T(n) = T\left(\frac{9n}{10}\right) + n$ (Case 3)

(iii) $T(n) = 16T\left(\frac{n}{4}\right) + n^2$ (Case 2)

Q. Consider the following recurrence ~~tree~~ using recursion tree method

$$T(n) = T\left(\frac{n}{3}\right) + T\left(\frac{2n}{3}\right) + n$$

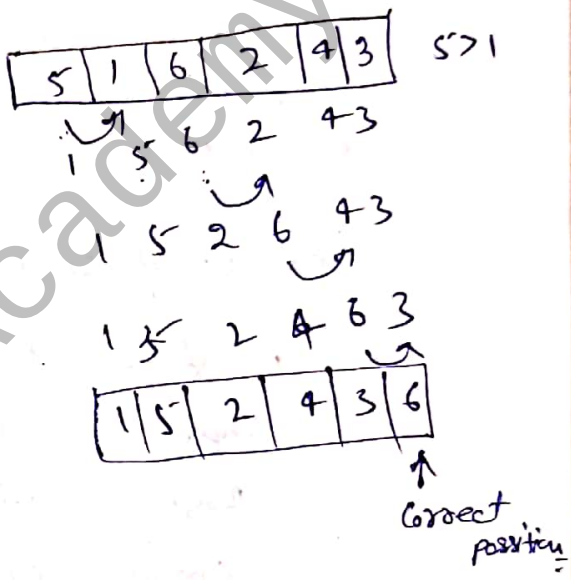
Sorting Algorithms:

1. Bubble Sort
2. Selection Sort
3. Insertion Sort
4. Shell Sort
5. Quick Sort
6. Merge Sort
7. Heap Sort

1- Bubble sort :→

input: a non-sorted list of n numbers $a, a_2 \cdots a_n$

```
for i ← n to 1
    for J : 1 to i-1
        if a_J > a_J+1 then
            { //swap
            t ← a_J
            a_J ← a_J+1
            a_J+1 ← t
            }
```



Complexity

worst case : $(n-1) + (n-2) + (n-3) \cdots \cdots 3+2+1 \cdots$

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

Average case : $O(n^2)$

Best case : when the list is already sorted.

$O(n)$

space complexity : $O(1)$ because only single additional space is required for temporary variable.

2. Selection Sort:

```
function selection-sort(A,n)
    for i=1 to n-1 do
        min ← i
        for J = i+1 to n do
            if A[J] < A[min] then
                min ← J
            end if
        end for
        swap(A[i], A[min])
    end for
end function.
```

n=5

0	1	2	3	4
31	25	12	22	64

```
for(i=1 to ≤n-1 i++)
{
    min = i;
    for(J=i+1; J≤n; J++)
    {
        if(a[J] < a[min])
        {
            min = J;
        }
    }
    if(min != i)
    {
        swap(a[i], a[min])
    }
}
```

Analysis of selection sort.

Worst case = $O(n^2)$

Avg = $O(n^2)$

Best = $O(n^2)$

Space = $O(1)$

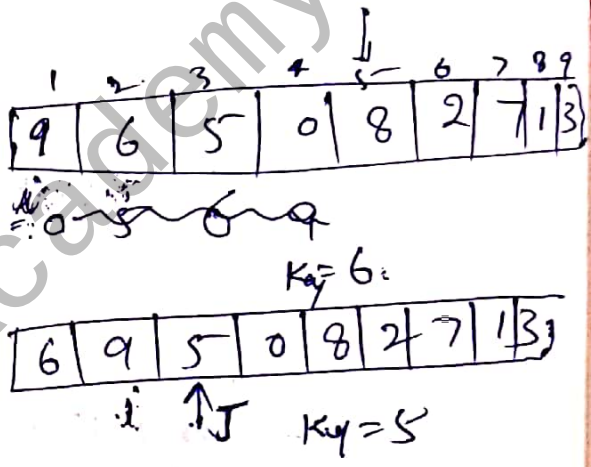
---

the selection sort algorithm sorts an array repeatedly finding minimum element from unsorted part and putting it at the beginning

64 25 12 22 11

11 25 64 22 64

```
Insertion-Sort(A)
{
  for J=2 to A.length
     key=A[J]  //insert A[J] into sorted sequence :A[1.... J-1]
     i=J-1
     while( i>0 and A[i]>key)
          A[i+1]=A[i]
          i=i-1
     A[i+1]=key
}
```



i = 3-1 = 2

Analysis:

	comparison + movement	5 6 9
J=2	1+1 = 2	2(1)
J=3	2+2 = 4	2(2)
J=4	3+3 = 6	2(3)
J=5 →	4+4 = 8	2(5-1) = 2(4)
n	(n-1)+(n-1) = 2(n-1)	

2(1) + 2(2) + ... 2(n-1)

insertion Algorithm = 2(1 + 2 + ... n-1) = $\frac{2(n-1)(n)}{2}$ = $O(n^2)$

worst case.

Finding = Comparison

	comparison	movement
Binary (loop)	n.	
Recursive	n	1

(n-1) times. $O(n^2)$

Best case = Already sorted list.

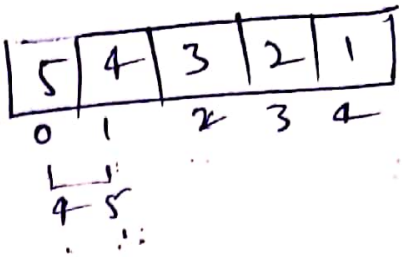
	1	2	3	4	5
Comp:	1	1	1	1	1

No movement.

$\Omega(n-1) = \Omega(n)$

Space: key, i, J — $O(1)$

n=4




```
for (i=0 ; i<n-1 ; i++)        n-1-i
{                               0<4
                                1
   for (j=0 ; j<n-1 ; j++)
   {                    0<4
      if (a[j] > a[j+1])
      {
         temp = a[j]
         a[j] = a[j+1]
         a[j+1] = temp
      }
   }
}
```

i=0 1st elemt

i=1 2nd elem

i=2 3rd elemt

i=3 4th elem

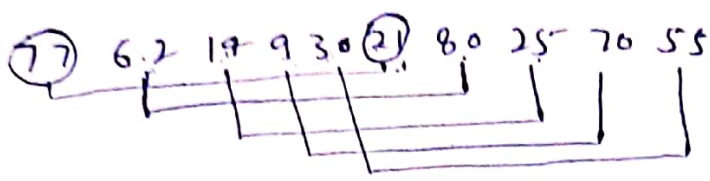
```
for (i=0 ; i<n-1 ; i++)
{
   min = i;
   for (j=i+1 ; j<n ; j++)
   {
      if (a[j] < a[min])
      {
         min=j;
      }
   }
   temp = a[i];
   a[i] = a[min]
   a[min] = temp
}
```

Shell Sort

77	62	14	9	30	21	80	25	70	55

N=10

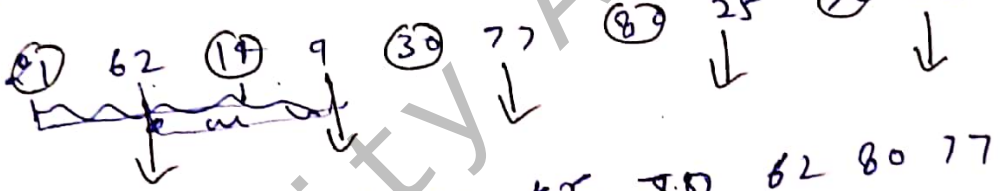
gap (Pass1) = floor(N/2)
= floor(10/2)

Pass1 

temp = 21

21 62 14 9 30 77 80 25 70 55

Pass2 gap = floor(5/2) = 2



14 9 21 25 30 55 70 62 80 77

Pass 3: gap = floor(2/2) = 1

9 14 21 25 30 55 62 70 77 80

```
Shellsort (int arr[], int n)
{
    for (gap = [n/2] ; gap > 0 ; gap = gap/2)
    {
        for ( i = gap ; i < n ; i++)
        {
            temp = arr[i]
            for ( j = i ; j >= gap && arr[j-gap] > temp ; j = j-gap )
            {
                arr[j] = arr[j-gap] ;
            }
            arr[j] = temp;
        }
    }
}
```

Best case = nlogn
Worst case = $n \log^2 n$
Avg = $n \log^2 n$
Best (n) to $(n)^2$
Space = O(1)

0	1	2	3	4	5	6	7	8	9	10	11	12
25	30	10	90	72	77	87	91	67	85	38	47	56

i = 7

temp = 91

J = 7

i = 10

temp = 38

n = 13

gap = n/2 = 6

i = gap = 6

temp = arr[i] = 87

J = 6

A[6] = temp = 87

25 30 10 70 38 47 87 91 67 85 72 77

i = 12

tem = 56

J = 12

56

87

0	1	2	3	4
12	34	54	2	3

Pass 1

i = gap = 2

temp = 54

J = 2

A[2] = 54

i = 4

temp = 3

J = 4

n = 5

gap = 2

i = 3

temp = 2

J = 3

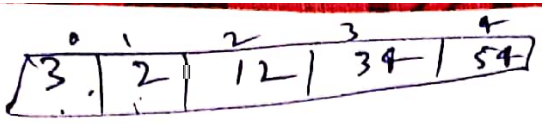
J = 1

0	1	2	3	4
12	2	54	34	3

12 2 3 34 54

3	2	12	34	54

Pass 2



Ans:

gap = gap/2 = 1|1

i=1

temp: 2

j=1

2 3 12 34 54

i=2

temp = 12

j=2 2 3 12 34 54

i=3

temp = 34

Quick Sort

Quicksort is based on divide and conquer paradigm there are three steps divide-and-conquer process for sorting a array $A[p \ldots r]$.

1. **Divide**: Partition the array $A[p \ldots r]$ into two subarray $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ such that

$$A[p \ldots q-1] \leq A[q] < A[q+1 \ldots r].$$

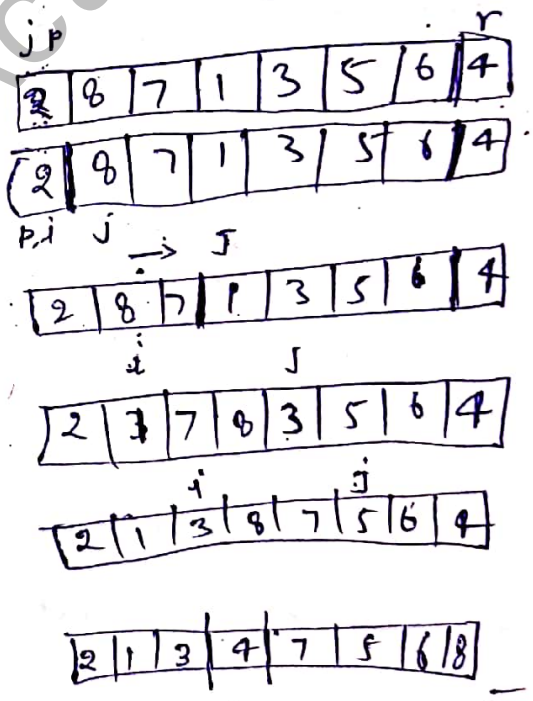
2. **Conquer**: Sort the two sub subarray $A[p \ldots q-1]$ and $A[q+1 \ldots r]$ by Recursive call.

3. **Combine**: ~~Bec~~ Entire Array $A[p \ldots r]$ is now sorted.

Algo

```
Quicksort (A, p, r)
1: if p<r
2.      q= Partition(A, p, r)
3       Quicksort (A, p, q-1)
4       Quicksort(A, q+1, r)

Partition(A, p, r)
1.  x = A[r]
2   i = p-1
3.  for j = p to r-1
4       if A[j] ≤ x
5           i = i+1
6           exchange A[i] with A[j]
7   exchange A[i+1] with A[r]
9.  return i+1
```



Best case: $T(n) = T(n/2) + T(n/2) + n$

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

by master method $= O(n \log n)$

Analysis:

Time = Partition cost + recursive call.

worst case $T(n) = T(n-1) + T(0) + \Theta(n)$

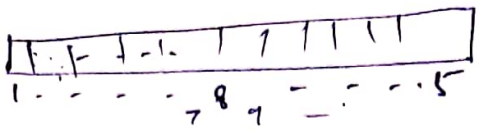
$= T(n-1) + 1 + \Theta(n)$

$= T(n-1) + n$

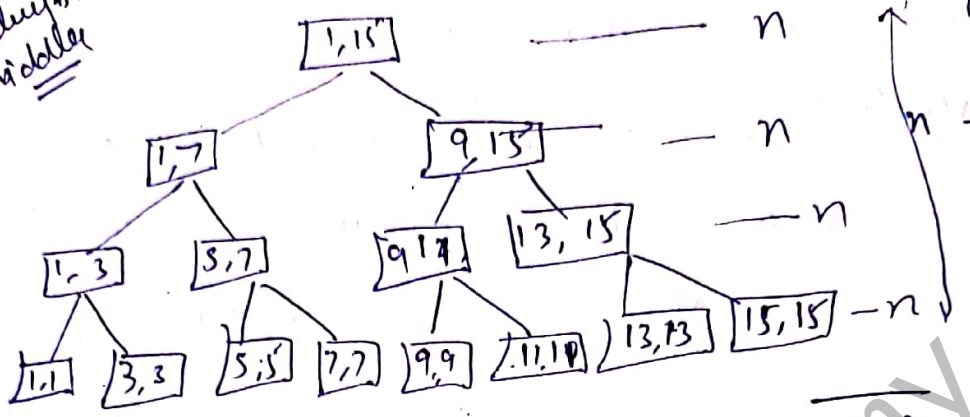
by substitution method

$= O(n^2)$

Best case



If partition always in middle



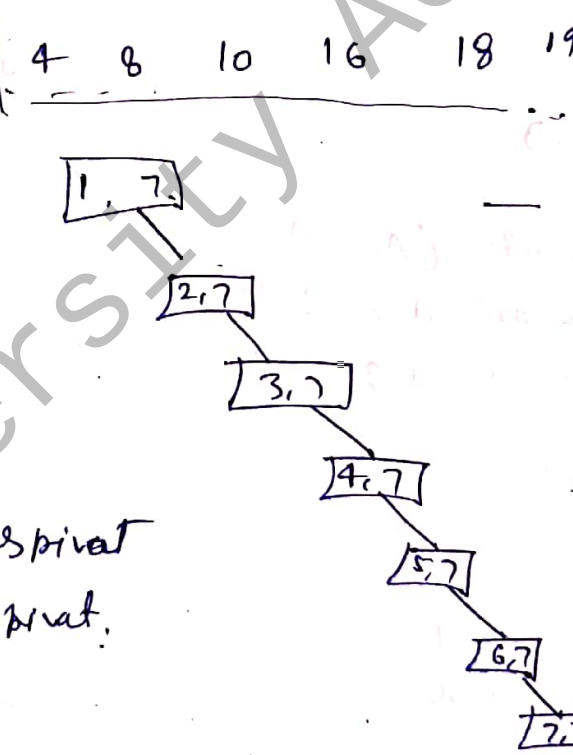

```
Quicksort(A,p,r)
if (p<r)
  q = Partition(A,p,r)
  Quicksort(A,p,q-1)
  Quicksort(A,q+1,r)
}
```

$\frac{n}{2^k} = 1 \quad k = \log n$

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

Best case

Worst case



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

① Select middle element as pivot

② Select random element as pivot.

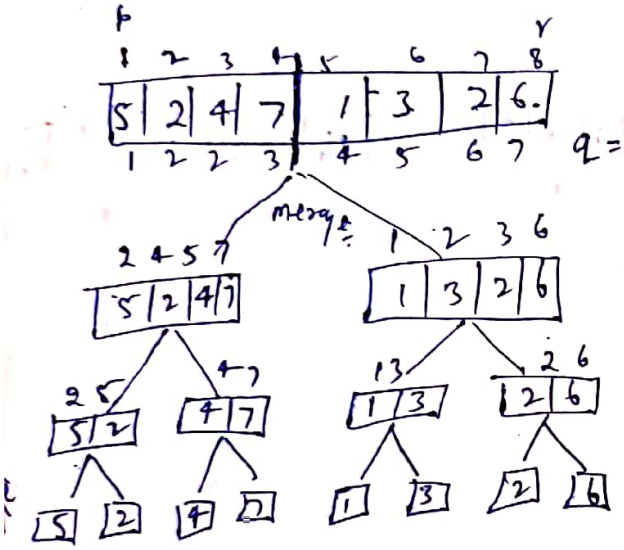
Space:

Stack: Best case: $O(\log n)$

worst case: $O(n)$

Merge Sort

```
MERGE-SORT(A, p, r)
if (p < r)
    q = ⌊(p+r)/2⌋        // mid
    MERGE-SORT(A, p, q)
    MERGE-SORT(A, q+1, r)
    MERG(A, p, q, r)
```



$\frac{n}{2^k} = 1$, $n = 2^k$, for log n, $k = \log n$

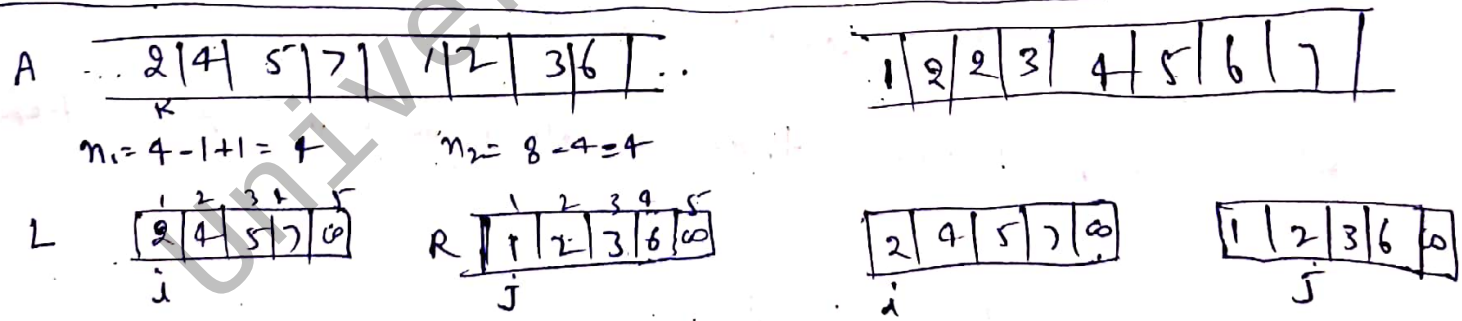
```
MERGE(A, p, q, r)
1  n1 = q - p + 1
2  n2 = r - q
3  let L[1 ... n1+1] and R[1 ... n2+1]
4  for i = 1 to n1
5      L[i] = A[p + i - 1]
6  for j = 1 to n2
7      R[j] = A[q + j]
8  L[n1+1] = ∞
9  R[n2+1] = ∞
10 i = 1
11 j = 1
12 for k = p to r
13     if L[i] ≤ R[j]
14         A[k] = L[i]
15         i = i + 1
16     else A[k] = R[j]
17         j = j + 1
```

... n elements merged

... n elements merge

--- n elements merge

kn



i

J

J

i

J

i

i

Analysis

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

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

Space complexity: $O(n)$

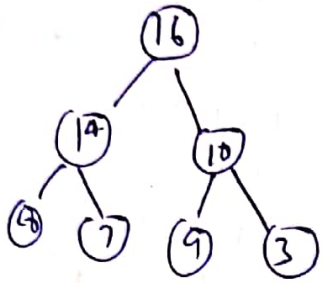
each node at most two children.

Heap Sort =

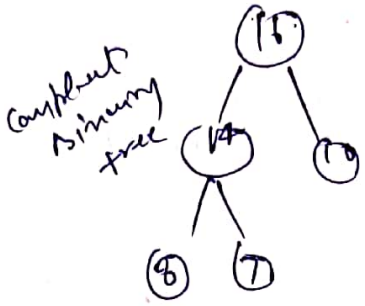
Binary tree and its Array Representation
- Full Binary tree + Complete binary tree
- not a complete Binary tree

Array Representation of BT.

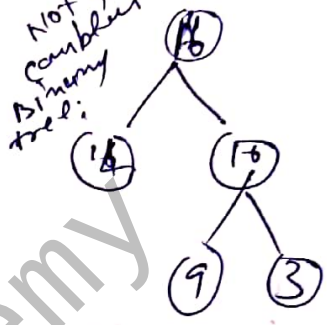
Full Binary tree →



Complete Binary tree



Not a complete Binary tree:



16	14	10	8	7	9	3
1	2	3	4	5	6	7

10	14	10	8	7
1	2	3	4	5

16	14	10	-	-	9	3
1	2	3	4	5	6	7

if a node at i
left child at 2i
Right child at 2i+1
Parent of i at $\lfloor i/2 \rfloor$

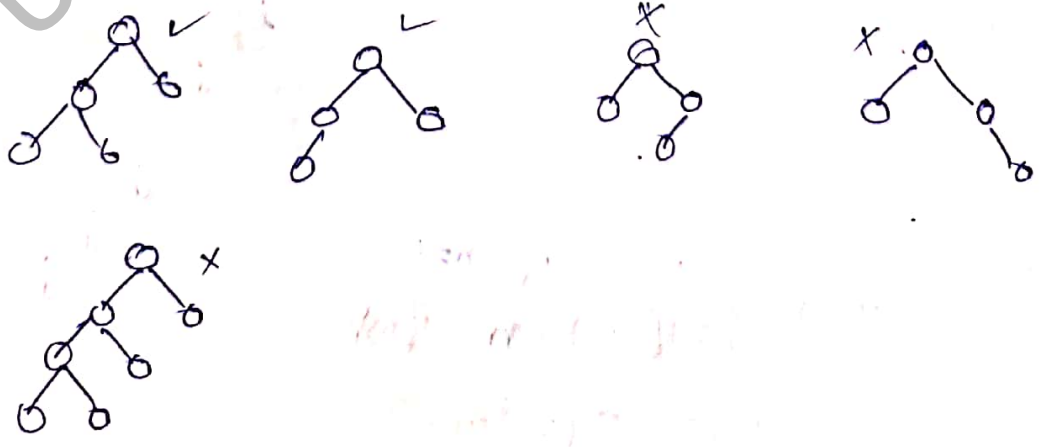
Binary tree

If height h the $2^{h+1}-1$ element in full Binary tree:

height h=0,1,2...

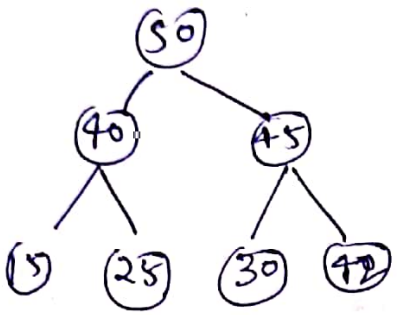
Complete Binary tree — No Blank in Array Representation

eg.

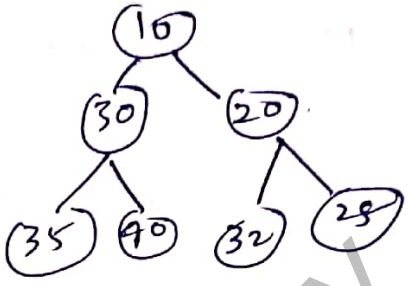


Heap → Heap is complete Binary tree. where item are stored in a special order such that value in a parent node is greater or smaller than its two child node

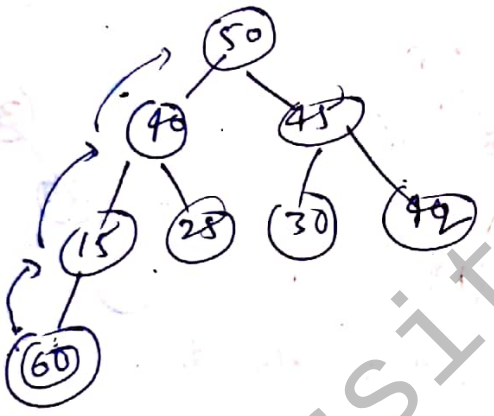
max heap

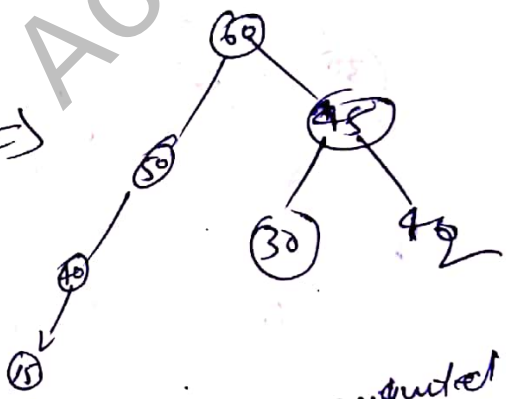


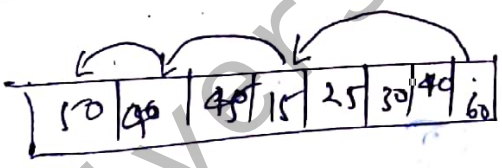
min heap



Insert 60

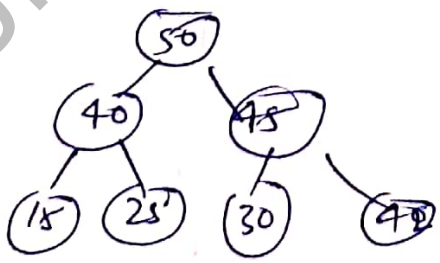


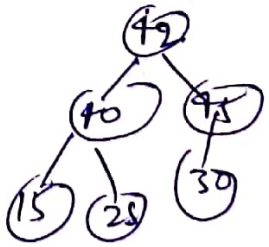


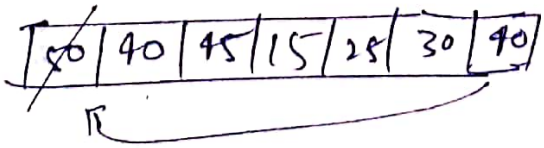


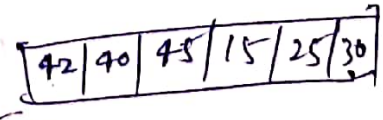
time required to Adjust heap **Same** **log n**

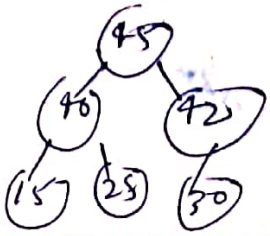
Delete 50 Root always will be deleted











log n

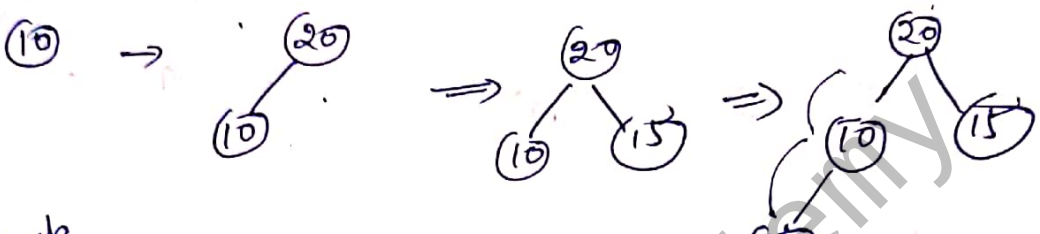
Heap Sort

max heap

Create heap (insert)

delete all elements from heap

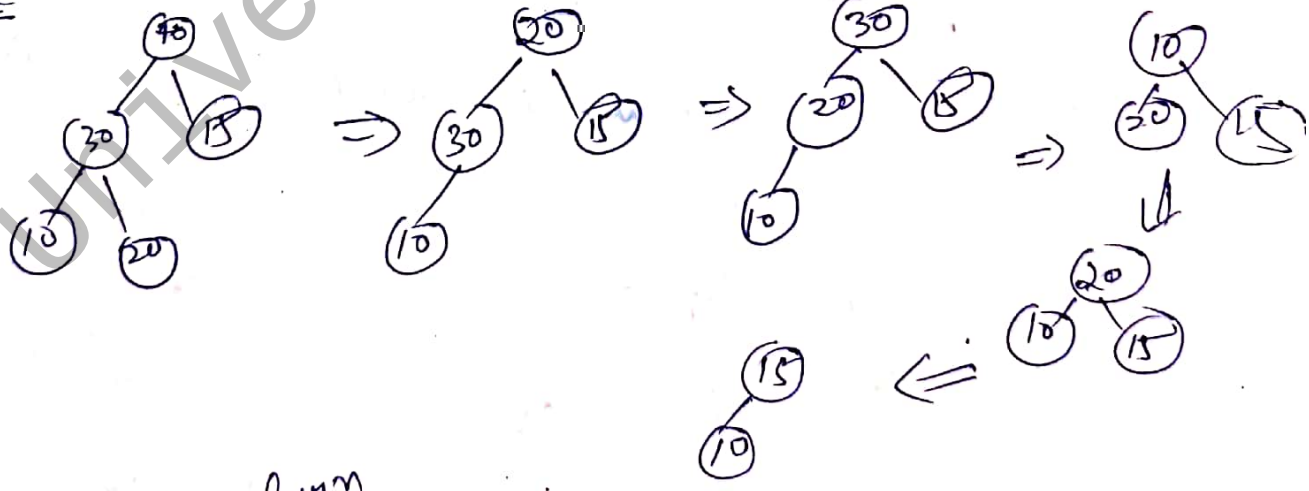
10, 20, 15, 30, 40



max heap

time: nlogn to insert an element:

delete.



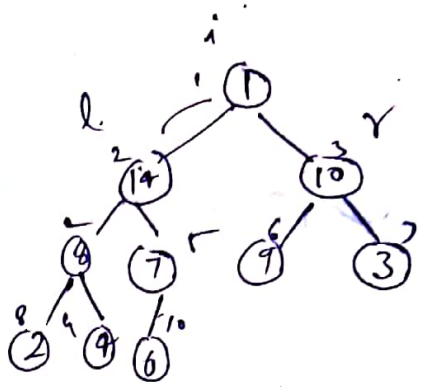
time = nlogn

TC = O(nlogn)

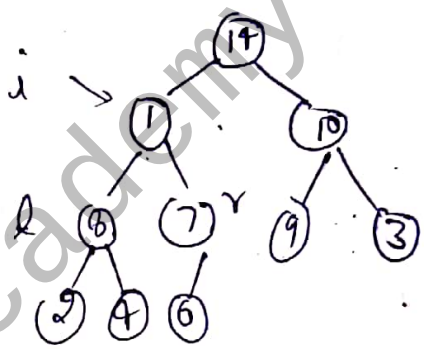
10, 15, 20, 30, 40

Heap Sort Algorithm:

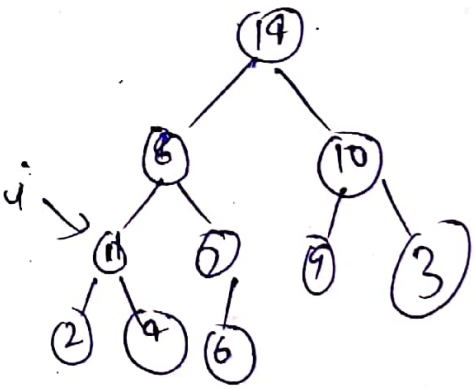
A=[1, 14, 10, 8, 7, 9, 3, 2, 4, 6]



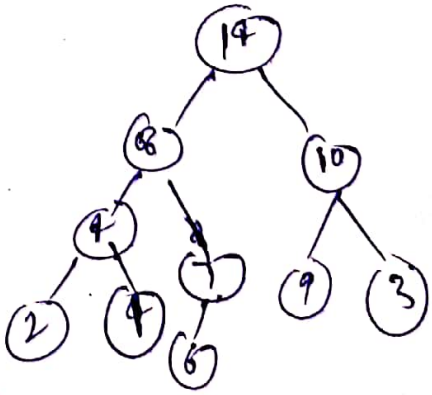
```
Heapsort(A)
1. Build-Max-Heap(A)                    — n.
2  for i = A.length down to 2           → n
3      exchange A[1] with A[i]
4      A.heap-size = A.heap-size - 1
5      Max-Heapify(A, 1)                → log n.
```



```
Build Max-Heap(A)
1. A.heap size = A.length
2. for i = ⌊A.length/2⌋ down to 1 — n
3      Max-Heapify(A, i)                n log n
```



```
Max-Heapify(A, i)
1  l = left(i)
2  r = Right(i)
3  if l ≤ A.heap-size and A[l] > A[i]
4      largest = l
5  else largest = i
6  if r ≤ A.heap-size and A[r] > A[largest]
7      largest = r
8  if largest ≠ i
9      exchange A[i] with A[largest]
10     Max heapify(A, largest)
```

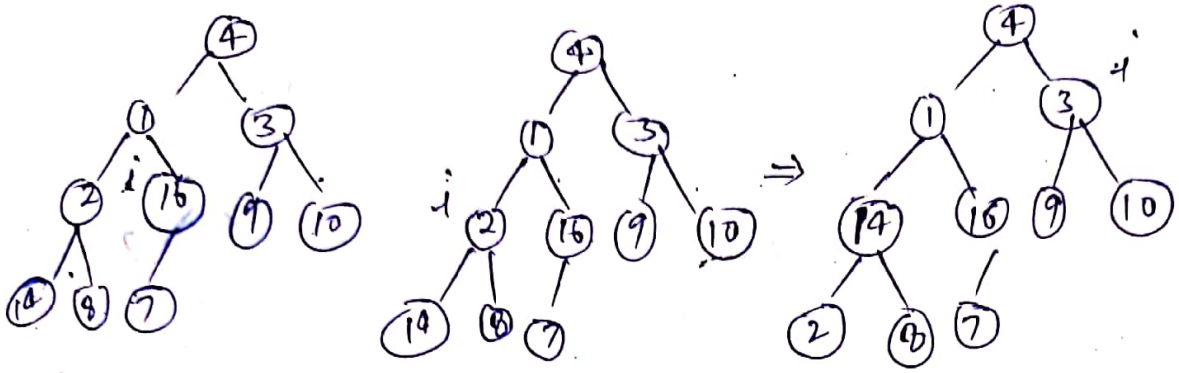



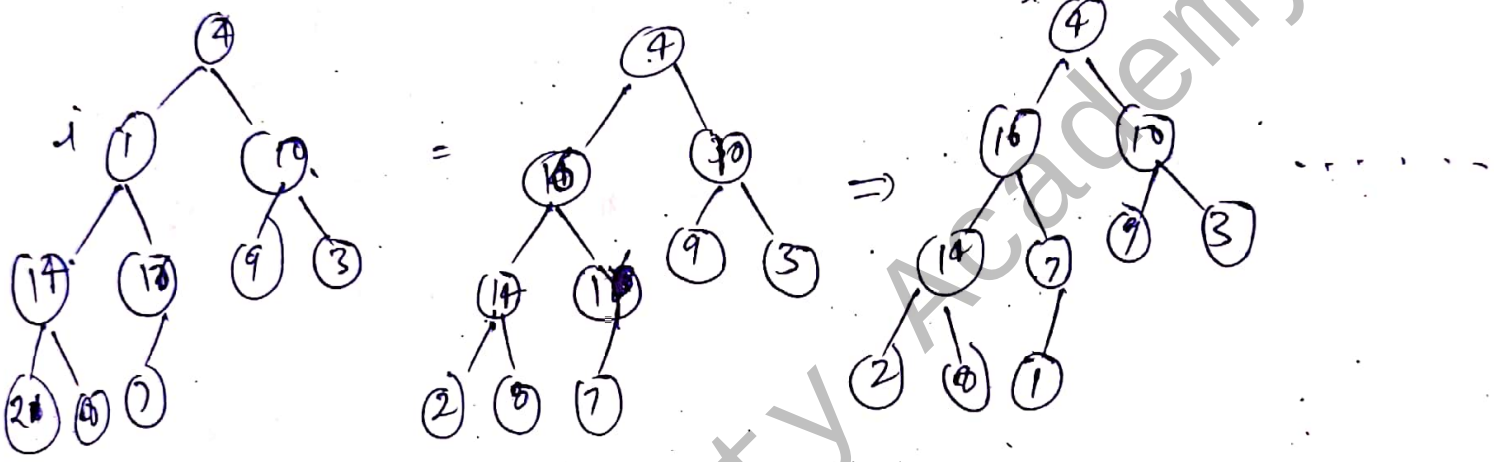
Time Complexity = O(n log n)

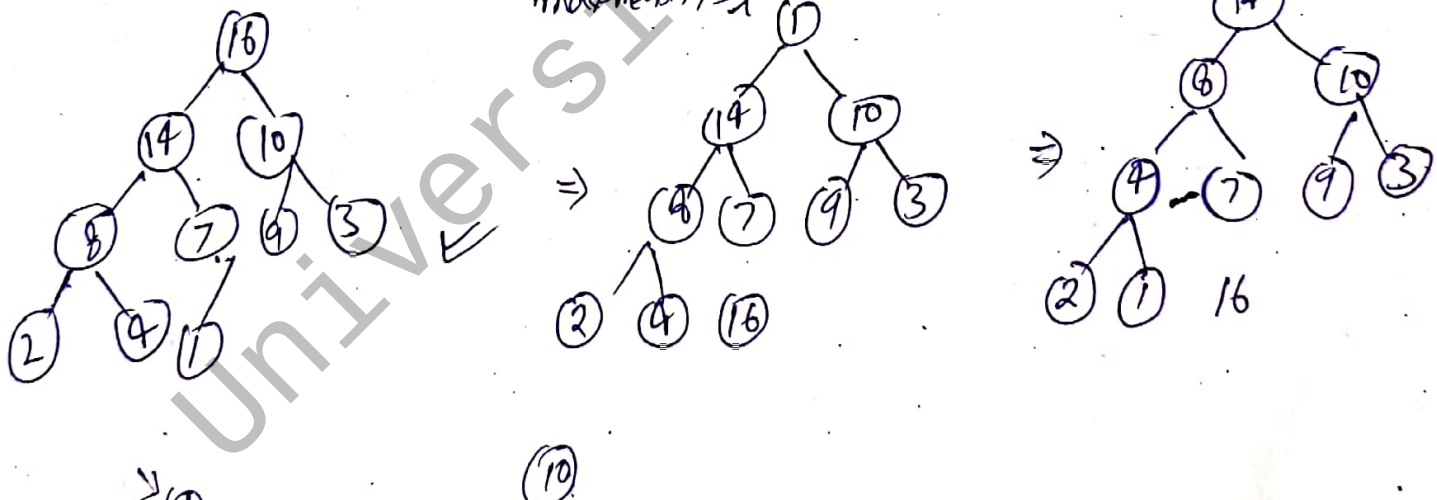
Space Complexity = O(1).

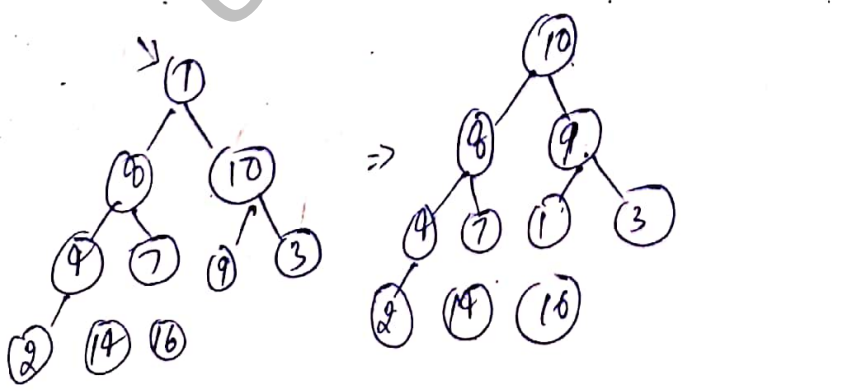
Should call Recursively
stack: log n
SP = O(log n)

0	1	2	3	4	5	6	7	8	9
4	1	3	2	16	9	10	14	8	7









Sorting in linear time.

Counting Sort

Counting Sort Assume that each of n input element is an integer in the range 0 to k for some k. the counting sort run in $\Theta(n)$ time.

```
Counting-Sort (A, B, k)
1- for i ← 0 to k
2      do C[i] ← 0
3  for j ← 1 to length [A].
4      do C[A[j]] ← C[A[j]] + 1
5  for i ← 1 to k
6      do C[i] ← C[i] + C[i-1]
7  for j ← length[A] down to 1
8      do B[C[A[j]]] ← A[j]
9         C[A[j]] ← C[A[j]] - 1
```

A

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

max = 5, min = 0

k = 5, range 0 - 5

Algo line 1-2

C

0	1	2	3	4	5
0	0	0	0	0	0

C

0	1	2	3	4	5
1	1	1	1	1	

line 3-4

C

0	1	2	3	4	5
2	0	2	3	0	1

line 5-6 sum

C

0	1	2	3	4	5
2	2	4	7	7	8

line 7-9.

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

Example

1	2	3	4	5	6
10	7	12	4	9	13

min = 4, max

1<=13

range 4 - 13

C

4	5	6	7	8	9	10	11	12	13
1	0	0	1	0	1	1	0	1	1

sum C

4	5	6	7	8	9	10	11	12	13
1	1	1	2	2	3	4	4	5	6

B

1	2	3	4	5	6
4	7	9	10	12	13

Radix Sort

Radix sort is a non-comparison sorting algorithm. Radix is not limited to number it can sort Alphabets (word) also.

input

329
457
657
839
436
720
355

⇒ 720, 355, 436, 457, 657, 329, 839

⇒ 720, 329, 436, 839, 355, 457, 657

⇒ 329, 355, 436, 457, 657, 720, 839 ✓

Sort the english word using Radix sort.

COW, DOG, SEA, RUG, ROW, MOB, BOX, TAB, BAR, EAR
TAR, DIG BIG, TEA, NOW, FOX.

Example 170, 45, 75, 90, 802, 24, 2, 66

make all number of three digit

170, 045, 075, 090, 802, 024, 002, 066
170, 090, 802, 002, 024, 045, 075, 066
802, 002, 024, 045, 066, 170, 075, 090

0	170, 090	802, 002	002, 024, 045, 066, 075 090)
1	~~090~~		170
2	802, 002	024	
3			~~002 090~~
4	024	045	002, 024, 045, 066, 075, 090, 170 802
5	045, 075		✓
6	066	066	
7		170, 075	time complexity.
8			O(d(n+k)
9		090	802

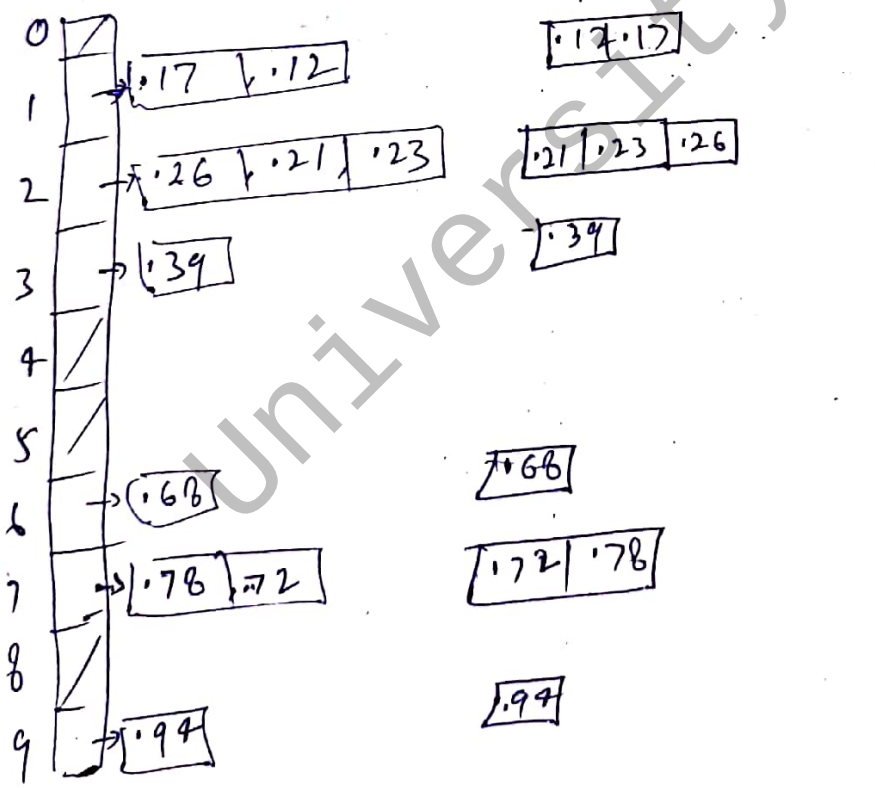
k possible value
number - 0-9
word A-Z (26)

Bucket Sort

Bucket sort algorithm that works by distributing the element of an array into a number of bucket. each bucket then sort individually by different sorting algo.. It is based on comparison based.

```
Bucket-sort(A)
1 n ← length(A)
2 for i ← 1 to n
3     do insert A[i] into list B[⌊n A[i]⌋]
4 for i ← 0 to n-1
5     do sort list B[i] with insertion sort.
6 concatenate the list B[0] B[1] ··· B[n-1].
```

1	2	3	4	5	6	7	8	9	10
.78	.17	.39	.26	.72	.94	.21	.12	.23	.68



Result array

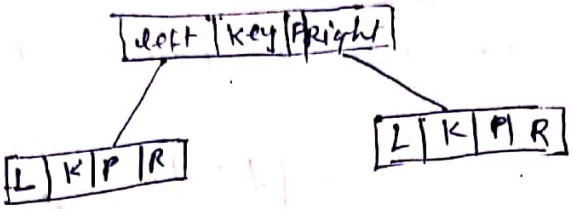
.12, .17, .21, .23, .26, .39, .68, .72, .78, .94.

Time Complexity: $O(n)$

RED-BLACK TREE

Background: Binary search tree:

Each Node contain field, key, left, Right, and P,



if x is a node in BST. then.

(i) Left subtree(y) of x has less or equal.

$$key[y] \leq key[x]$$

(ii) Right subtree(y) of x has greater or equal value.

$$key[y] \geq key[x]$$

e.g.

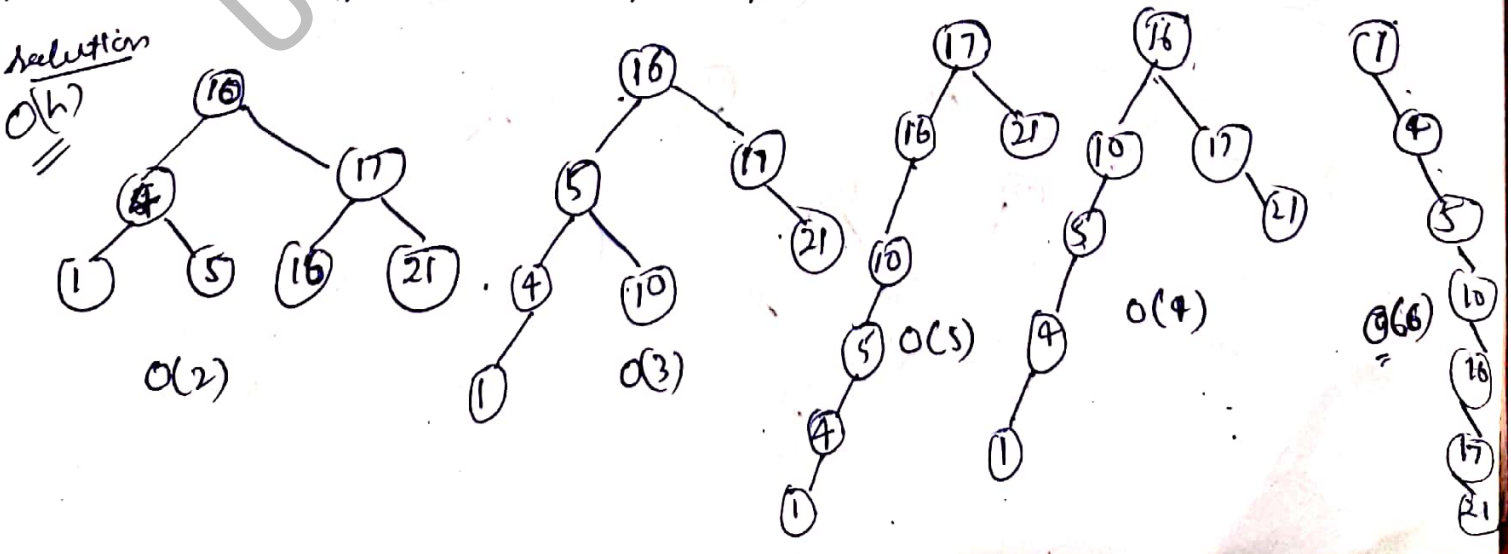
(a) BST of height 2

(b) BST of height 4

Q: For the set of key { 1, 4, 5, 10, 16, 17, 21} draw binary search tree of height 2, 3, 4, 5, 6.

Solution

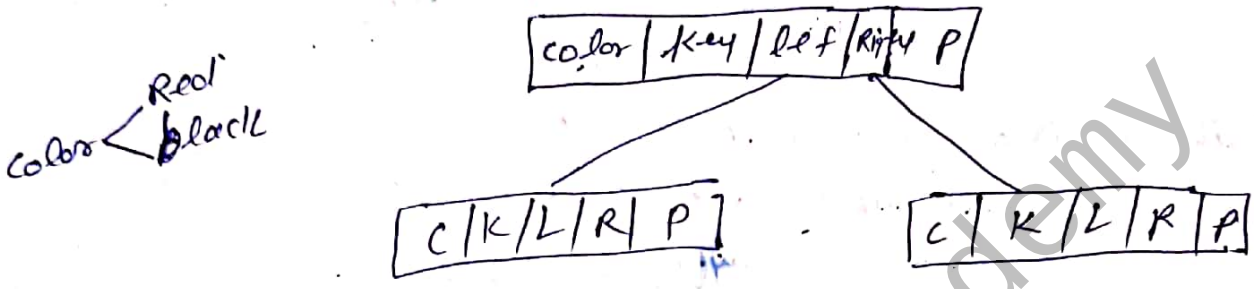
O(h)



Red-Black Tree:

The red black tree are balanced search tree schemes in order to guarantee that ~~basic~~ $O(\log n)$ time in worst case.

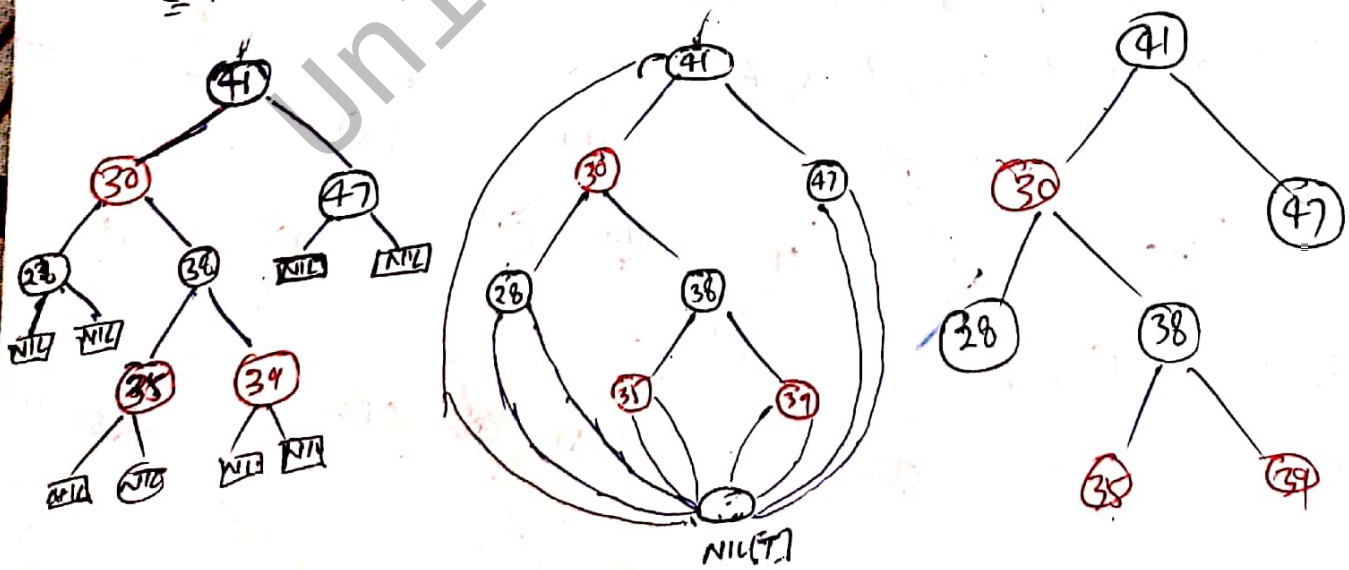
A red black Tree is BST with extra fieild color. hence the node in Red-Black tree contain.



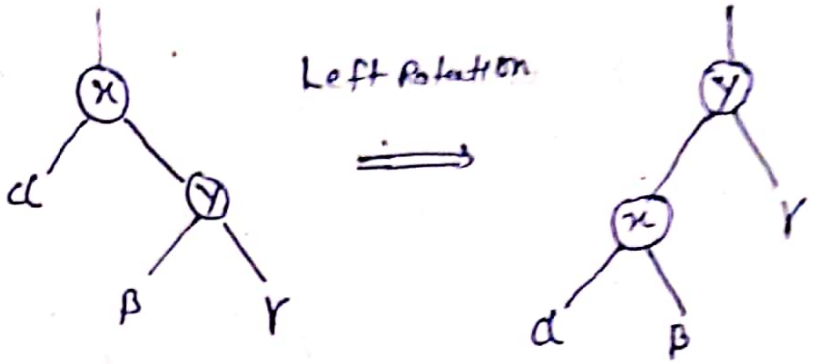
Properties:

1. Every Node is either red or black
2. The root is black
3. Ever leaf (NIL) is black.
4. if a Node is red, then both it's children are black.
5. For each Node all path from the node to descendant leaves contain the same Number of black nodes.

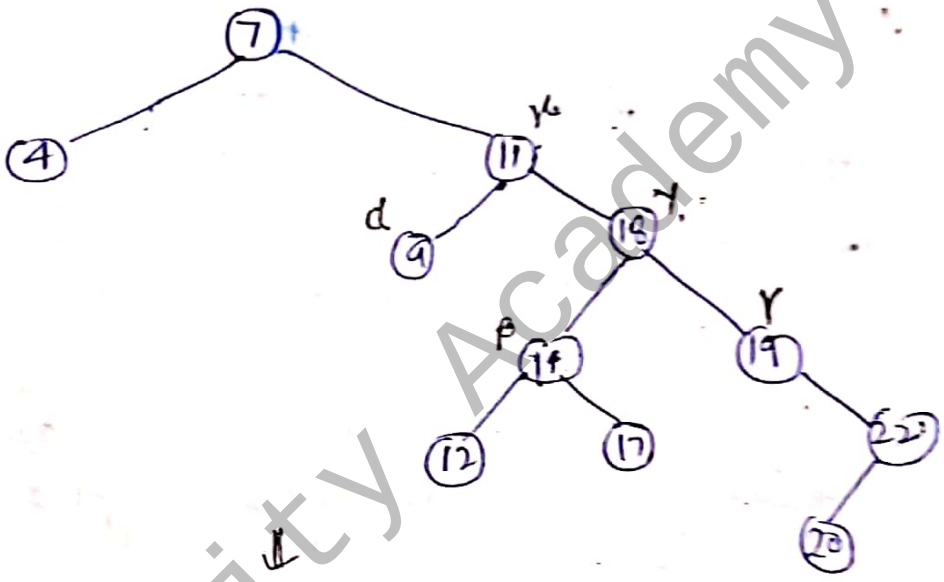
e.g.



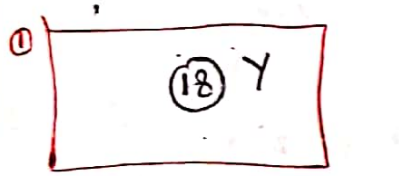
Rotations.

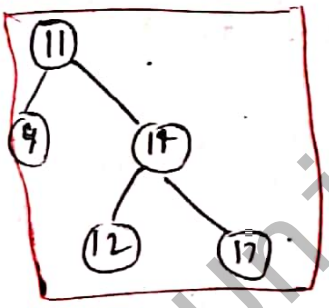


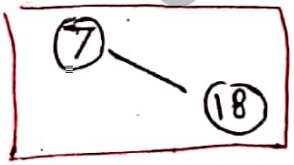
eg.

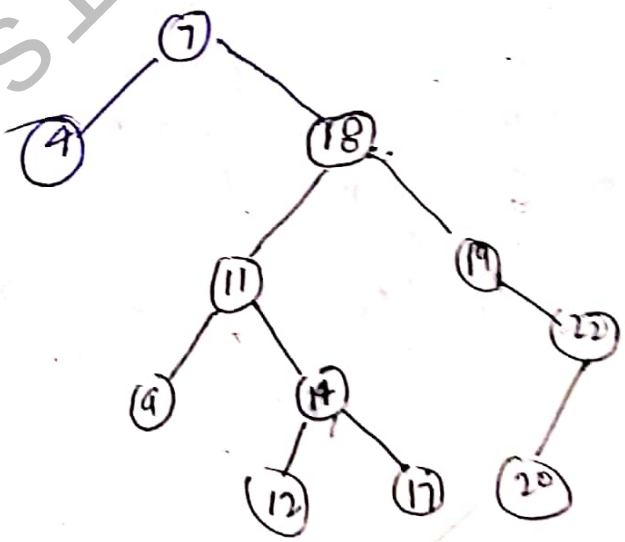


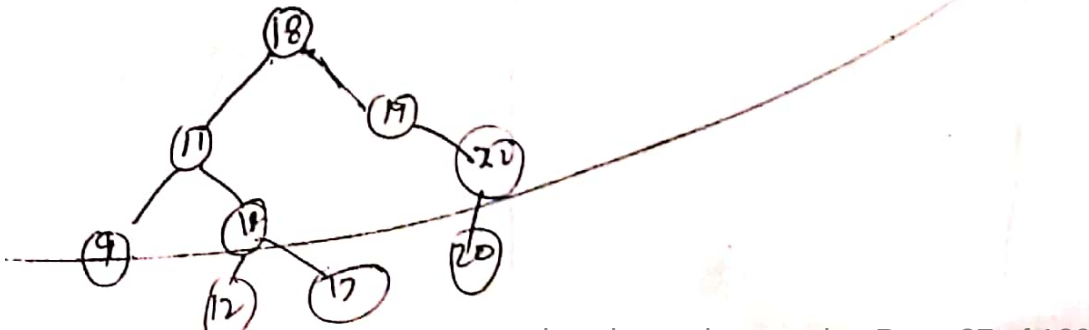
with algorithm.



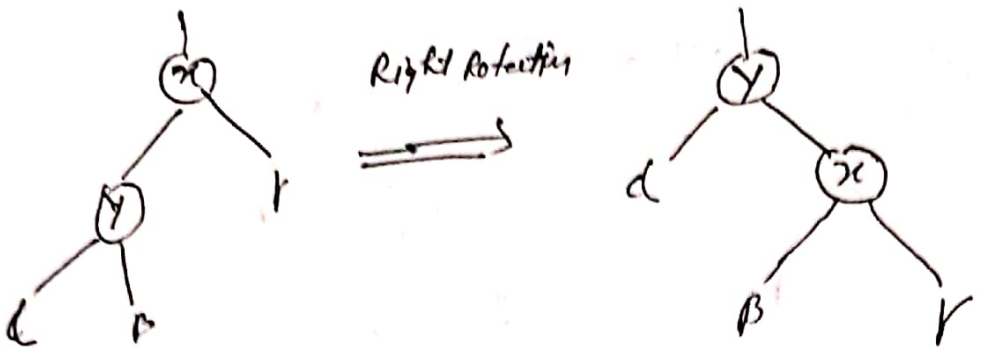






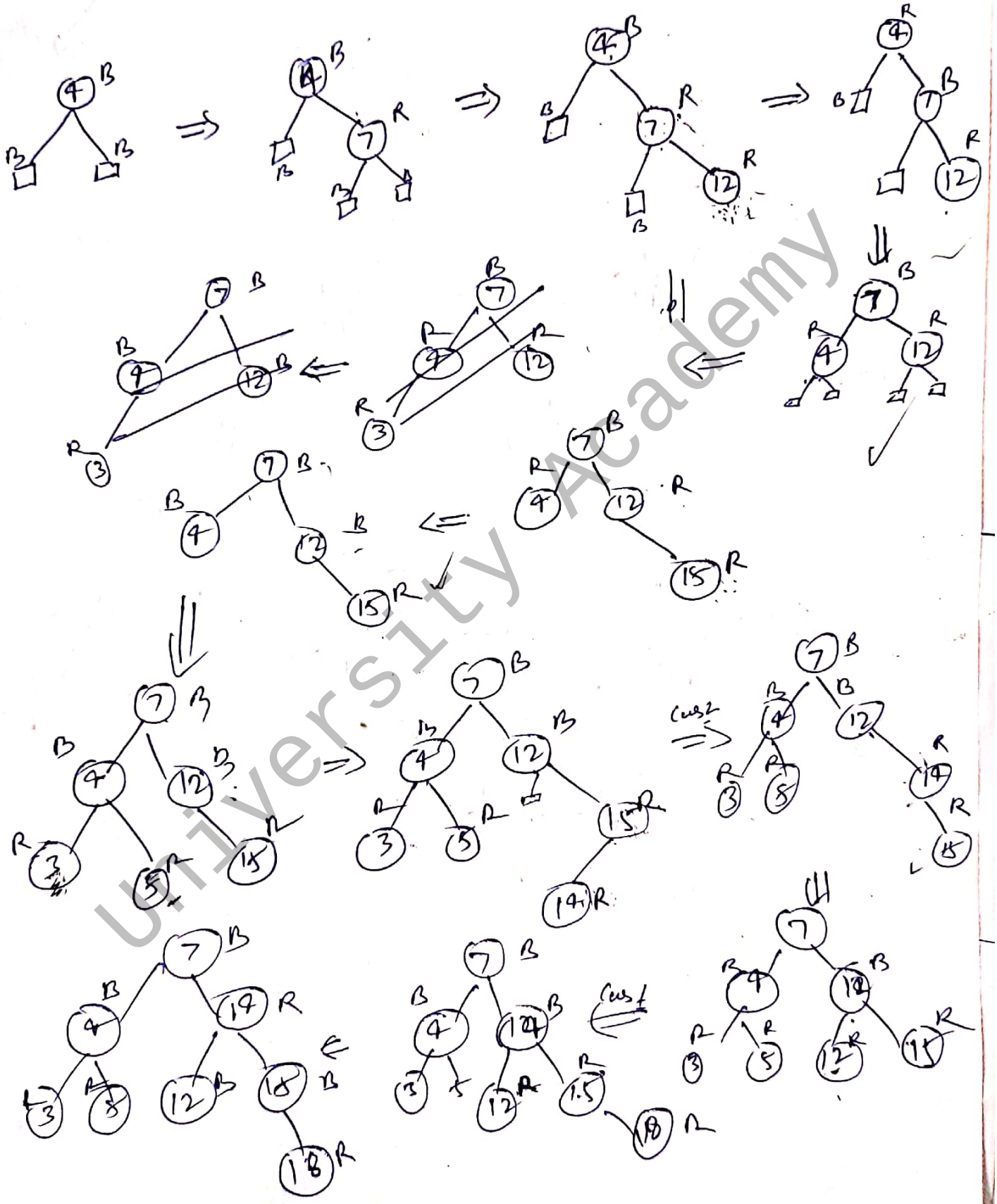


Right Rotation.



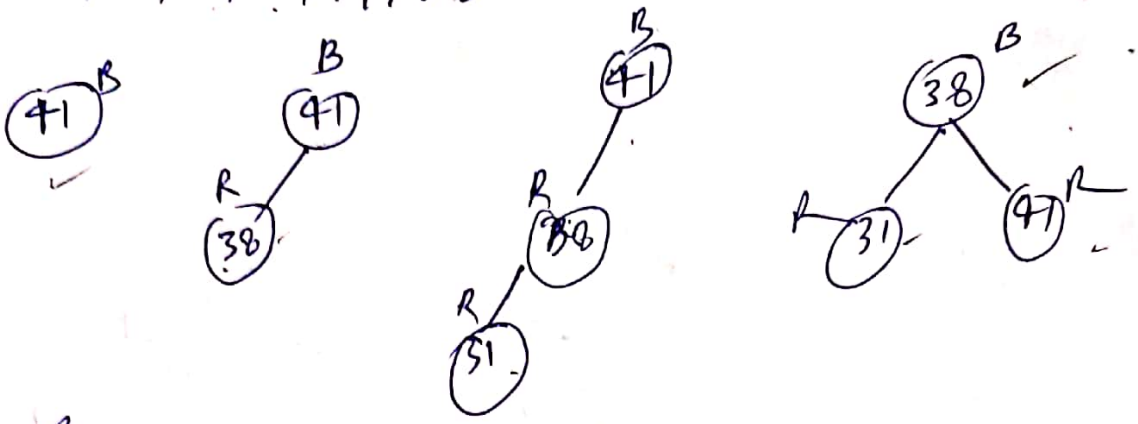
z is a root node	z is not root node.		
	z's parent is Black	z's parent is not Black	
		z's parent is left child of grandparent of z then	z's parent is Right child of the grandparent of z
then simply color it Black	there is nothing to do, as Adding red node doesn't Affect RB. Properties.	Case 1: if z's uncle y is Red. and z is the left or Right child. Case 2: if z's uncle is y is black and z is the right child Case 3 if z's uncle y is black and z is left child.	Case 1 if z uncle is R and z. is the left or Right child. Case 2 if z's uncle y is Black and z is left child. Case 3 if z's uncle is y is black and z is Right child.

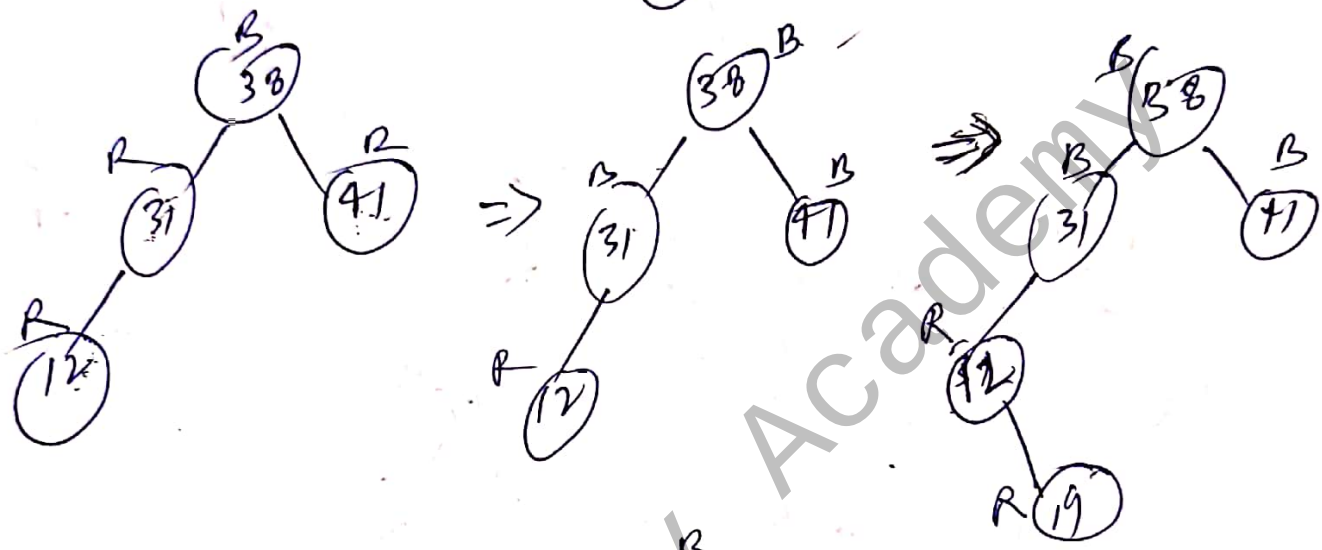
Question. Insert the following key into and initially empty Red-black Tree : <4, 7, 12, 15, 3, 5, 14, 18, 16>

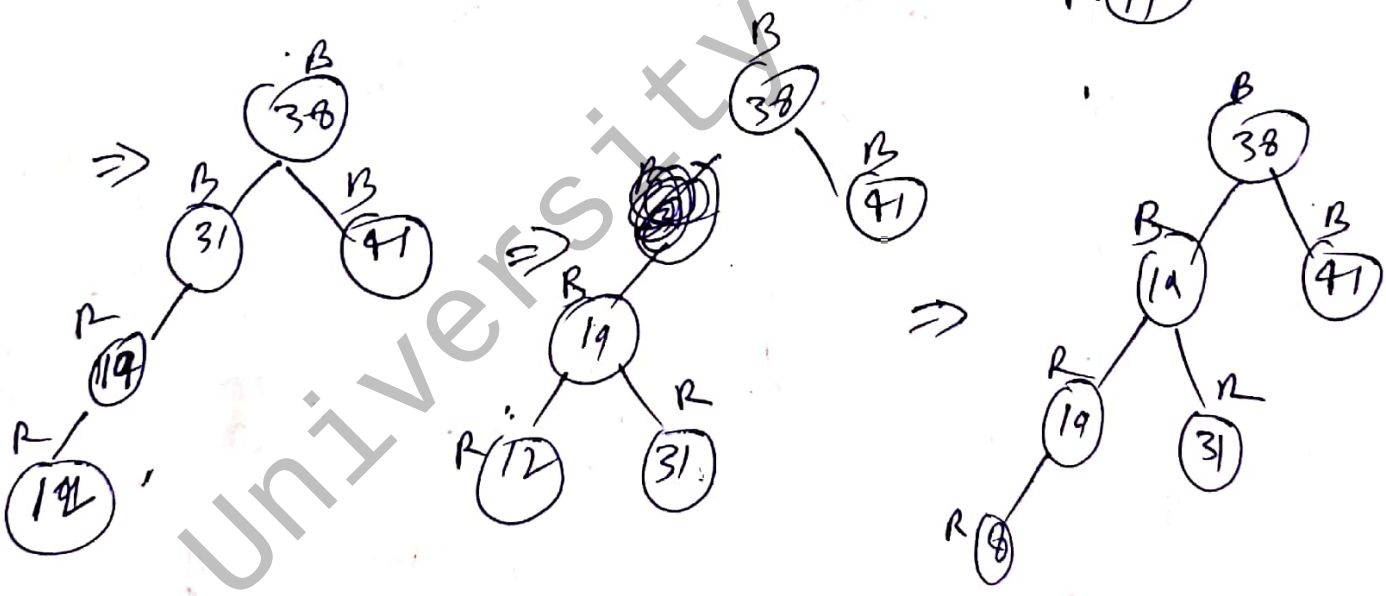


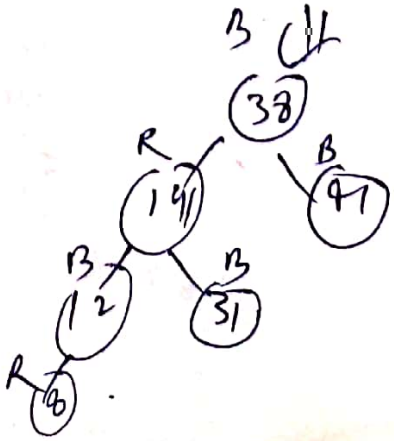
R.B Tree.

Q. Insert 41, 38, 31, 12, 19, 8 ...









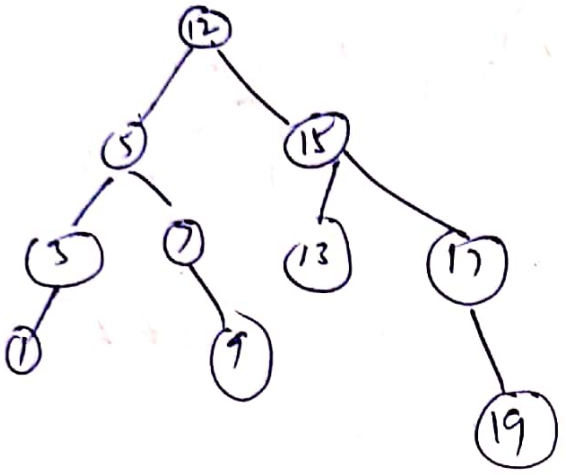
Red-Black Tree deletion:

Backgrounds: BST deletion:

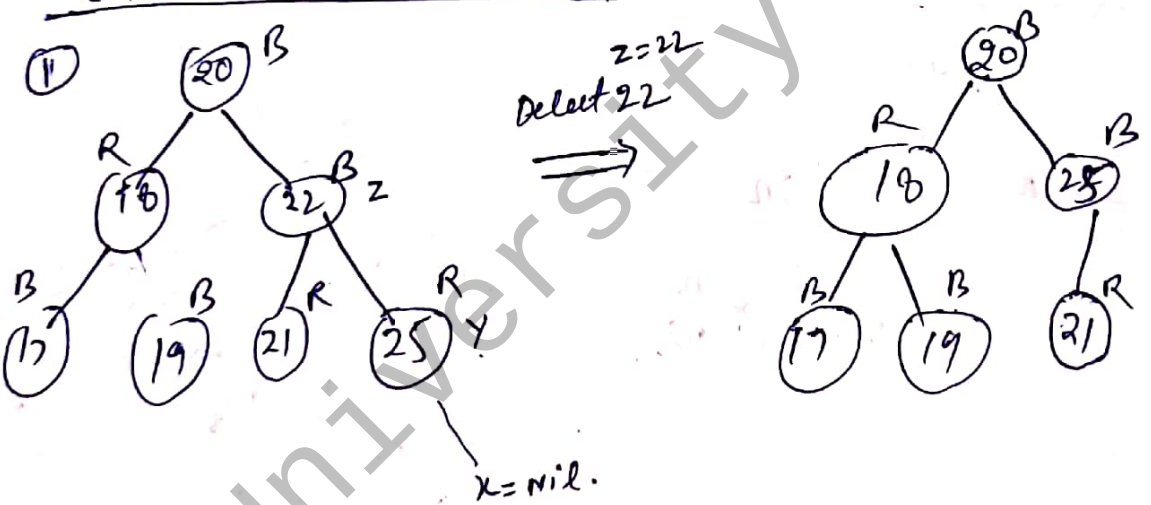
Case : 1 No child.

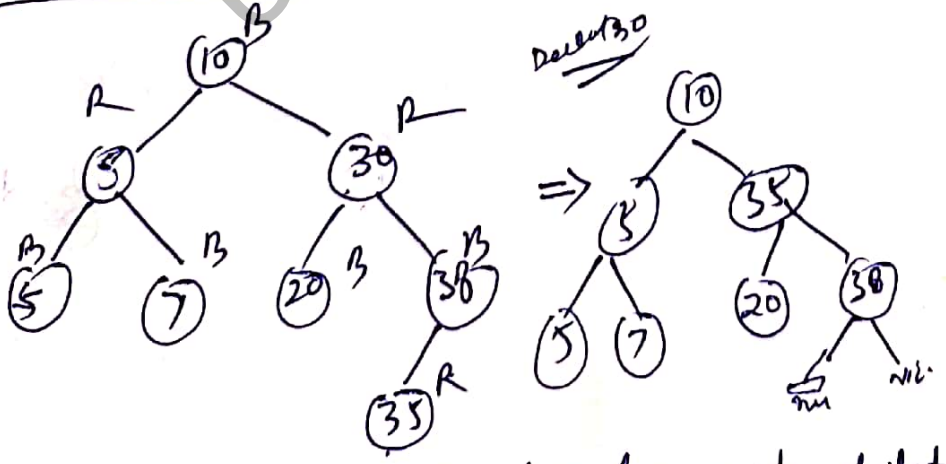
Case : 2. one child

Case 3: 2 childrens.



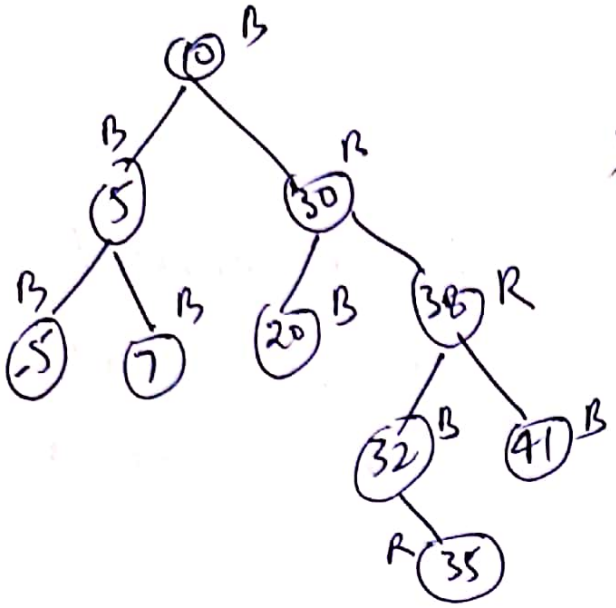
Red. Black Tree deletion.



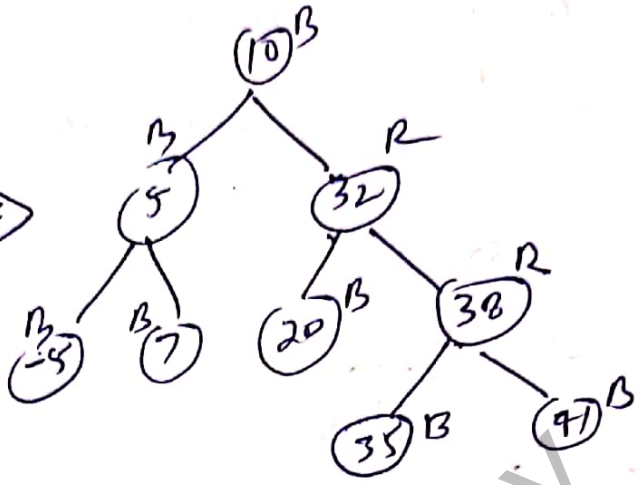


if red node just delet it.

inorder



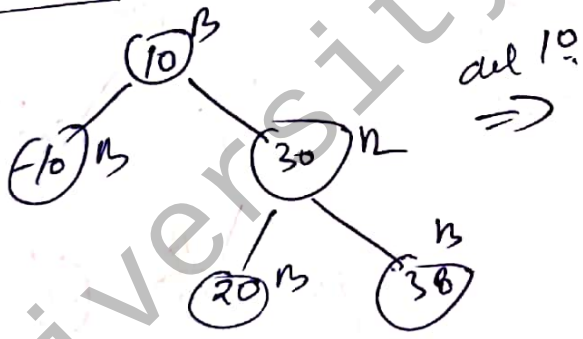
Del 30 ⇒



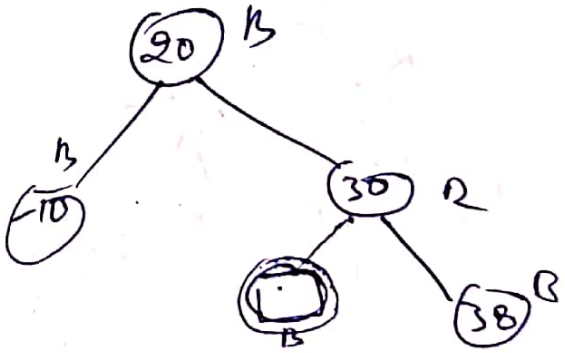
① if black node with red child then delete black node and turn red into black node.

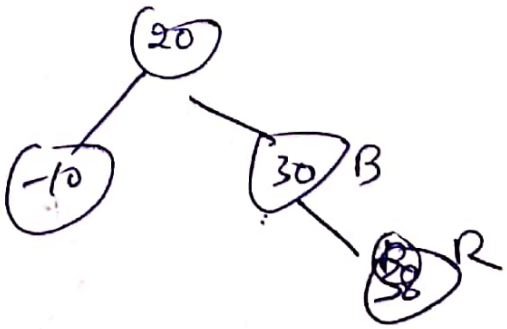
if black node with black children then there are 4 cases:

Double Black Node:



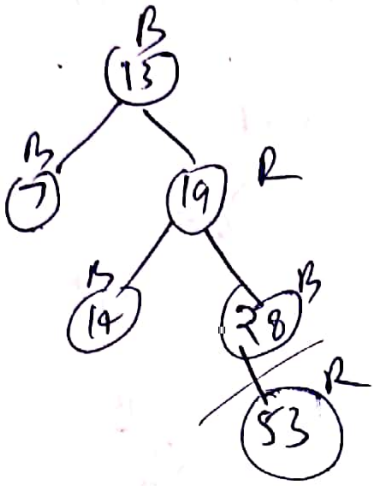
del 10 ⇒



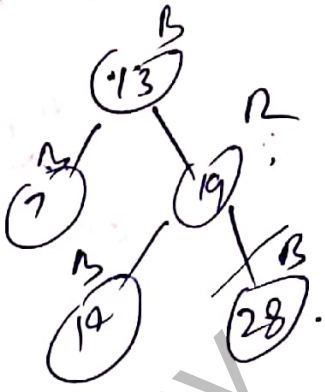


Q: Consider following RB Tree. delete the following key.

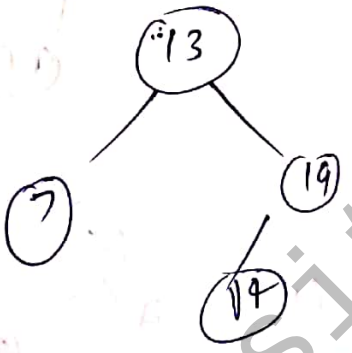
53, 28, 19, 14, 13, 7



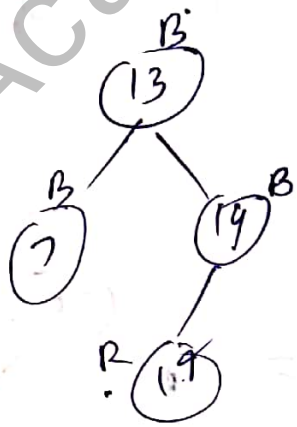
Delete 53 ⟹



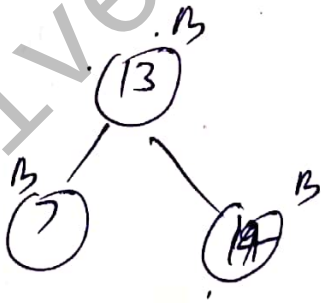
Delete 28 ⇒



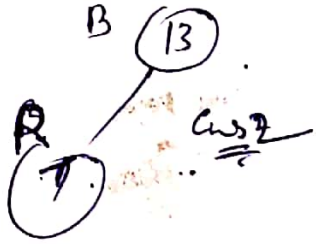
Case 2 ⇒



Delete 19 ⇒



Delete 14 ⇒

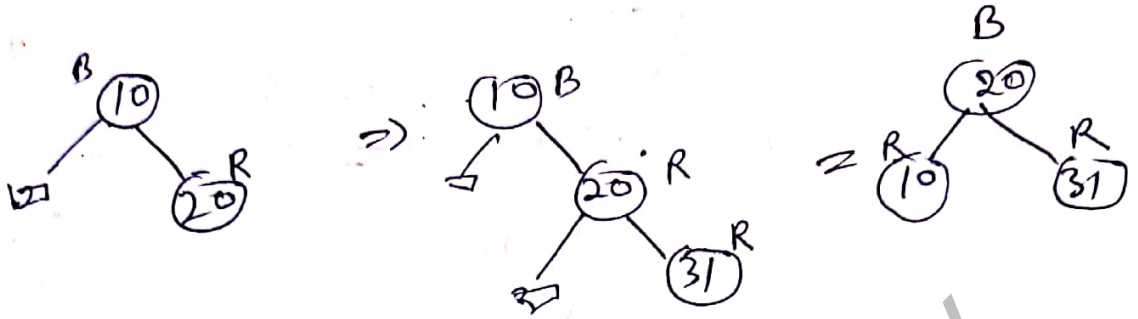


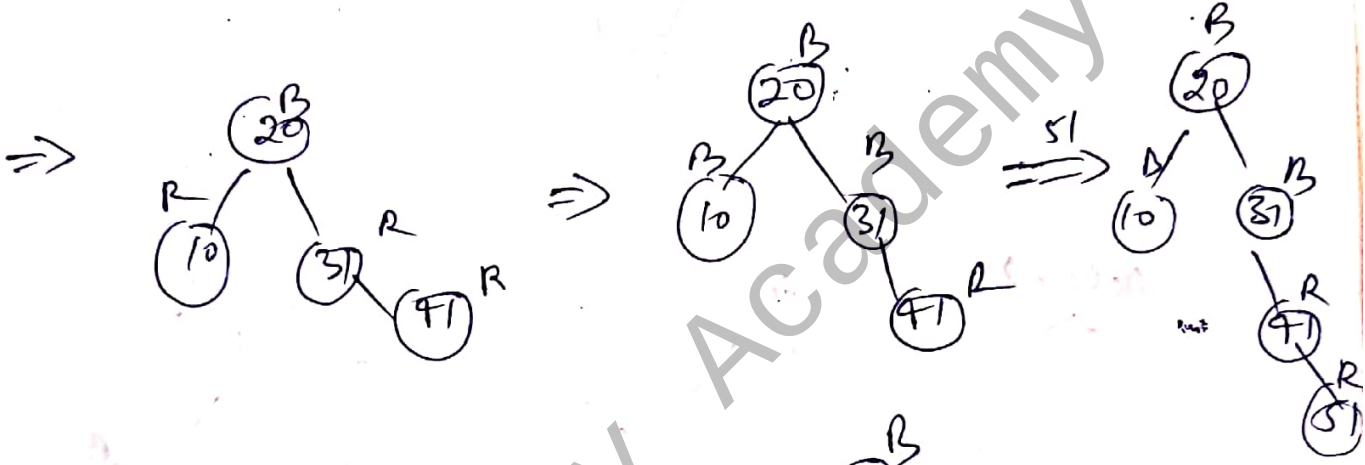
Case 2

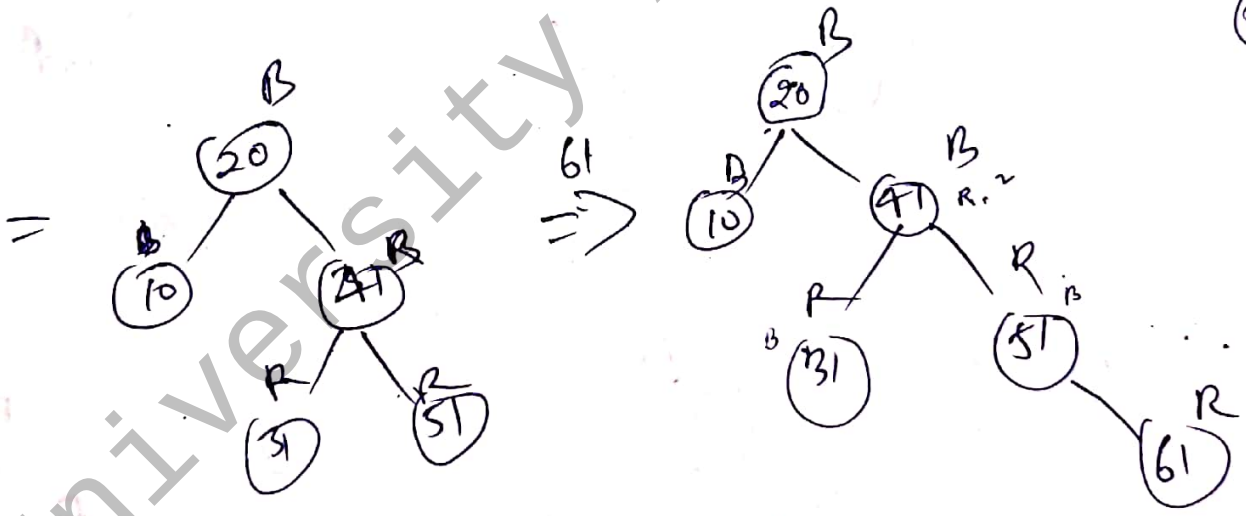
Delete 13 ⇒ 7 B ⇒ Delete 7 = No Tree

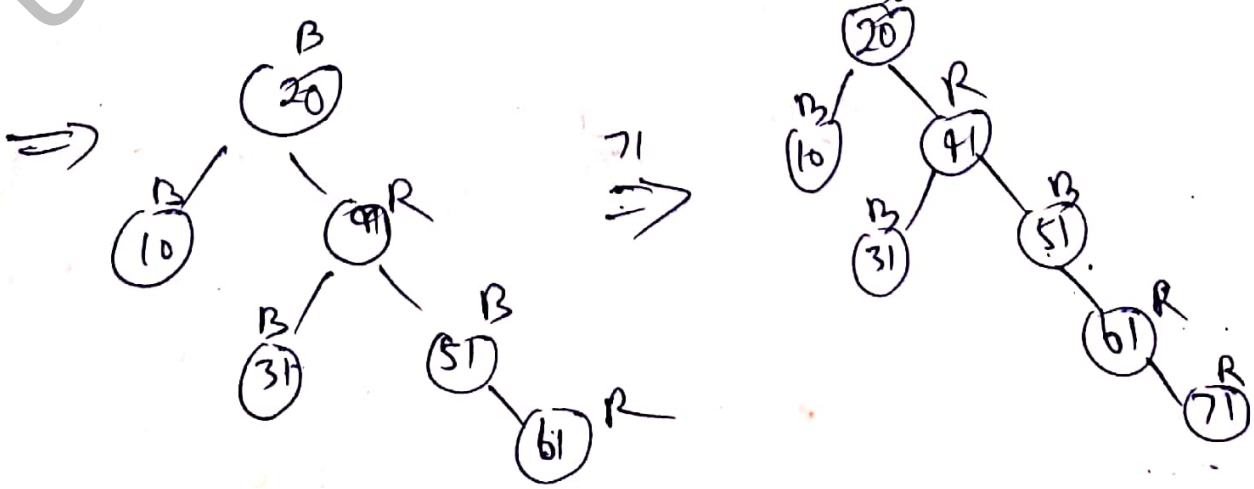
Draw Red-Black tree starting with an empty. Inv

10, 20, 31, 41, 51, 61, 71, 81, 91, 101



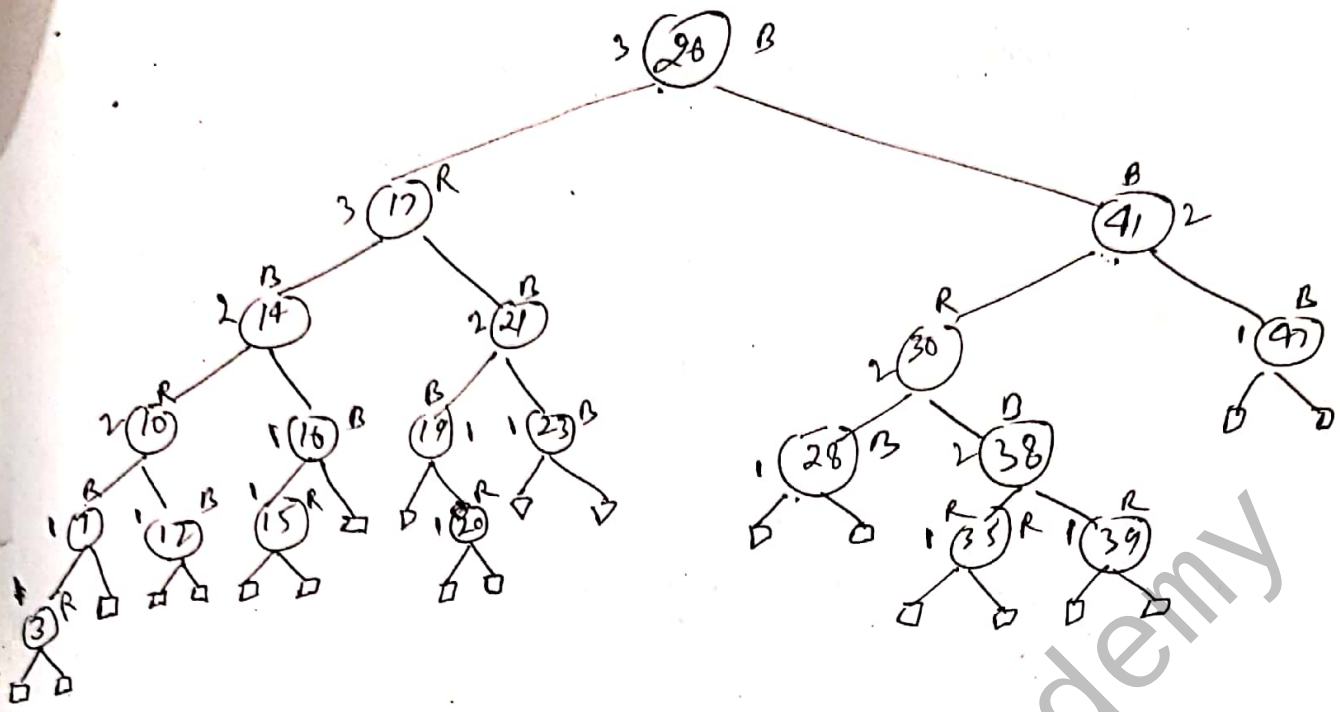






Theorem:

A Redblack Tree with n internal node has height at most $2 \log(n+1)$.



→ we start by showing that subtree rooted at any node x contains at least $2^{bh(x)} - 1$ internal nodes.

→ If height of x is 0 then x must be leaf (nil).

$= 2^{bh(x)} - 1 = 2^0 - 1 = 0$ internal nodes.

→ Consider the node(x) with two children. each child having either $bh(x)$ or $bh(x)-1$ height.

then each child has at least $2^{bh(x)-1} - 1$ internal nodes

then subtree rooted at x contains at least

$$\left(2^{bh(x)-1} - 1\right) + \left(2^{bh(x)-1} - 1\right) + 1 = 2^{bh(x)} - 1$$

Let h be the height of tree → at last half of the node on any simple path from root to leaf must be black,

the black height of root must be at last $h/2$ the

$$n \geq 2^{h/2} - 1.$$

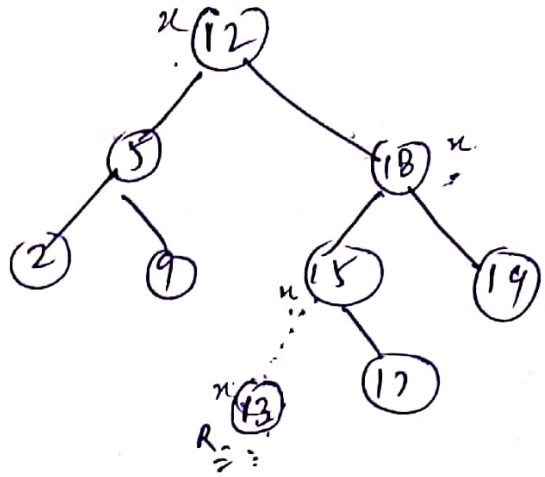
$$n+1 \geq 2^{h/2}$$

$$\therefore \log(n+1) \geq h/2 \quad \text{or} \quad h \leq 2\log(n+1) \text{ proved}$$

Insertion

- RB-INSERT(T, z)

```
1- y ← nil[T]
2  x ← root[T]
3. while x ≠ nil[T]
4     do y ← x
5        if key[z] < key[x]
6           then left[x] ← left[x]
7           else x ← right[x]
8     p[z] ← y
9     if y = nil[T]
10       then root[T] ← z
11       else if key[z] < key[y]
12            then left[y] ← z
13            else right[y] ← z
14    left[z] ← nil[T]
15    right[z] ← nil[T]
16 .  color[z] ← RED
17    RB-Insert-Fixup(T, z).
```



z = 13
y = nil[T]
x = root[T]
y = root[T], y = 18, y = 15.

RB-INSERT-FIXUP(T, z)

```
1 while color[p[z]] = Red
2 do if p[z] = left[p[p[z]]]
3       then y ← right[p[p[z]]]
4            if color[y] = RED
5               then color[p[z]] ← BLACK
6                    color[y] ← BLACK
7                    color(p[p[z]]) ← RED
8                    z ← p[p[z]]
9               else if z = right[p[z]]
10                   then z ← p[z]
11                        LEFT ROTATE (T, z)
12                   color[p[z]] ← Black
13                   color(p[p[z]]) ← RED
14                   RIGHT-ROTATE (T, p[p[z]]).
15      else (Same as then with right and left exchange)
16 - color[Root[T]] = Black.
```

RedBlack tree violations & Selection.

Case 1

1. Uncle is Red

Selection: 1. Change color of parent, uncle and grand parent
2. Grand parent becomes new node to check violation.

Case 2

1 - Uncle is Black
2a. Node is a left child of a Right child
2.b Node is a Right child of a left child

Selection

1a. Right Rotate around parent for 2a = Case 3
1.b. left rotate around parent for 2b = Case 3
2. Parent become the new node to check for violation

Case 3

1. Uncle is Black
2a. Node is a Right child of Right child
2b. Node is a left child of left child

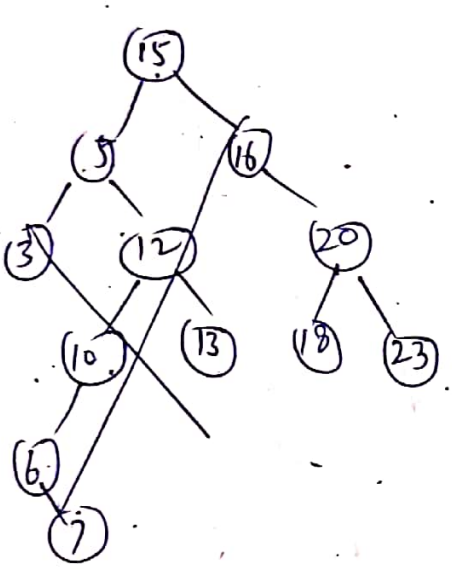
Selection

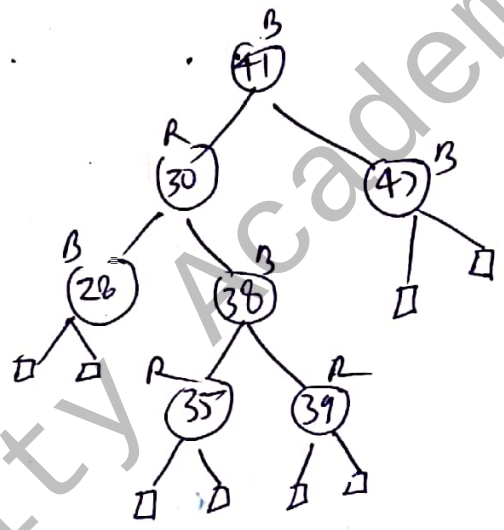
1a. left Rotate around grand parent for 2a
1.b Right Rotate around grand parent for 2b
2. ~~Swap~~ change color of parent and grand parent.
3. grand parent became the new node to check violation.

Deletion (Red Black Tree) –

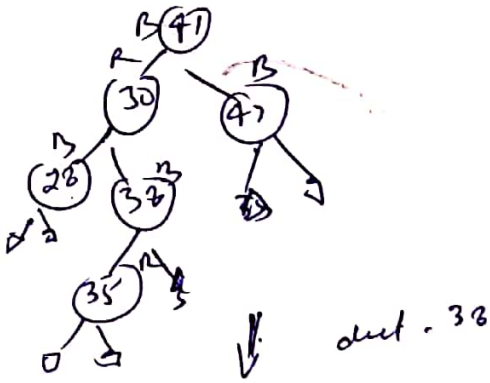
Deletion of node takes O(log n) time. To understand deletion notion of double black is used. when Black node is deleted and Replaced by Black child the child is marked as double Black

Deletion steps: ~~Perform RB delete~~ when we perform delete operation in BST or RBT we always end up deleting a node which is either leaf or has only one child.

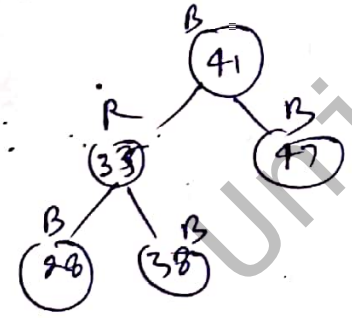




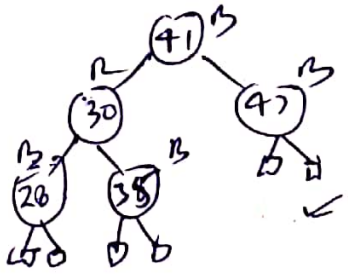
delete 39



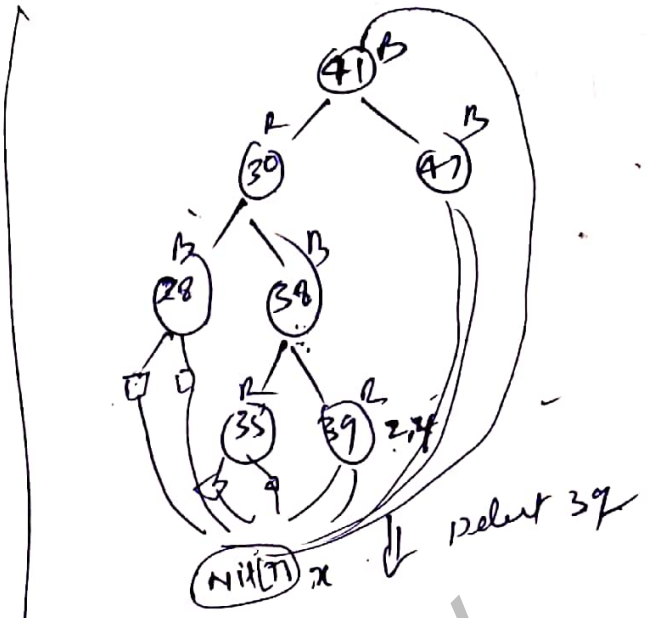
delete 30

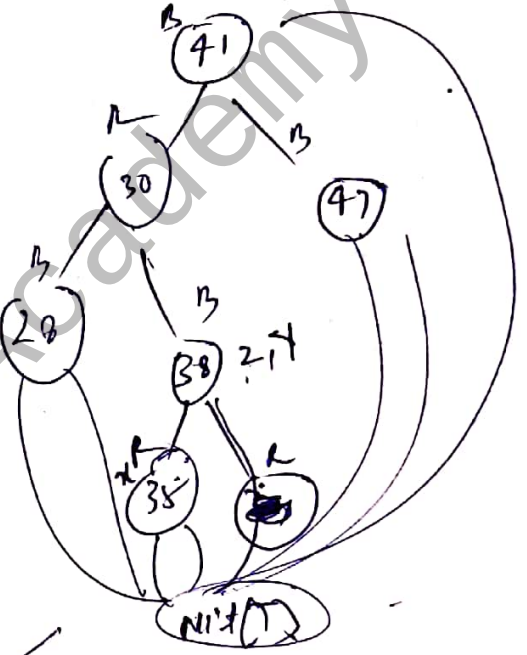


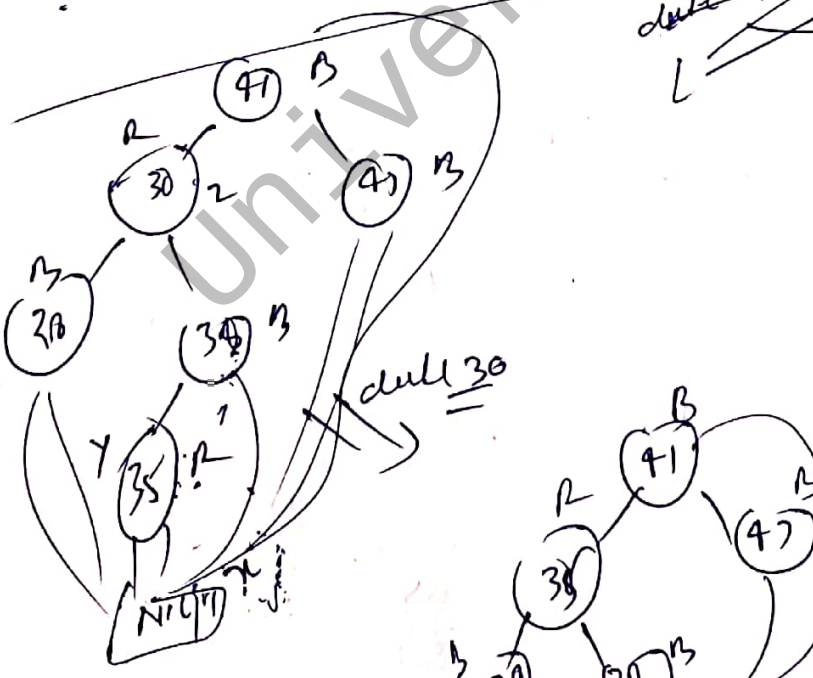
delete 38

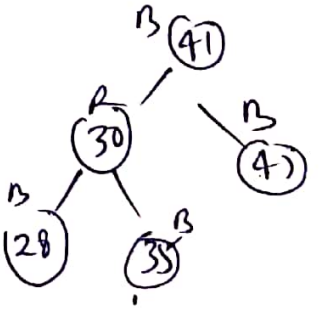


```
RB-Delete (T, z)
1. if left[z] = nil[T] or right[z] = nil[T]
2       then y ← z
3       else y ← Tree-Successor(z)
4. if left[y] ≠ nil[T]
5       then x ← left[y]
6       else x ← right[y]
7. p[x] ← p[y]
8 if p[y] = nil[T]
9       then root[T] ← x
10      else if y = left[p[y]]
11              then left[p[y]] ← x
12              else right[p[y]] ← x
13 if y ≠ z
14      then key[z] ← key[y]
15           copy y's satellite data into z
16 if color[y] = Black
17      then RB-Delete Fixup(T, x)
18. return y
```





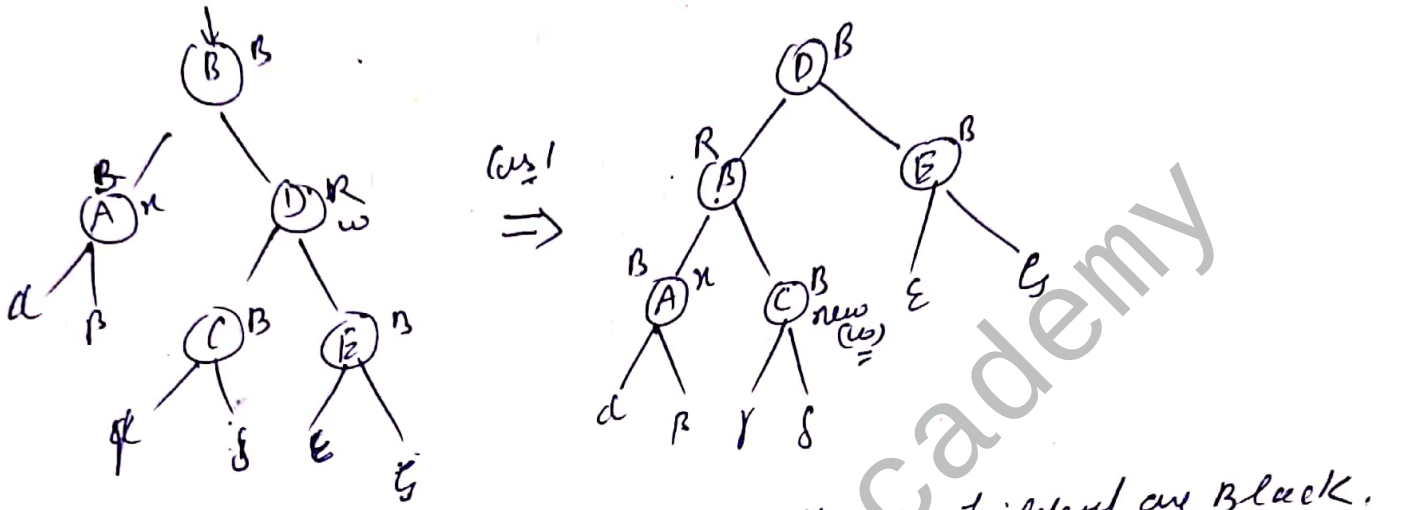




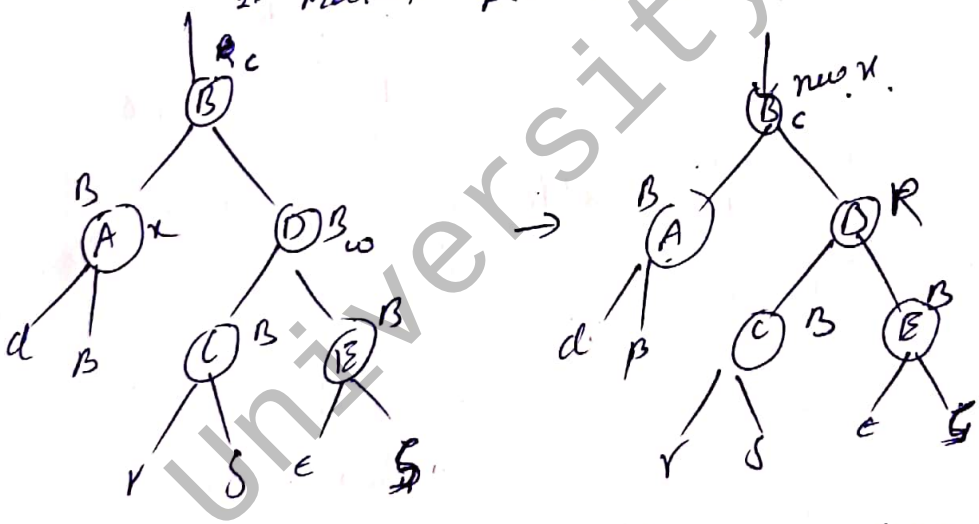
α - alpha
β - Beta
γ = gamma
δ - delta
ε - Epsilon
ζ - zeta

RB-Delete Fixup
⇒ Color of x = Black and x is not root =

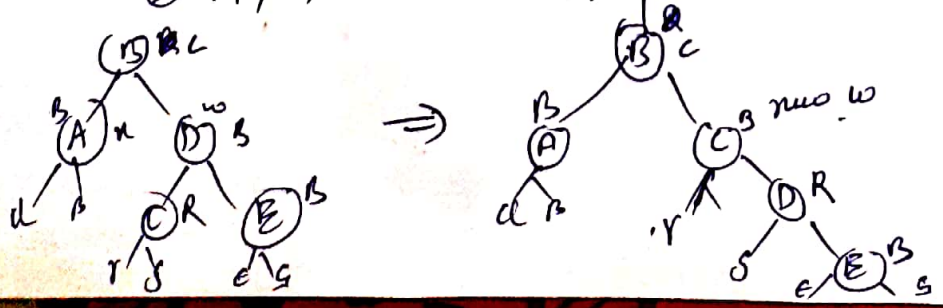
Case 1. x's sibling w is red!
(1) Change the color of w and P(x)
(2) Perform left Rotation on P(x) : Case 2, Case 3, Case 4



Case 2 x's sibling w is black and both of w's children are Black.
1. change the color of w
2. make P(x) as new x.

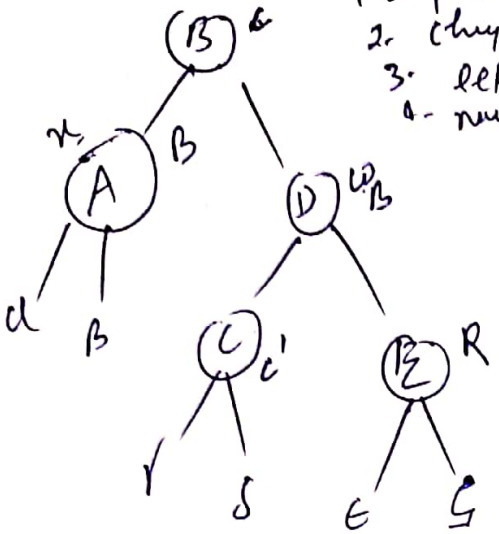


Case 3 x's sibling w is Black w's left child is Red and w's right child is Black
1- change the color of w and it's left child color
(2) Right rotation on w = Case 4.

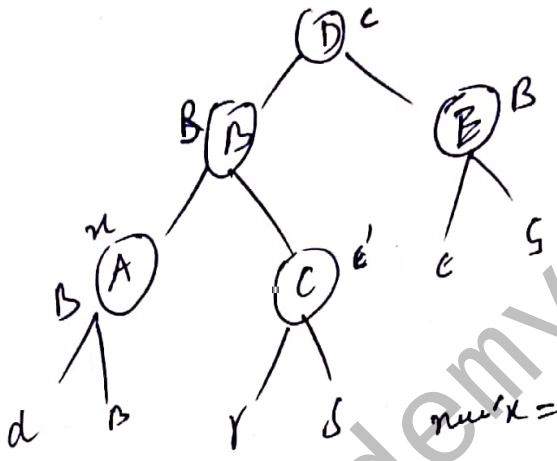


Case 4 x's sibling w is Black and with Right child is Red.

1. Change Color of w to the color P $P[x]$, and color $P[x]$'s is Black.
2. Change the colour of Right w.
3. Left Rotate around $P[x]$
4. now x is root



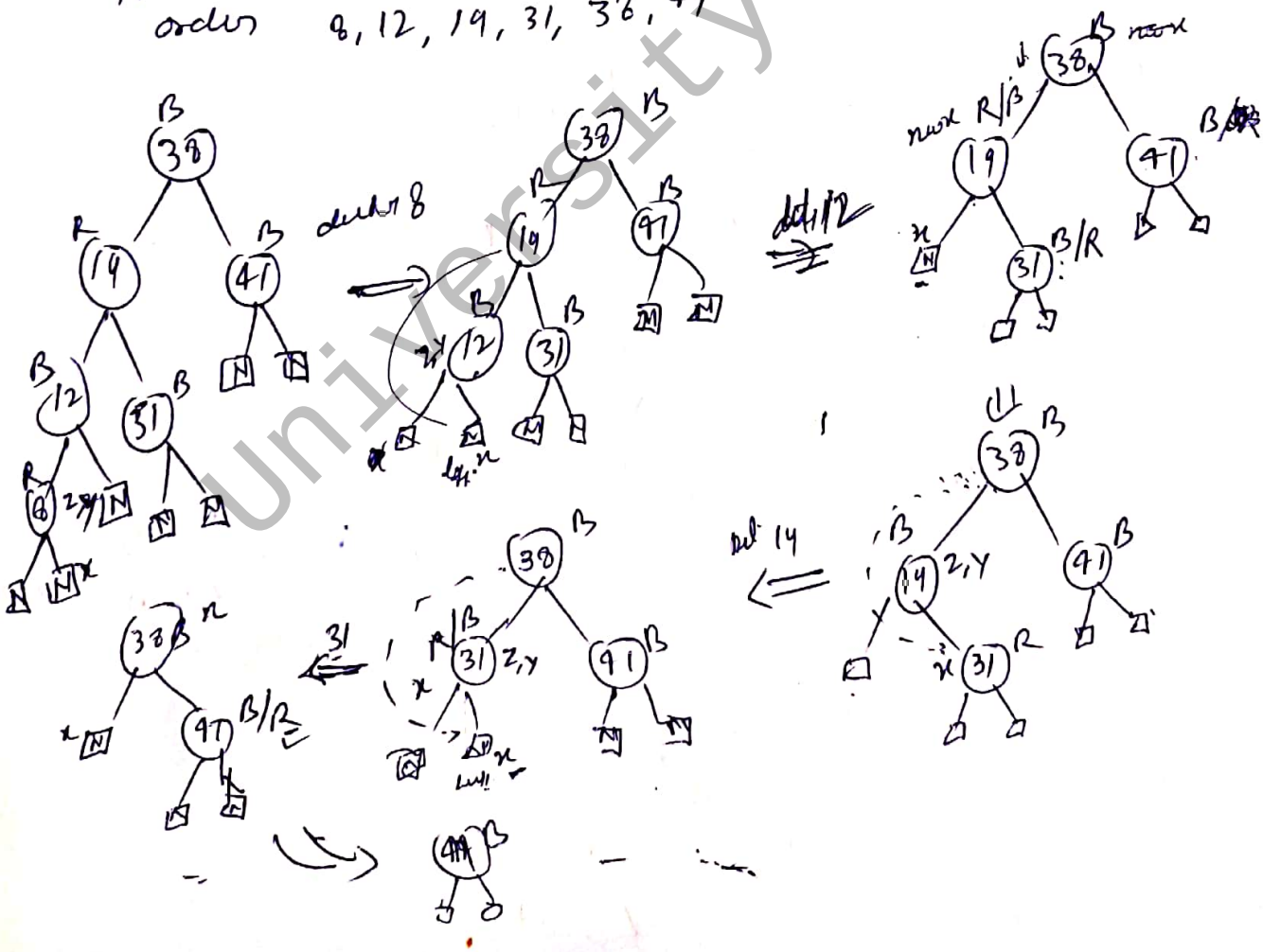
Case 4 ⇒



now x = root[T]

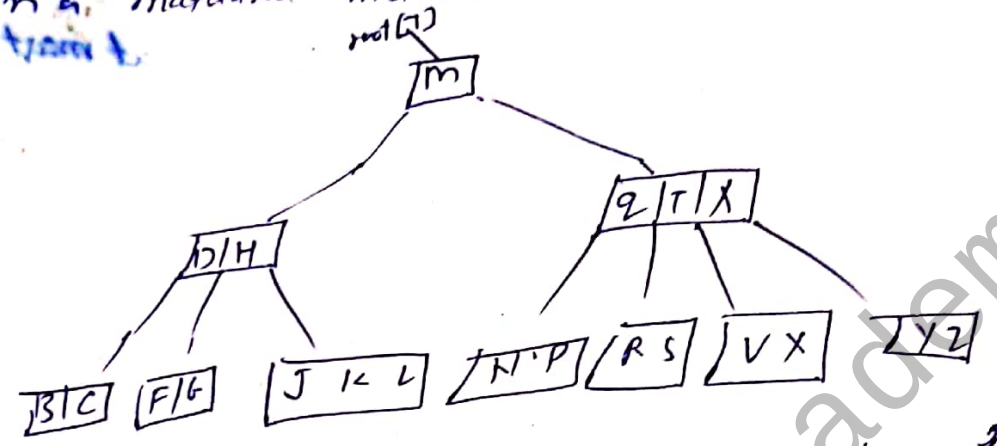
=

In the R-B Tree given below show the ~~result~~ ~~show~~ red-black tree that result from successive deletion of the keys in the order 8, 12, 19, 31, 38, 41



B-Tree B-Tree are Balanced search tree designed to work well on magnetic disks or other direct access secondary storage devices.

B-Tree node may have many children from handfull to thousands. B-Tree generalize binary search tree in a natural manner.



if internal node x contain $n[x]$ keys then x has $n[x]+1$ children, and all leaves are at the same depth.

A B-Tree T is a rooted tree (where root is Root[T]) having following properties.

1. Every node x has following properties.
   a. $n[x]$ the number of keys currently stored in node x.
   b. the $n[x]$ key themselves sorted in non-decreasing order.
      $key_1[x] \leq key_2[x] \leq \cdots \leq key_{n[x]}[x]$
   c. $leaf[x] = \begin{cases} true & \text{if } x \text{ is leaf} \\ False & \text{if } x \text{ is an internal node.} \end{cases}$
2. Each internal node also contain $n[x]+1$ pointers $c_1[x], c_2[x], \cdots c_{n[x]+1}[x]$ to its children.
3. The keys $key_i[x]$ separate the range of keys stored in each subtree.
   $k_1 \leq key_1[x] \leq k_2 \leq key_2[x] \leq \cdots key_{n[x]} \leq k_{n[x]+1}$
4. All leaves have same depth, which is the tree's height.
5. there are lower and upper Bound on the number of keys a node contain.

Thus Bound can be expressed in term of fixed integer $t \geq 2$ called minimum degree.

Lower Bound: every node other than root must have atleast $t-1$ key and Atleast t children

Upper Bound: every node can contain at most $2t-1$ key :-

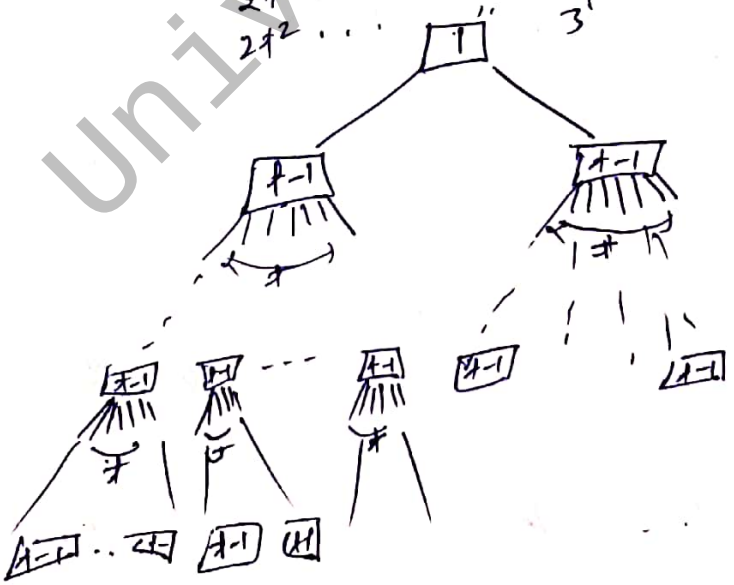
Every internal node has at most $2t$ children

Note: the simplest B-Tree occurs when $t=2$. Every internal node then has either 2, 3, or 4 children also called 2-3-4 tree.

Theorem if $n \geq 1$ then for any n-key B-Tree T of height h and minimum degree $t \geq 2$, $h \leq \log_t \frac{n+1}{2}$

Proof-

if B-Tree has height h
the root contains atleast 1 key
all other node contains $t-1$ key
there as atleast 2 nodes at Depth 1,
$2t$ " " " 2
$2t^2$ " " " 3



Depth	Now/next
0	1
1	2
2	$2t$
3	$2t^2$

Now/next

$$n \geq 1 + (t-1)\sum_{i=1}^{h} 2t^{i-1}$$

$$\geq 1 + 2(t-1)\left(\frac{t^h - 1}{t-1}\right)$$

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

$$n = 2t^h - 1 \Rightarrow t^h = \frac{n+1}{2}$$

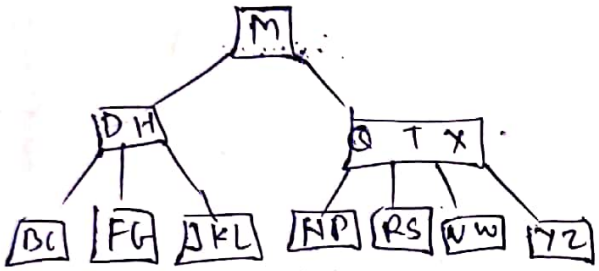
$$h = \log_t \frac{n+1}{2}$$

B-Tree operations-

Root of B-Tree always in main memory.
(Disk Read Never required)

1. B-Tree search.
2. B-Tree - Create
3. B-Tree - Insert
4. B-Tree. delete.

B Tree Search.



Search (x, k)

K = R

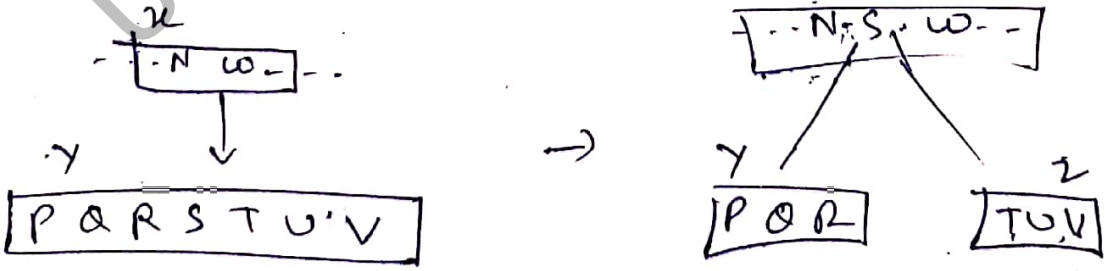
```
B-Tree-Search (x, k)
1 i ← 1
2 while i ≤ n[x] and k > key_i[x]
3     do i ← i+1
4 if i ≤ n[x] and k = key_i[x]
5     then Return (x, i)
6 if Leaf [x]
7     then Return NIL
8     else DISK-READ (c_i[x])
9          Return B-Tree Search(c_i[x], k)
```

~~Ins~~ BTree Create → create empty node then calls B-Tree Insert to add new key

B-Tree Insert →

Splitting a node.

t = 4



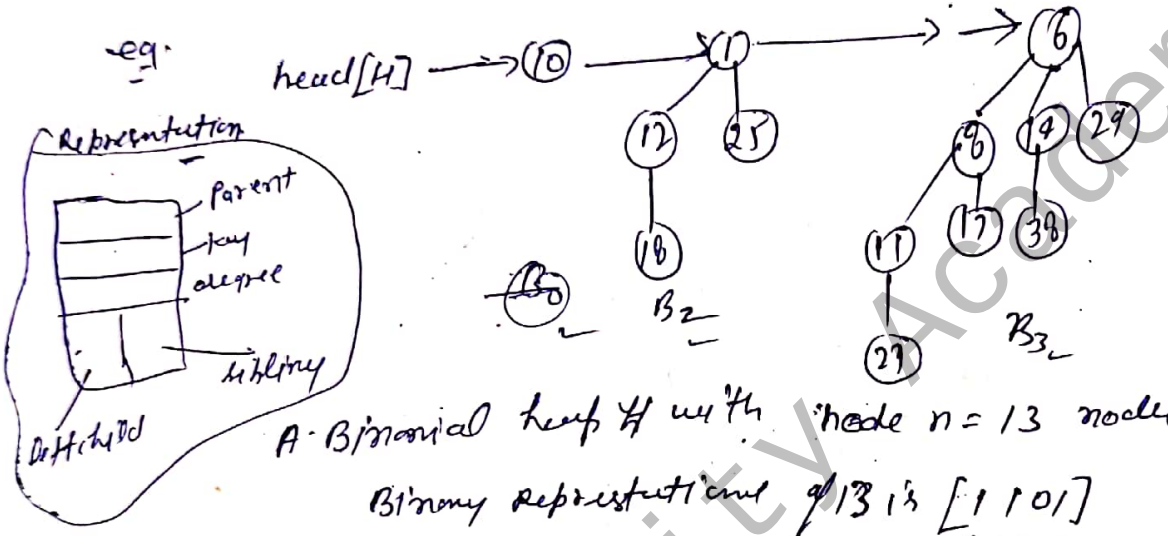
LMNOR

Binomial heap

A Binomial heap is a set of binomial trees that satisfies following Binomial - heap properties.

1. Each Binomial - heap obeys the min - heap properties.
2. for any non- negative integer k, there is at most one binomial tree in H whose root has degree k.

eg.



A Binomial heap H with node n = 13 nodes.

Binary representation of 13 is [1 1 0 1]
$B_3\ B_2\ B_1\ B_0$

Operation on Binomial heap

1. Creating a new Binomial heap → To make an empty binomial heap — O(1)
2. Finding the minimum key → the minimum key reside at root node. Almost O(log n + 1) times to take finding minimum.
3. Union of two Binomial heap — O(log n)
4. Inserting a node — O log n
5. Removal of the root of tree — (log n)
6. Decreasing a key — (log n)
7. Deleting a node — (log n) → Decrease key x to −∞ then call extract min — (1) heap

Binomial Heaps

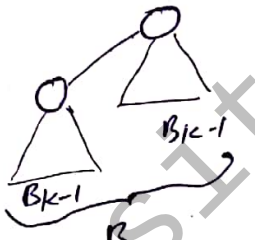
A Binomial Heap is a data structure that has similar behaviours to a heap but it allows the merge operation.

To understant Binomial heap you must understad Binomial trees.

A Binomonical heap is a collection of Binomaical trees eah of which satisfies the heap property (min. heap)

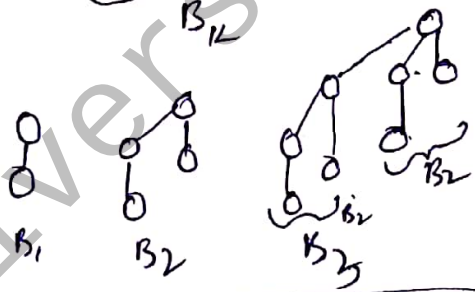
Binomial tree- Binomial tree B_k is an ordered tree define recursively
* the Binomial tree B_0 consist of single node.
* the Binomial tree B_k consist two tree B_{k-1} that are linked together (Root of one is the leftmost child of the root of the other).

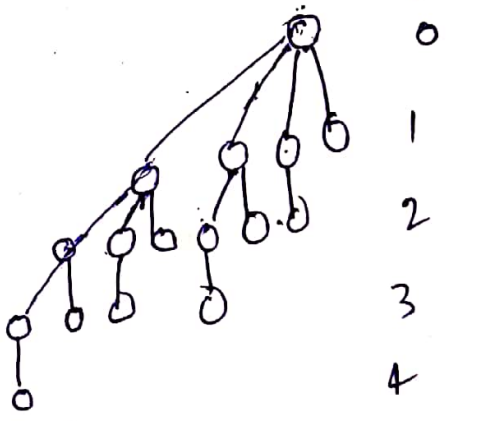
a)



B_0

b)

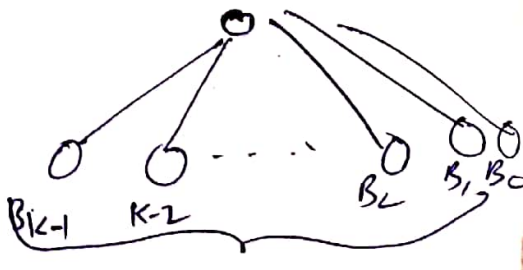




B_4

Properties of Binomial trees for B_k

1. there are 2^k nodes
2. the height of tree is k.
3. there are exactly kc_i nodes of depth $i = 0, 1, \ldots, k$.
4. the root has degree k which is greater than any other node. the children of the root are numbered from left to right $[k-1, k-2, \ldots, 0]$



B_k

$kc_i = \frac{k}{i!(k-i)!}$ $k=4$ $i=3$

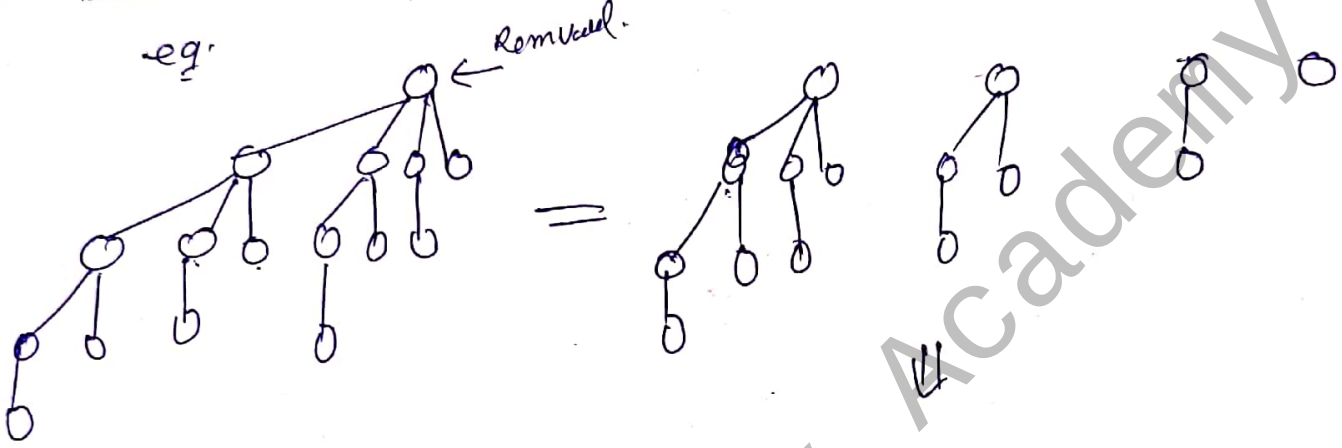
$i=2 = \frac{4}{2!2!} = 6$ $\frac{4}{3!(4-3)!} = 4$

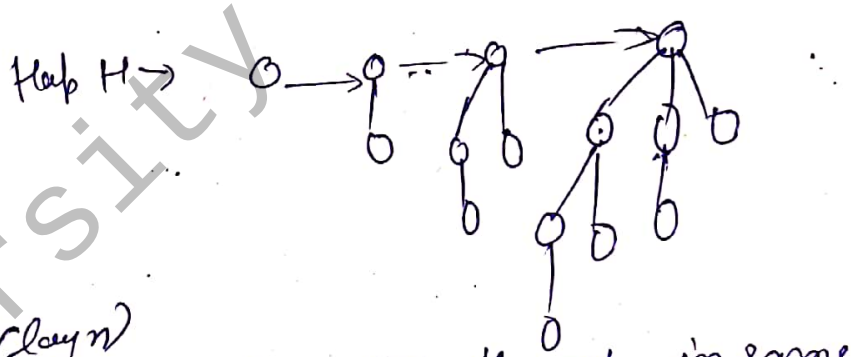
Inserting a node. (log n)

To Insert an element (x) into a heap H, simply create a new heap containing x and Unify it with H. the following procedure Insert node x into binomial heap H.

Removal of the Root of a tree or Extracting the node with minimum key (log n)

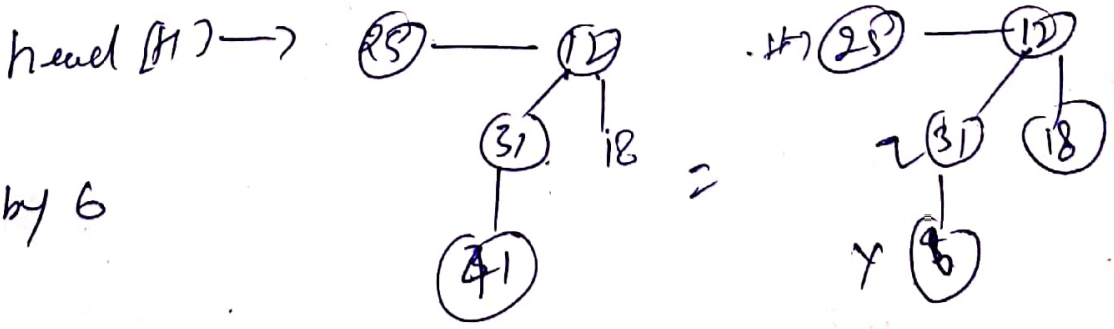
eg.

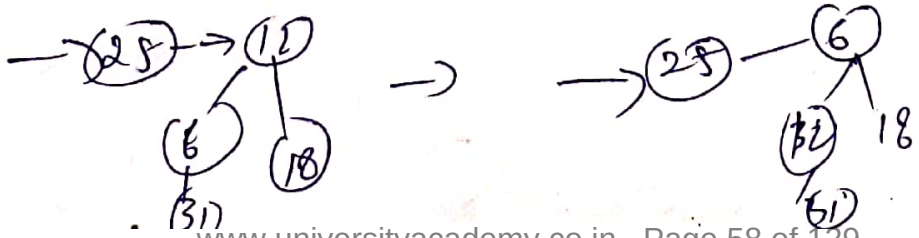




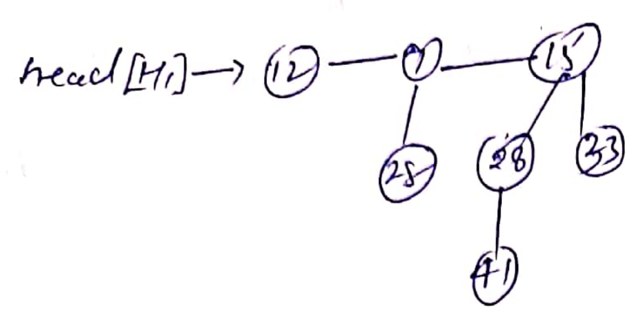
Decreasing a key (log n)

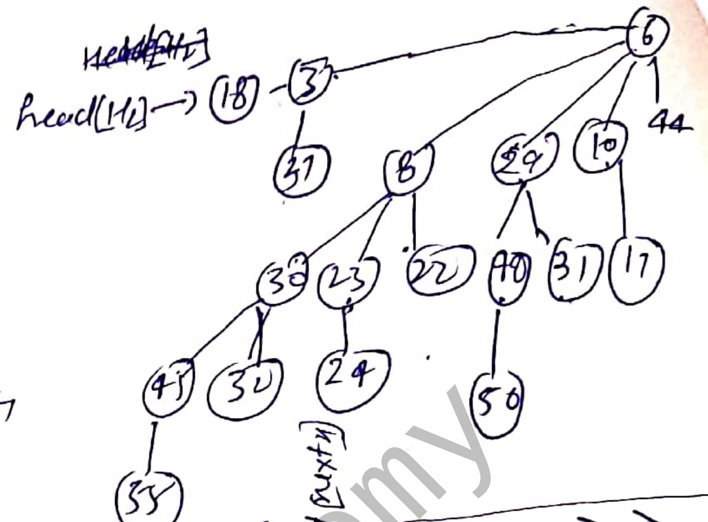
Procedure decreasing ~~key~~ in same manner as in ~~binary~~ min-heap



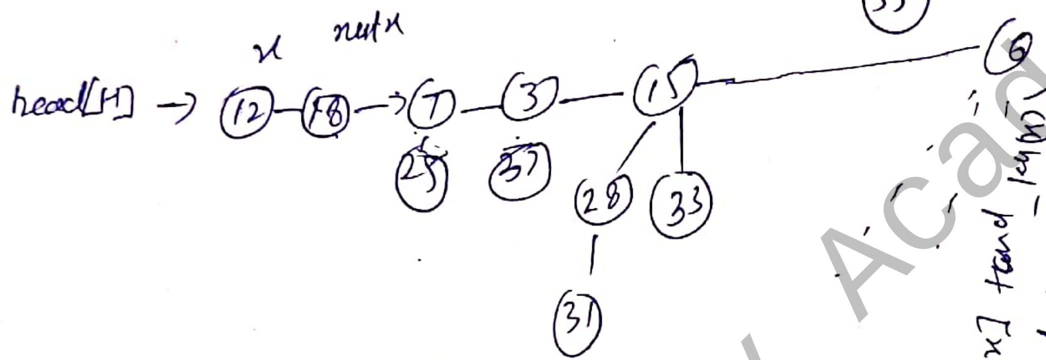


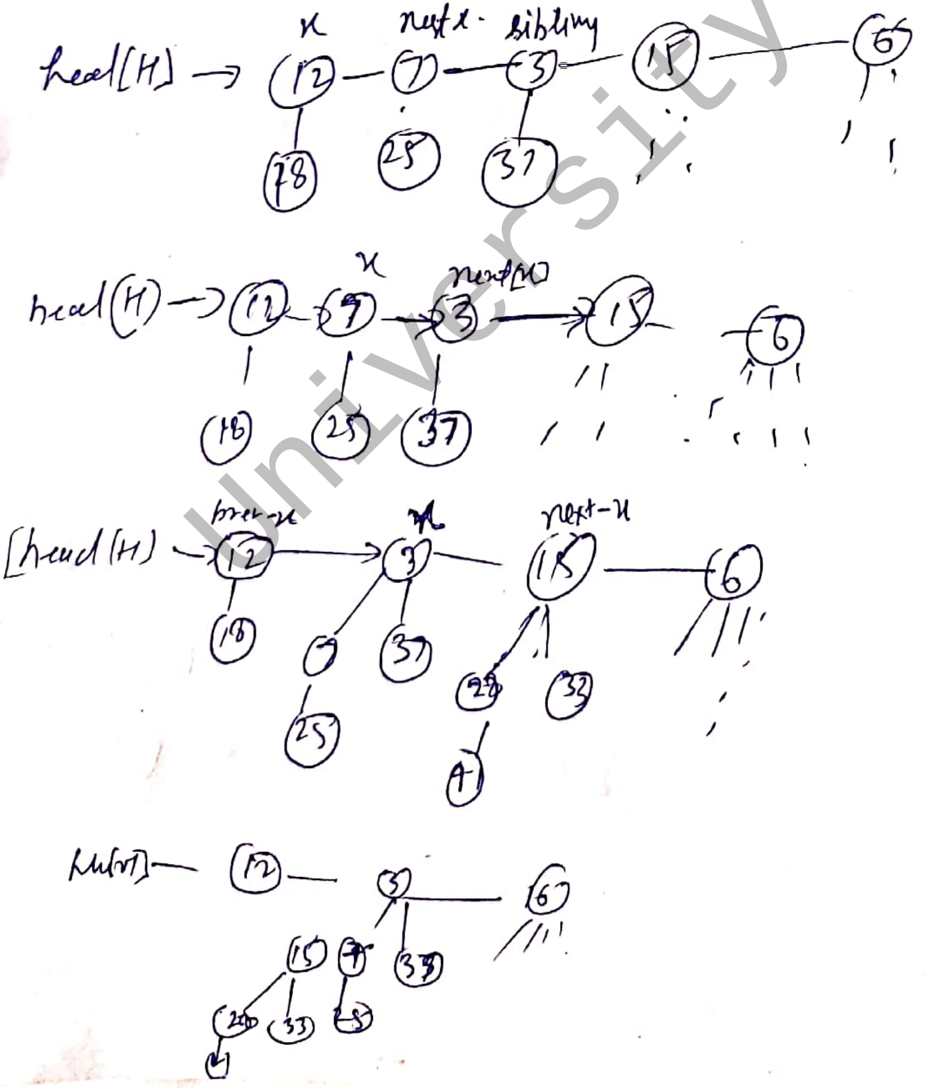
3- Union of two Binomial heap.





merge in increasing order of root degree





Case IV if degree (x) = degree[next x] ≠ degree sibling[next x] and key[x] ≤ key[next x] then remove x from root list attach it next x and B_{k+1} created.

Case I if degree[x] ≠ degree[next x] then move the pointer ahead.

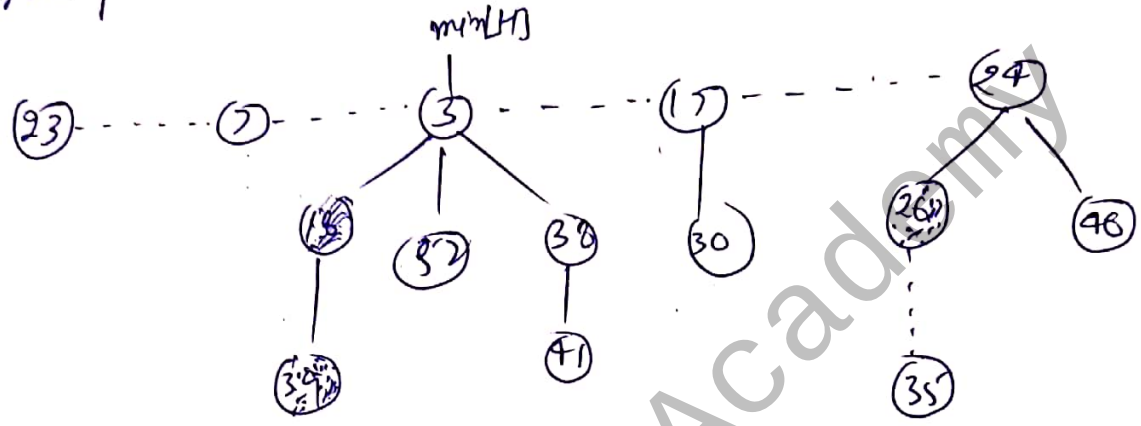
Case II if degree[x] = degree[next x] = degree[sibling[next x]] then again move pointer ahead.

Case III if degree[x] = degree[next x] ≠ degree[sibling[next x]] and key[x] ≤ key[next x] then remove next-x from root and attach it to x in order to create B_{k+1} tree.

Fibonacci Heap

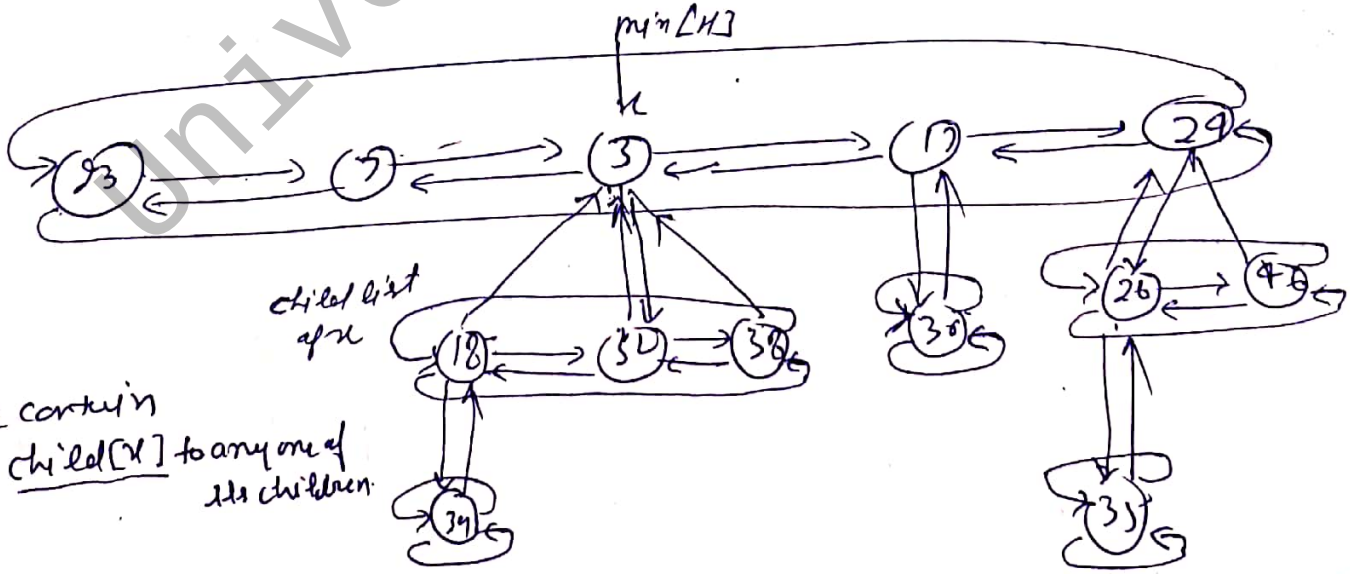
The Fibonacci heap is a collection of min-heap ordered trees. the tree in fibonacci heap are not constrained to be binomial tree operation Insert, find minimum, decrease key, and merge (union) work in constant O(1), operation delete and delete minimum work in O(log n) time.

Example of Fibonacci heap shown in figure



- unlike binomial heap the tree in fibonacci heap does not have exactly 2^k nodes
- tree within Fibonacci heap are rooted but unordered.
- roots and sibling lists kept as circular doubly linked list.

~~Allows constant time deletion~~



Node x contain p(x), child[x] to any one of its children

* No. of children in child list of node x is stored in degree[x].
* each node has mark[x], a boolean field indicating whether x has lost a child since the last time x was made a child of another node
* newly created node is unmarked.
* Fibonacci heap H is ~~associated~~ accessed by a pointer min[H] to the root of tree containing minimum key if min[H] = nil then Fibonacci heap is empty
* n[H] keep number of node in Fibonacci heap.

Potential function Analyze the performance of Fibonacci heap

$$\Phi(H) = t(H) + 2m(H)$$

t(H): no. of tree root; m(H): number of marked node

$\Phi(H) = 5 + 2 \times 3 = 11$

Fibonacci heap Operations.

1.	Creating a new Fibonacci heap	O(1)
2.	Inserting a node	O(1)
3.	Finding minimum node	O(1)
4.	Uniting two Fibonacci heaps	O(1)
5.	Extracting the minimum node	O(log n)
6	Decreasing a key	O(1)
7.	Deleting a node	O(log n)

① Creating a new Fibonacci heap:

→ the operating creating a new heap creates empty fibonacci heap

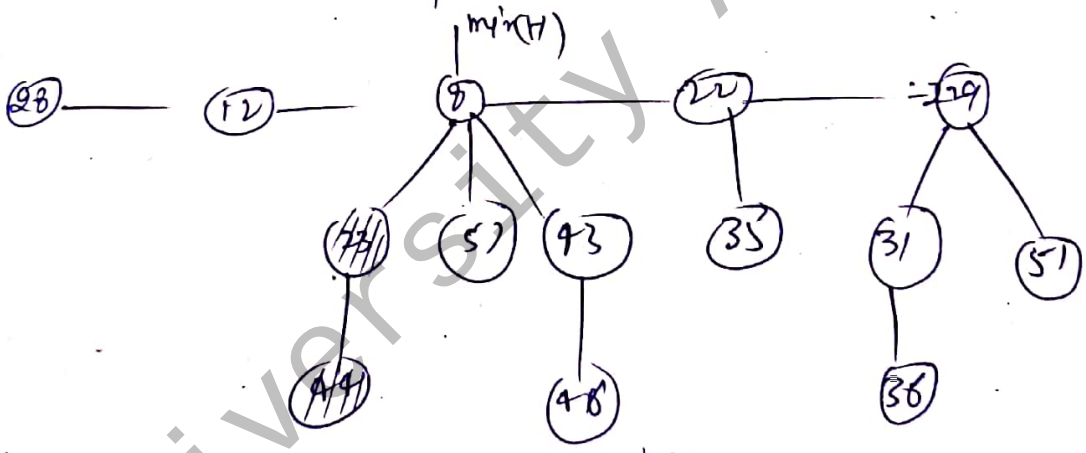
eg: min[H] → NIL

n(H) = 0
ϕ(H) = 0
t(H) = 0
m(H) = 0

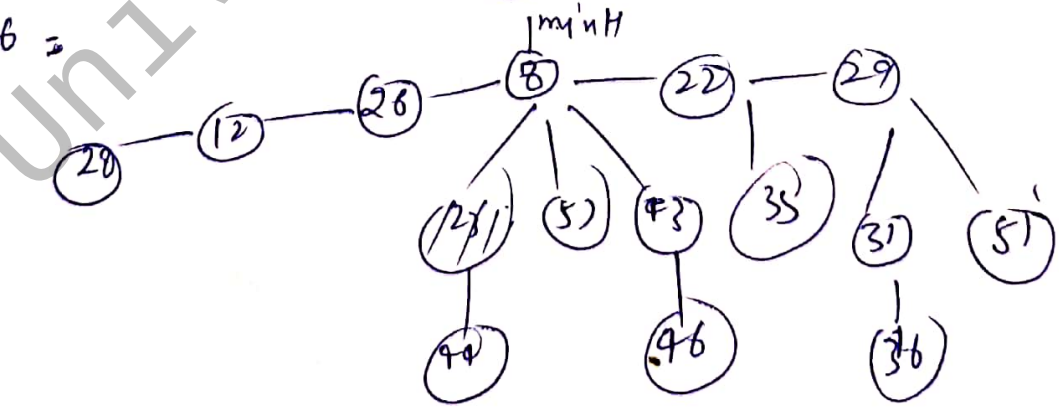
Inserting a node-

Create a new singleton tree and add it to the left of min[H] pointer

t(H') = t(H) + 1
m(H') = m(H)
ϕ(H') = ϕ(H) + 1



insert 26 =

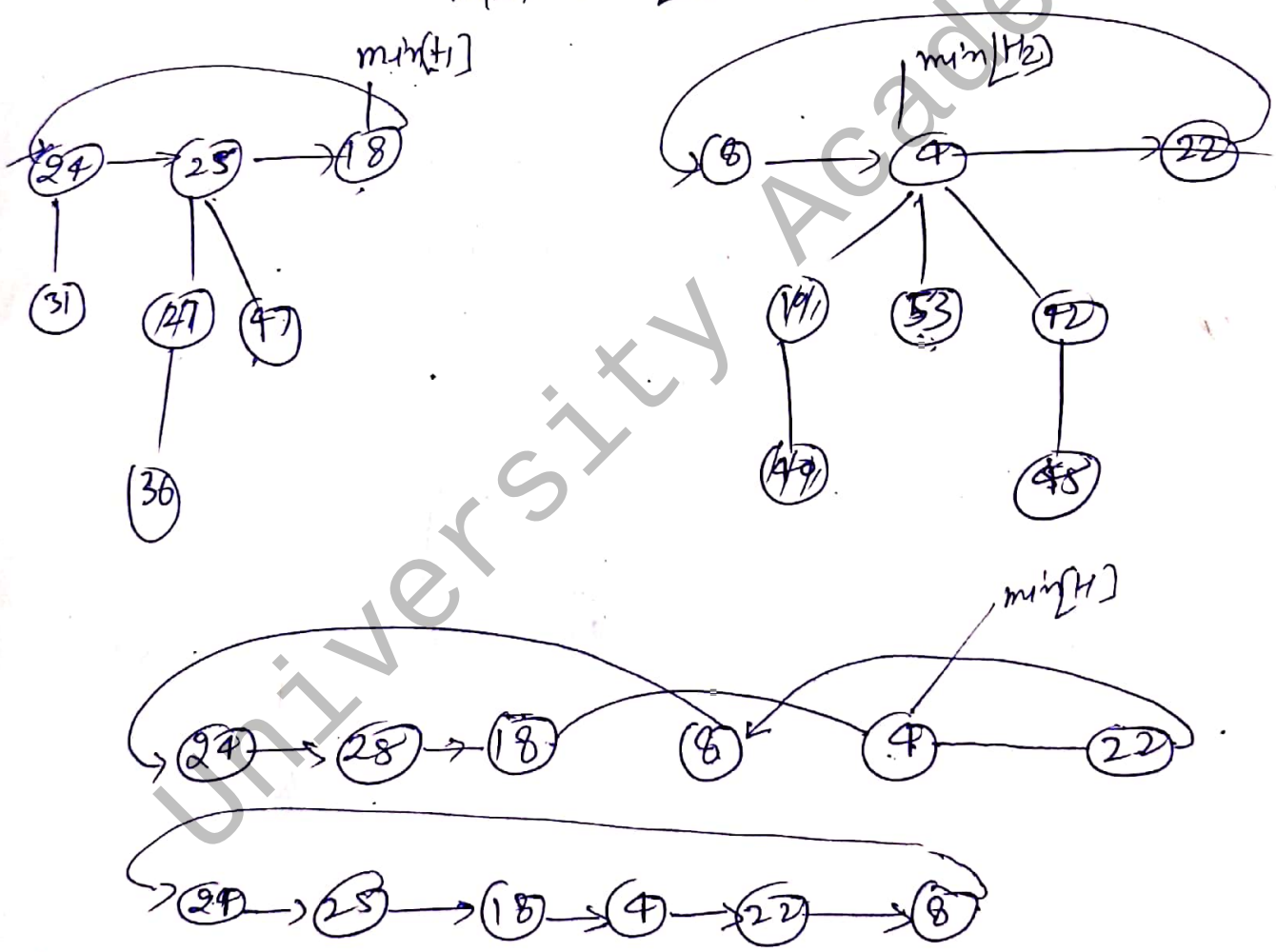


Finding minimum :-> minimum node in Fibonacci heap (H) is given by pointer min[H]. It always point to minimum key.

Uniting two fibonacci heap.

the operation simply concatenates the root list of H_1 and H_2 and maintains the new min[H] pointer

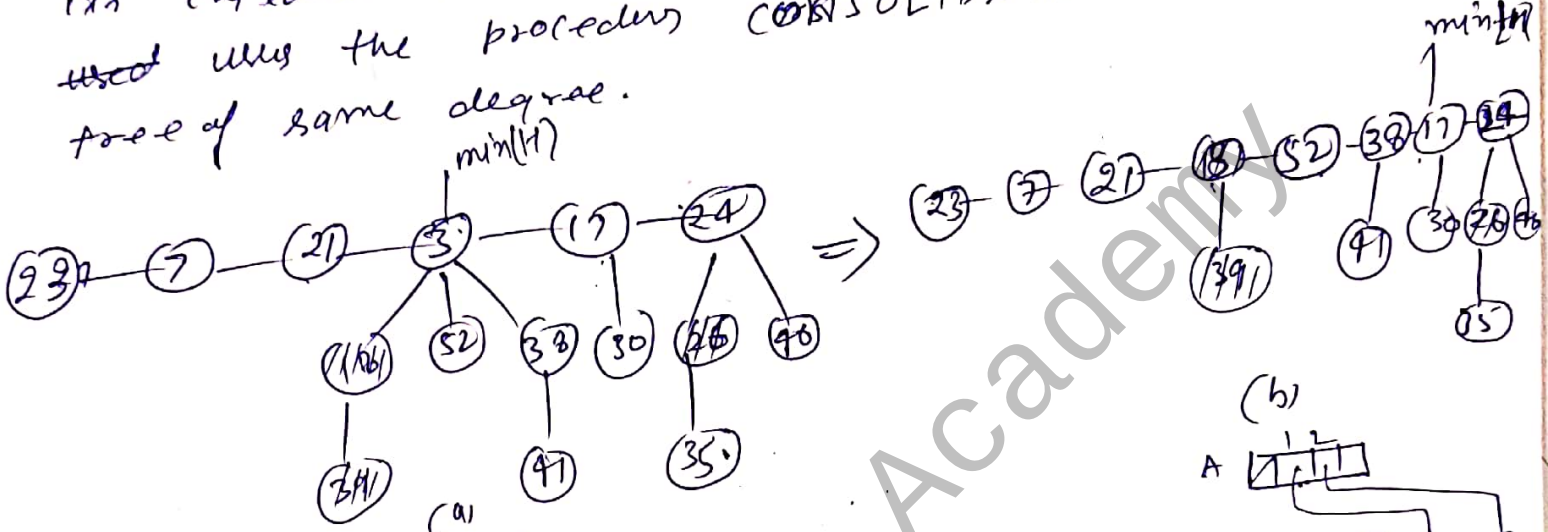
$$\phi[H] = \phi[H_1] + \phi[H_2]$$
$$t(H) = t[H_1] + t[H_2]$$
$$m(H) = m[H_1] + m[H_2]$$

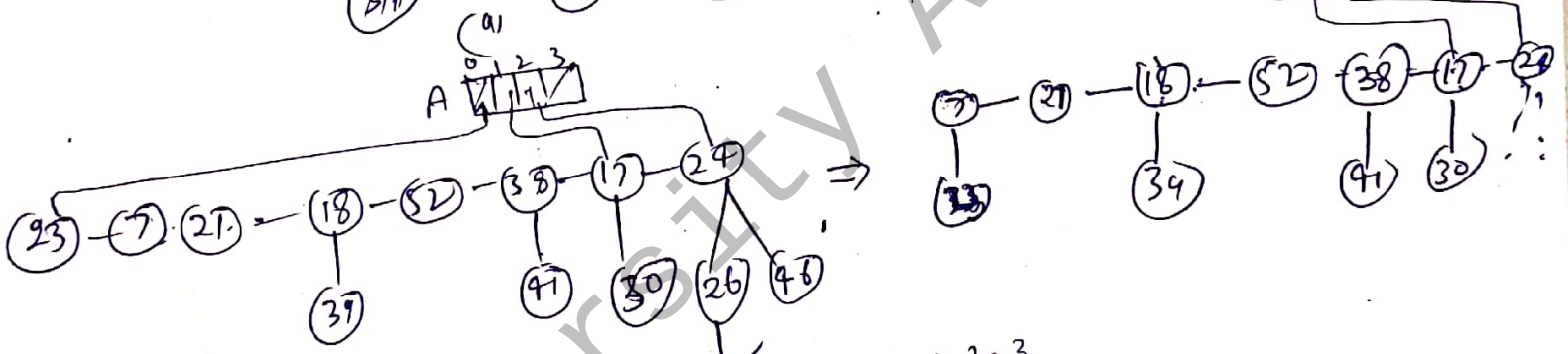


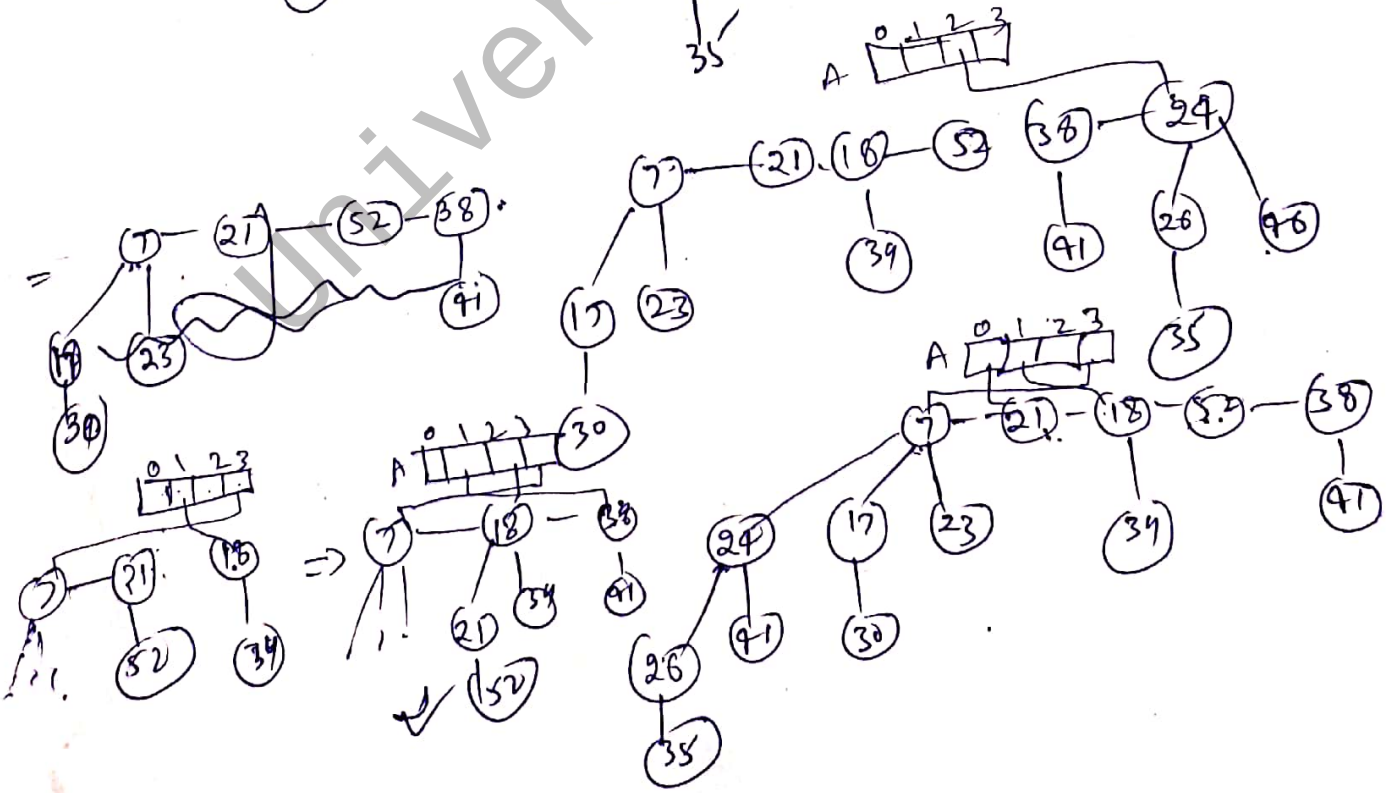
Extract min[H]

The process of extracting the minimum node is the most complicated of the operations presented in this section.

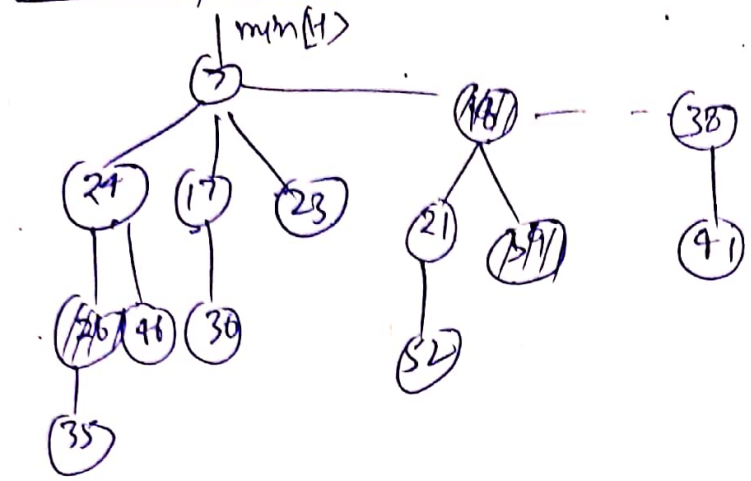
The operation is accomplished by first deleting the minimum key node and then moves all its children to root list of Fibonacci heap. It also uses the procedure CONSOLIDATE to merge the trees of same degree.





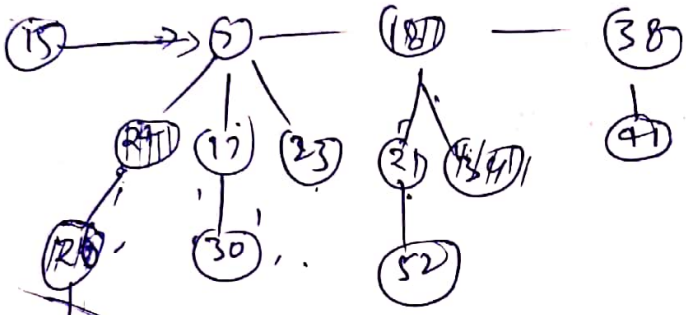


Decreasing key:



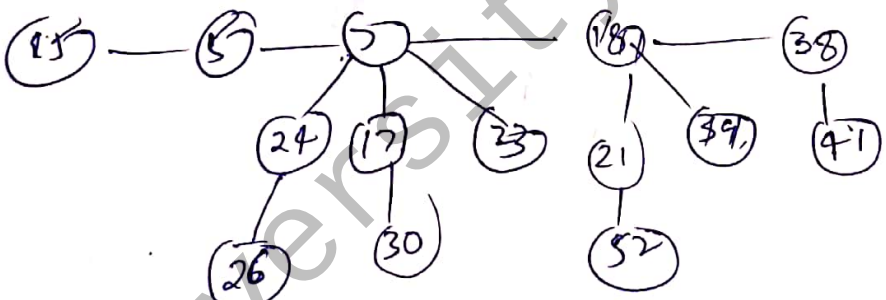
decrease case key 46 to 15

(1) insert key k
(2) cut the x from tree
(3) if parent of x is un marked, marked it
else: cut parent and mel insert to tree and unmarked

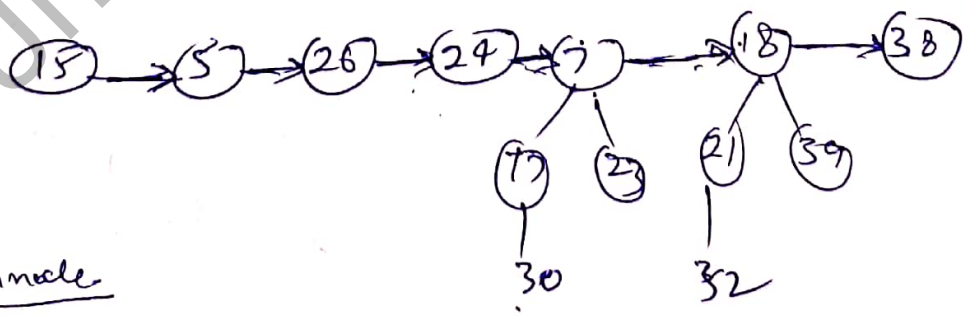


decrease 35 to 5

Cascading cut because Parent is marked



Cascading cut.



Deleting a node

(1) Fib-heap decrease key to(-∞)
(2) Fib-heap-extract min(H)

Trie (digital tree or Prefix tree):

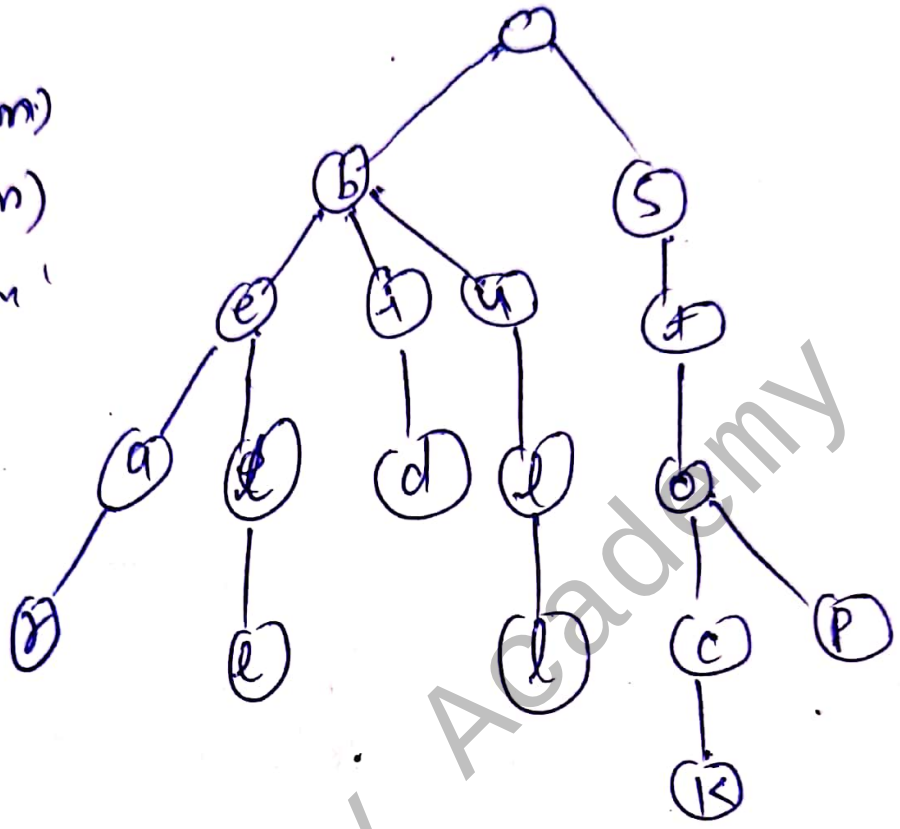
Tries are a special and useful data structure that are based on prifix of a string. They are used to Represent the Retrieval of data and thus the name Trie.

Tries consist of node and edge each node consist of 26 children and edge to connect each parent node to children. Here 26 children are aletter of english Alphabets.

example → Tree
→ stores a set of string
→ every node (except root) will store a letter of Alphabet.

ex $S = \{$ bear, bell, bid, bull, stock, stob $\}$

Search - O(m)
Insert O(m)
delete O(m)



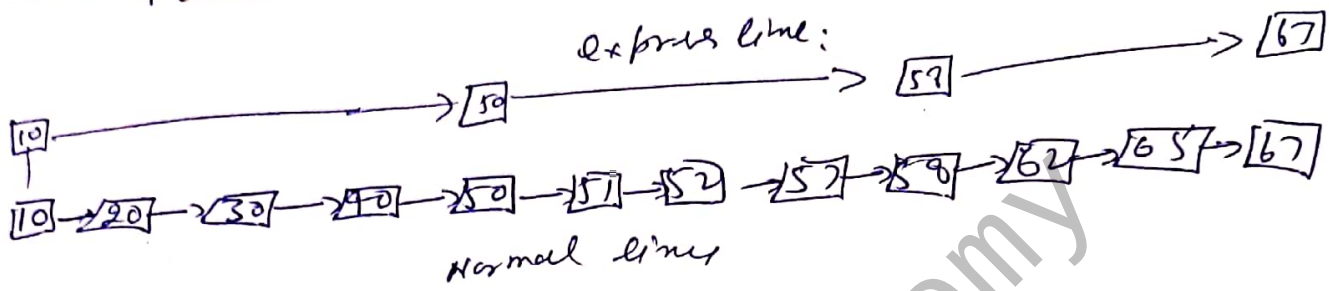
Application

1. ~~dictionary~~ dictionary.
2. Spell checker
3. Prefix matching
4. Network Browser history

Skip list

The skip list is a probabilistic data structure that is build subsequent layer of linked list upon an original linked list. each additional layer of links s contain fewer element. that not new.

eg.



time complexity

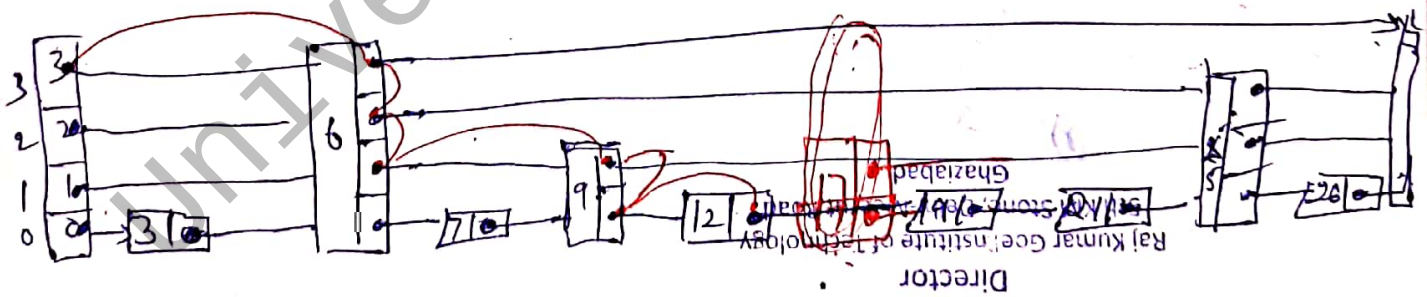
linear search . O(n)

Binary search O(log n)

Skip list → time complexity depend on number of layer. in average case time complexity for search, Insert delete O(log n)

Insertion

insert - 17



Search → 12

Delete: 16 . Reduce one level:

Divide and Conquer

- Divide and conquer is a top down technique for designing algorithms that consist of dividing the problem in following major phases.

1- Divide: Breaking the Problem into sub-problem that are similar to original problem but smaller in size.

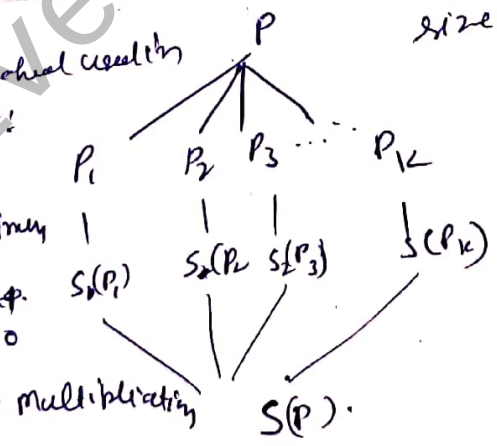
2- Conquer: solve the sub-problem Recursively

3. Combine: combine these solution to subproblems to create a solution to the original problem.

```
Algo DAndC(P)
{
  if Small(P) then return S(P)
  else
    {
      divide P into smaller P1 P2 ..... Pk  k >= 1
      Apply DAndC to each subproblem: Recursively
      return Combine(DAndC(P1), DAndC(P2) ... DAndC(Pk))
    }
}
```

Divide and Conquer Approached used in following problem:

1- Binary Search
2- Finding Minimum-Maximum
3. Merge sort Lecture-24
4- Quick sort Lect-20
5. Strassen's Matrix Multiplication
6. Convex-Hull problem.



Binary Search.

```
Algorithm BinSrch(a, i, l, x)
{
if (l==i) then //small p
  { if (x == a[i]) then Return i;
    else Return 0;
  }
else
  { // Reduce p into sub problem.
    mid = ⌊(i+l)/2⌋ ;
    if (x == a[mid]) then return mid;
    else
      if (x < a[mid]) then
      Return BinSrch(a, i, mid-1, x)
      else Return BinSrch(a, i+1, l, x);
  }
}
```

T(n)

1 — T(n/2)

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

master method

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

example.

a

1	2	3	4	5	6	7	8	9	10	11	12	13	14
-15	-6	0	7	9	23	54	82	101	112	125	131	142	151
i					mid								l

x=151

if(x == a[mid]) = (151 == 54) Falls.

if

x=151

i	l	mid
1	14	7
8	14	11
12	14	13
14	14	14
		Found

x=-14

i	l	mid
1	14	7
1	6	3
1	2	1
2	2	Not found
2	1	Not found

Finding ~~Minimum & Maximum~~ (Maximum & minimum)

(Problem to find maximum and minimum item in the set of n elements)

```
Algo MaxMin(i, j, max, min)                    T(n)
{
if (i==j) then max := min := a[i]  // small(P)
else if (i = j-1) then                 // small P)
    { if (a[i] < a[j]) then
            max = a[j]; min = a[i]
      else
            max = a[i]; min = a[j];
    }
else
 { // not a small P
   mid := [(i + j)/2];
   MaxMin(i, mid, max, min)
   MaxMin(mid+1, j, max1, min1)
   if (max < max1) then max := max1;
   if (min > min1) the min = min1;
   }
}
```

T(n/2)
T(n/2)
1

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

O(n)

$T(n) = aT(\frac{n}{b}) + f(n)$

$= a = 2 \quad b = 2 \quad f(n) = 1$

$n^{\log_b a} = n^{\log_2 2} = n$

$f(n) = 1$.

Case 1 $f(n) < n^{\log_b a}$

$T(n) = O(n^{\log_b a})$

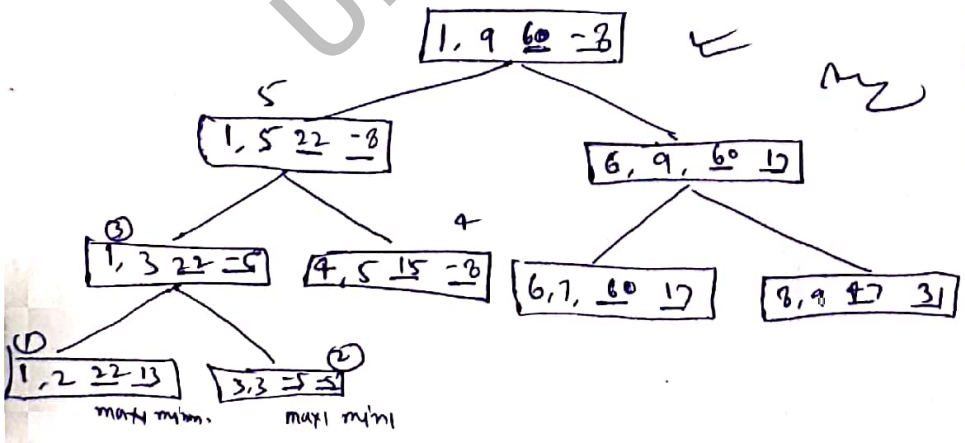
$= O(n)$

Ans

a

1	2	3	4	5	6	7	8	9
22	13	-5	-8	15	60	17	31	47

i J



STRASSEN'S MATRIX MULTIPLICATION:

1- Conventional method

2- Divide and conquer strategy

3. Strassen's matrix multiplication strategy

1. Conventional method.

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \times B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{bmatrix} = C = \begin{bmatrix} C_{11} & C_{12} \\ C_{21} & C_{22} \end{bmatrix}$$

2×2 $\quad$ 2×2 $\quad$ 2×2

$A \times B = C$

⇒ matrix multiplication (A B)

```
1. Multiply two n×n matrices A and B
2  n ← row[A]
3. let C be an n×n matrix.
3. for i ← 1 to n
4        do
5           for j ← 1 to n
6              do C[i,j] = 0
7              for k ← 1 to n
8                 do C[i,j] = C[i,j] + A[i,k] + B[k,j].
```

time complexity = $O(n^3)$.

2. Divide and Conquer strategy :-

Two $n \times n$ matrix, where $n = 2^k$ $k = 1, 2, 3, \ldots$ positive integer if n is not power of two then rows and columns of zero can be added. to but A and B.

$4 \times 4 = 4[n/2 \times n/2] = 4$ matrix 2×2

$C_{11} = A_{11} B_{11} + A_{12} B_{21}$

$C_{12} = A_{11} B_{12} + A_{12} B_{22}$

$C_{21} = A_{21} B_{11} + A_{22} B_{21}$

$C_{22} = A_{21} B_{12} + A_{22} B_{22}$

for 2×2 matrix:

$$\begin{bmatrix} A_{11} & A_{12} & A_{13} & A_{14} \\ A_{21} & A_{22} & A_{23} & A_{24} \\ - & - & - & \\ - & - & - & \\ - & - & - & A_{44} \end{bmatrix} \begin{bmatrix} B_{11} & B_{12} & B_{13} & B_{14} \\ B_{21} & B_{22} & B_{23} & B_{24} \\ - & - & - & \\ - & - & - & \\ - & - & - & B_{44} \end{bmatrix} = \begin{bmatrix} C_{11} & C_{12} & C_{13} & C_{14} \\ C_{21} & C_{22} & C_{23} & C_{24} \\ & & & \\ & & & \\ & & & C_{44} \end{bmatrix}$$

8 multiplication of $n/2 \times n/2$ matrices.
4 addition of $n/2 \times n/2$ matrices.

$$T(n) = \begin{cases} 1 & n \le 2 \\ 8\,T(n/2) + cn^2 & n > 2 \end{cases}$$

Recursive call — addition of two matrix

$a = 8 \quad b = 2 \quad \log_b a = \log_2 8 = 3$

$f(n) = n^2$

$n^k = n^2 \quad k < \log_b a$

Case 1 $O(n^3)$

```
Algo mm(A, B, n)
{
  if (n <= 2)
  {
    C11 = A11 x B11 + A12 x B21
    C12 = ...
    C21 = ...
    C22 = ...
  }
  else
  {
    mid = n/2
    mm(A11, B11, n/2) + mm(A12, B21, n/2)
    mm(A11, B12, n/2) + mm(A12, B22, n/2)
    ...
  }
}
```

---

3 Strassen's formula

This method compute 7 $n/2 \times n/2$ matrices P, Q, R, S, T, U, V as following

matrix multiplication of $n/2$ — 10

$$P = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$
$$Q = (A_{21} + A_{22}) \cdot B_{11}$$
$$R = A_{11} (B_{12} - B_{22})$$
$$S = A_{22} (B_{21} - B_{11})$$
$$T = (A_{11} + A_{12}) \cdot B_{22}$$
$$U = (A_{21} - A_{11}) \cdot (B_{11} + B_{12})$$
$$V = (A_{12} - A_{22}) (B_{21} + B_{22})$$

total
7 multiply and
10 + 8 addition & subtraction

$$C_{11} = P + S - T + V$$
$$C_{12} = R + T$$
$$C_{21} = Q + S$$
$$C_{22} = P + R - Q + U$$

8 :

$$T(n) = \begin{cases} 1 & n \le 2 \\ 7\,T(n/2) + cn^2 & n > 2 \end{cases}$$

master meth
$a = 7 \quad b = 2$
$\log_b a = \log_2 7 = 2.81$
$f(n) = n^2 = n^k$ ~~k matrix~~
$k < 2.81$
Case 1 $T(n) = O(n^{2.81})$

CONVEX-HULL Problem

→ Convex hull is the ~~line~~ is the smallest convex Polygon that contains all the point in the Plane.

→ given a set X of n points $P_1(x_1, y_1) \ldots P_n(x_n, y_n)$ in the plane we want to find convex hull of X.

→ The divide and conquer Algorithm takes O(nlogn) time to Compute convex hull in clockwise order

→ The problem reduce to finding upper and lower hull and then putting together.

→ Computer of upper and lower hull are similar because we ~~can~~ could Compute for upper hull.

→ Partition X into ~~two~~ nearly equal ~~halves~~. half. according x - coordinates. value.

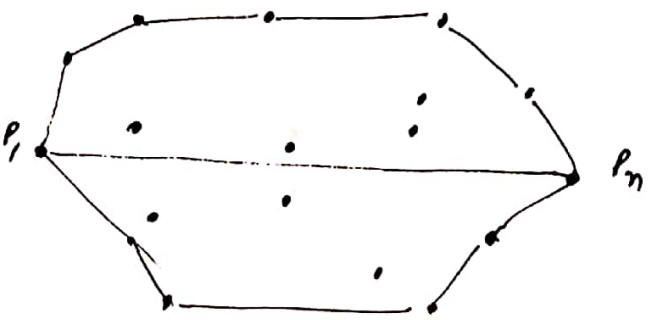
Process of finding upper hull.

① if X_1 is empty then upper hull is simply line with end point P_1 and P_n.

2. if X_1 is not empty the algorithm find P_{max} in X which is fastest from the line $\overline{P_1 P_n}$.

3. if there is tie then ~~max~~ point that maximum the Angle $\angle P_{max} P P_n$ can be selected.

4- Now Algorithm identifies all the points of X_1 that are left of line $\overline{P_1 P_{max}}$ goto step 1.



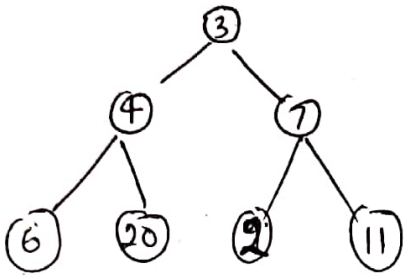
Greedy Algorithm.

A greedy algorithm is a strategy that makes the best optimal choice at each stage. with hope of finding a global optimal

Pros - simple, easy to implement, run fast.

cons: this algorithm do not always yield optimal solution.

finding the path with maximum sum and minimum sum



optimal solution.

minimum sum.

greedy approach.

3+4+6 = 13

actual: -

3+7+2 = 12

→ maximum sum:

greedy approach:

3+7+11 = 21

actual: 3+4+20 = 27

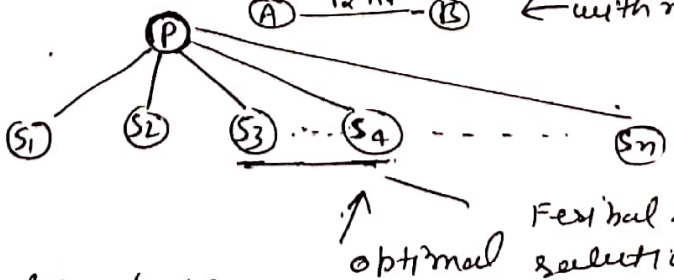
Note - greedy Algorithm always mak a choice that looks best at the moment.

optimization problem: optimization problem are those problem which objective to maximize or minimize some value. for exm

1- finding the shortest path between to vertices.

2- Select the maximum no. of activity performed by a person.

start time & Finish time.



← with minimum cost.

Feasible solution.

optimal solution.

optimal optimization problem is the problem of finding best solution from all feasible solution.

```
Algo Greedy(a, n)
   // a[1...n] contain n input
{
 Solution = 0;
 for i=1 to n do
    {
      x = select(a);
      if Feasible(solution, x) then
            solution Union(Solution, x)
    }
   return solution;
}
```

Application of Greedy Algorithm

1- Activity selection problem
2- Huffman coding
3- Job sequencing problem
4- Fractional knapsack Problem.
5- minimum spanning tree
6 single source shortest path.

Activity Selection Problem:

There are n different activity are given with their starting and ending time. Select the maximum no of activity to solve by a single person. greedy method will give optimal selection always:

1- sort the activity with their finishing time.
3- find compatible activity and add it ~~after~~ ~~select~~ list.

```
Greedy-Activity-selector (S, f)
1.   n ← length[S]
2.   A ← {1}
3.   j ← 1
4    for  i ← 2 to n
5            do if  s_i ≥ f_j
6                  the  A ← A ∪ {i}
7                       j ← i
8  return A
```

$S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

$s_i = \{1, 2, 4, 1, 5, 8, 9, 11, 13\}$

$f_i = \{3, 5, 7, 8, 9, 10, 11, 14, 16\}$

	1	2	3	4	5	6	7	8	9
s_i	1	2	4	1	5	8	9	11	13
f_i	3	5	7	8	9	10	11	14	16

$A = \{1, 3, 6, 8\}$ optimal solution.

this is not only optimal schedule but $\{2, 5, 7, 9\}$ is also optimal.

time complexity: sorting time $n \log n$. and other part take n time.

so total is $n \log n + n = O(n \log n)$

Question Given 10 activity along with their start and end time:

$S = \{A_1, A_2, A_3, A_4, A_5, A_6, A_7, A_8, A_9, A_{10}\}$

$S_i = \{1, 2, 3, 4, 7, 8, 9, 9, 11, 12\}$

$f_i = \{3, 5, 4, 7, 10, 9, 11, 13, 12, 14\}$

Step 1. Sort activity with increasing order of end time.

A_1	A_3	A_2	A_4	A_6	A_5	A_7	A_9	A_8	A_{10}
1	3	2	4	8	7	9	11	9	12
3	4	5	7	9	10	11	12	13	14

Ans $A = \{A_1, A_3, A_4, A_6, A_7, A_9, A_{10}\}$ optimal.

Huffman Coding

Huffman codes are used to compress the data upto 90%.

depending on the ~~the~~ ~~of~~.

suppose we have 100,000 character data having 6 different characters

	A	B	C	D	E	F
	45000	13000	12000	16000	9000	5000
ASCII code (8bit)	01000001	01000010	11	100	...101	110
fixed-length 3bit code	000	001	010	011	100	101
variable length	0	101	100	111	1101	1100

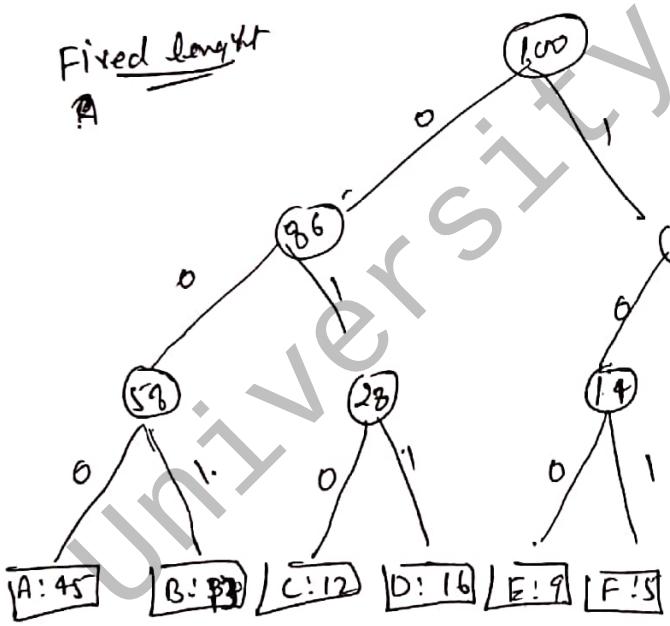
= 100K × 8 = 800K

= 100K × 3 = 300K bit

= 45K×1 + 13K×3 + 12K×3 + 16K×3 + 9K×4 + 5K×4 = 224000 bit

Huffman coading (Variable length)

Fixed length



A 45

B 13

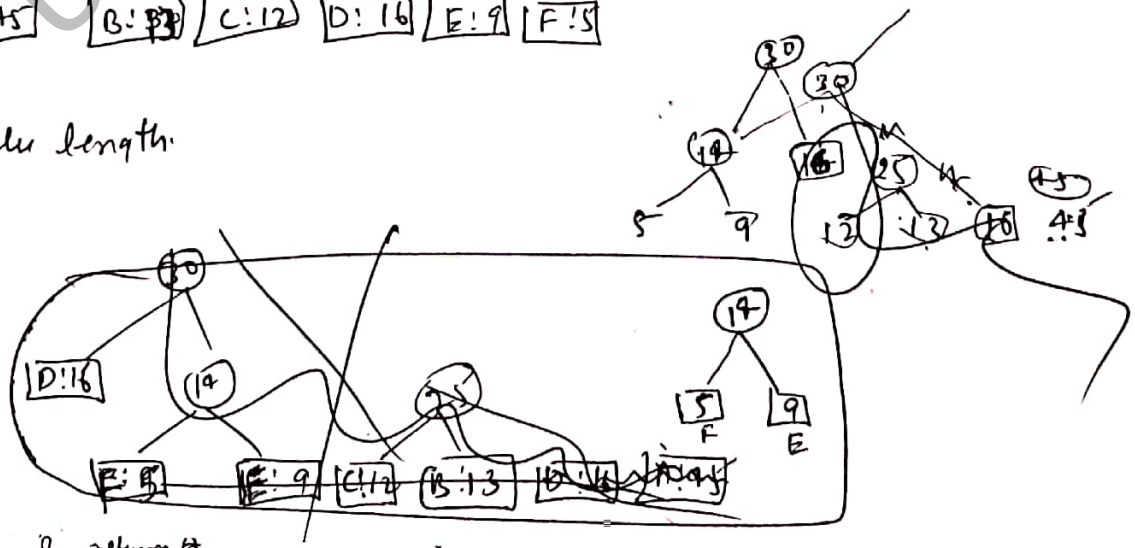
C 12

D 16

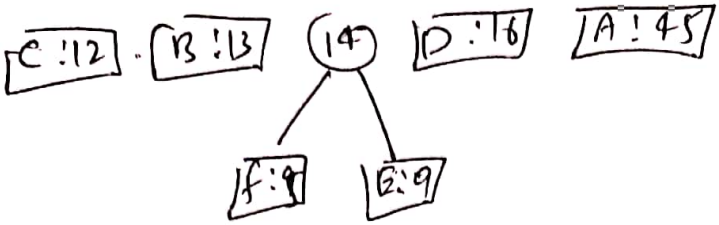
E 9

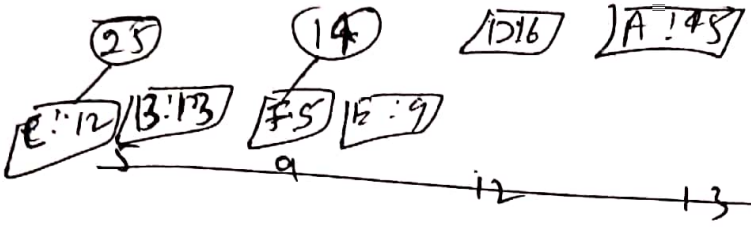
F 5

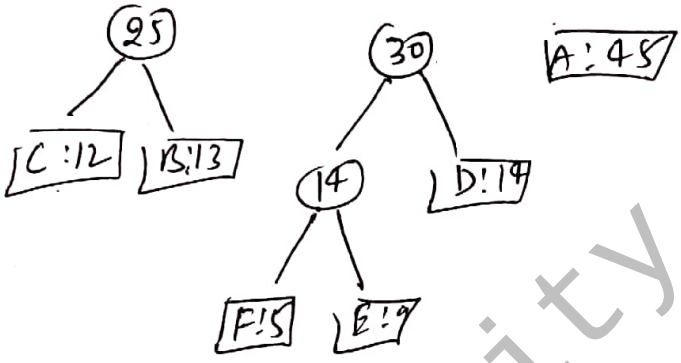
Variable length

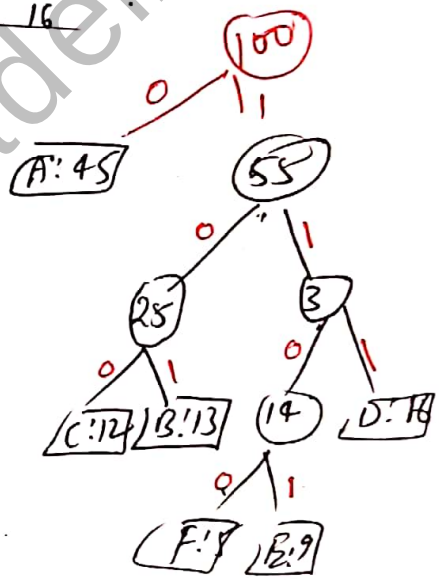


F:5 | E:9 | C:12 | B:13 | D:16 | A:45









time complexity (n log n)

- Extract min within loop executing n times.

(log n) * n

O(n log n)

Job Sequencing with Deadline.

we are given a set of n Jobs, each a Job, are associated with Profit $P_i > 0$ and deadline of completion $d_i \geq 0$. Profit is earned iff it is completed within deadline. Assum ~~the machine~~ that each Job take one unit of time to complet it.

$n = 4$

Jobs	J_1	J_2	J_3	J_4
Profit	P_1	P_2	P_3	P_4
	100	10	15	27
deadline	d_1	d_2	d_3	d_4
	2	1	2	1

mach unit time

maximization problem by Greedy method:

① sort Job by Profit in non-increasing order.

P_1 P_4 P_3 P_2
100 27 15 10

0 J_4 1 J_1 2

$(J_4\ J_1)$ optimy

100 + 27 = 127 profit

Cyber cafe example

feasible	Selection	sequence	
1	(1)	1 (Job)	100
2	(2)	(2)	10
3	(3)	3	15
4	(4)	4	27
5	1,2	2, 1	110
6	1,3	1, 3 or 3, 1	115
7	1,4	4, 1	127 optimal
8	2,3	2, 3	25
9	2,4 ×		
10	3,4	4, 3	42

Question

n=5

Job $(J_1 \cdots J_5)$ = 1, 2, 3, 4, 5

Profit $(P_1 \cdots P_5)$ = 10, 15, 10, 5, 1

deadline $(d_1 \cdots d_5)$ = 2, 2, 1, 3, 3

Knapsack Problem (fractin)

we have n object and a knapsack (bag). each object have some profit (P_i) and weight (w_i). the abacity of bag is m. the problem is to filling bag with object that maximize total profit earned.

Consider $n = 3$, $m = 20$

Profit $(P_1, P_2, P_3) = (25, 24, 15)$ and weight $(w_1, w_2, w_3) = (18, 15, 10)$

object	1	2	3
Profit	25	24	15
weight	18	15	10
P/w	1.38	1.6	1.5

$m = 20$.

greedy ability =

	(x_1, x_2, x_3)	$\sum w_i x_i$	$\sum P_i x_i$
1- maximum profit first	$(1, 2/15, 0)$	$18+2=20$	$25+3.2=28.2$
2- minimum weight first	$(0, 2/3, 1)$	$10+10=20$	$16+15=31$
3- randomly =	$(1/2, 1/3, 1/4)$	$9+5+2.5=16.5$	$12.5+8+3.75=24+25$
4= Profit / Weight (maximum first)	$(0, 1, 1/2)$	$15+5=20$	$24+7.5=31.5$ obtimal.

Fractionable item = Vegetables, grocery

Non fractionable item = Electronic item. eg: washing machine, AC, Refrigerator

1- $\frac{2}{15}$: $\frac{8}{24} \times 2 = \frac{16}{5} = 3.2$

2- $\frac{10}{15}$: $\frac{8}{24} \times 10 = 8 \times 2 = 16$

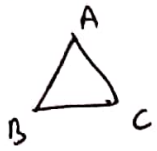
Q. $n = 7$, $m = 15$, $(P_1, P_2, P_3 \cdots P_7) = (10, 5, 15, 7, 6, 18, 3)$

$(w_1, w_2 \cdots w_7) = (2, 3, 5, 7, 1, 4, 1)$

Ans 54.6

Minimum spanning tree.

Spanning tree: A spanning tree is a subset of Graph G which has all the vertices covered with minimum possible number of edges. Hence tree does not have any cycle and it can't be disconnected.

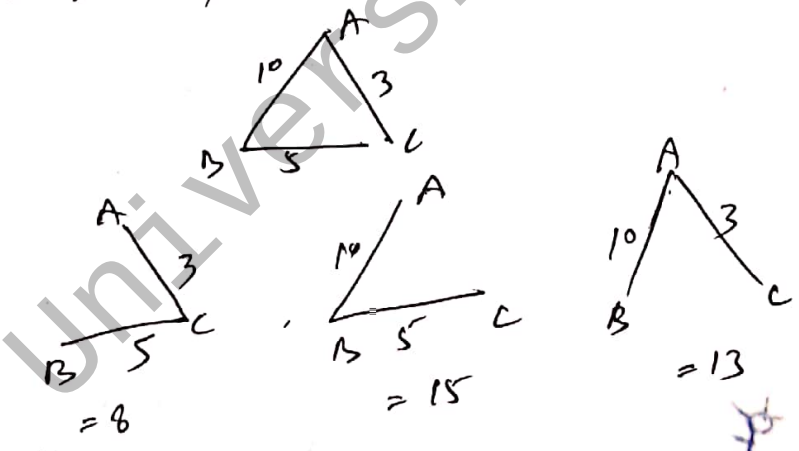


spanning tree:

1. spanning has $n-1$ edges. where n is no. of node
2. Comble graph can have maximum n^{n-2} number of spanning tree.
3. All possible spanning tree have same no. of edges and vertices.

Minimum Spanning tree -

In a weighted graph a minimum spanning tree is a spanning tree that has minimum weight than all other spanning tree of same graph.



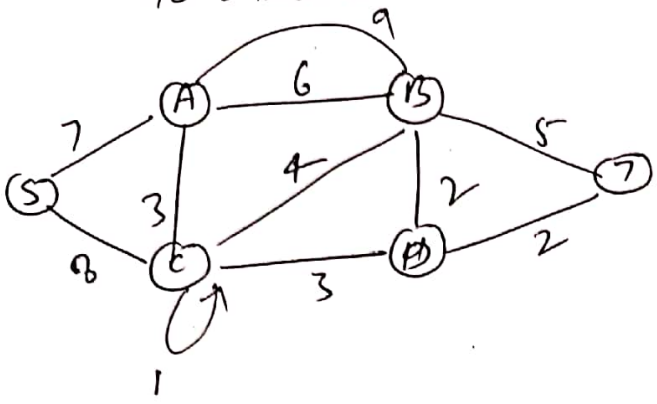
minimum spanning tree Algorithm.

1. Kruskal's Algorithm.
2. Prims Algorithm.

Kruskal's Algorithm

Kruskal's Algo to find minimum cost spanning tree uses greedy Approach.

to understand Kruskal Algo let us consider example :

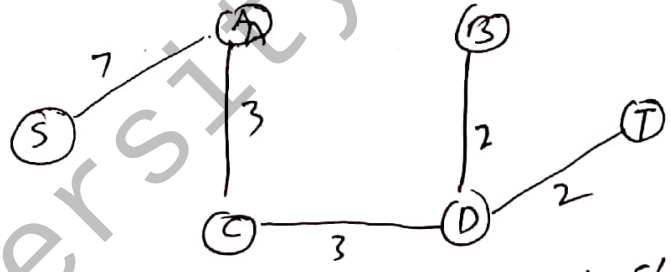


step-1. Remove all loops and parallel edge.

step-2 - Arrange all edges in their increasing order of weight.

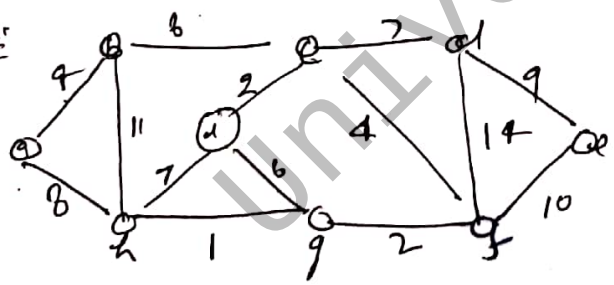
B,D	D,T	C,D	A,C	C,B	B,T	A,B	A,S	S,C
2	2	3	3	4	5	6	7	8
				×	×	×		

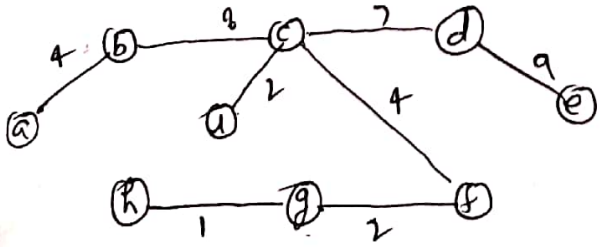
step-3 Add the edge which has the least weight



minimum spanning tree

Ex:






```
MST = KRUSKAL (G, w)                                    — 1
1- A ← ϕ                                                — |V|
2- for each vertex v ∈ V[G]
3       do MAKE-SET(v)                                  (Elog|E|)
4. sort the edges of into nondecreasing order by weight
5. for each edges (u,v) ∈ E taken in non-decreasing order by w  — |V|
6.      do if FIND-SET(u) ≠ FIND-SET(v)
7           then A ← A ∪ {(u, v)}
8           UNION (u, v)
```

Time complexity = E log E or E log V

E log V = E log E

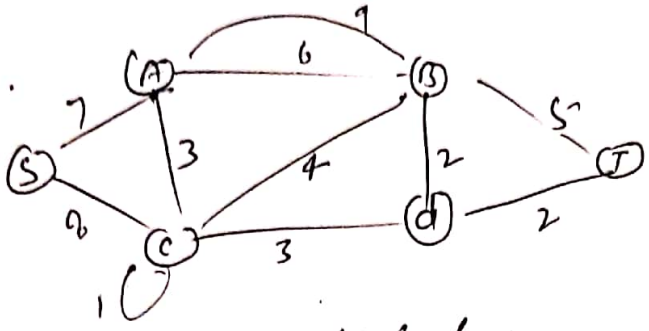
Total = 37 weight A = {(h,g), (i,c), (g,f), (a,b), (c,f), (c,d), (b,c)}

Prim's Algorithm

Prim's Algo to find minimum cost spanning tree uses greedy approach

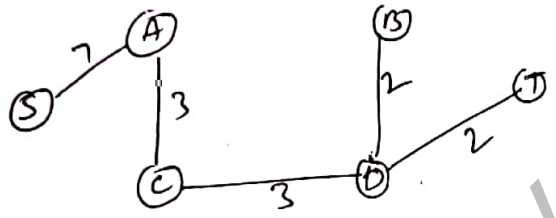
Prims algo treats the node as a single tree and keeps on adding new nodes to spanning tree.

Ex:

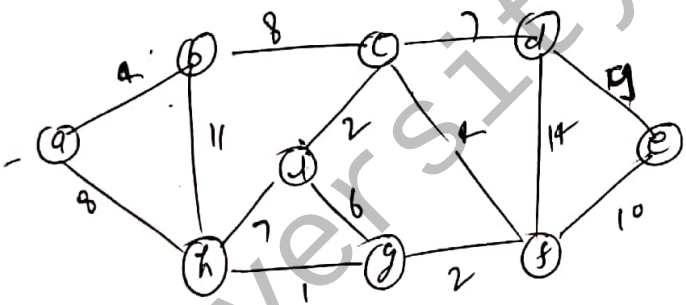


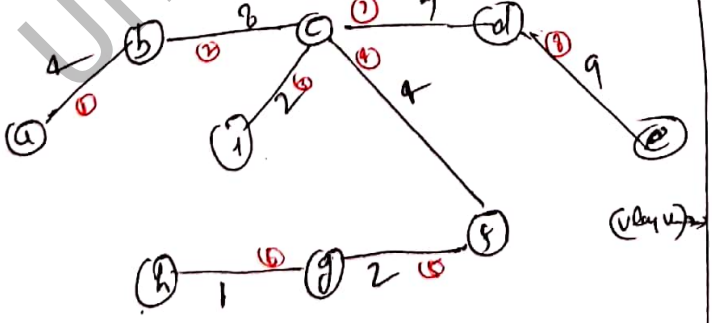
1- Remove loops and parallel edges.

2. Choose any arbitrary node as root node



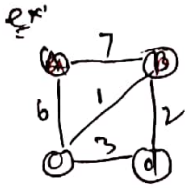
Qm

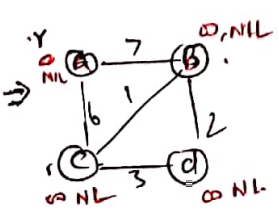


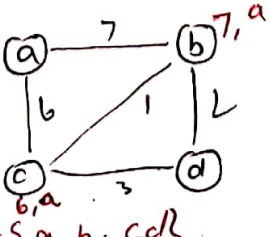



```
MST_PRIM(G, w, r)
1- for each u ∈ V[G]
2     do key[u] ← ∞
3        π[u] ← NIL
4  key[r] ← 0
5- Q ← V[G]
6- while Q ≠ ϕ
7     do u ← EXTRACT-MIN(Q)
8        for each v ∈ Adj[u]
9           do if v ∈ Q and w(u,v)
10                < key(v)
11             then π(v) ← u
                  key[v] ← w(u,v)
```

Ex:





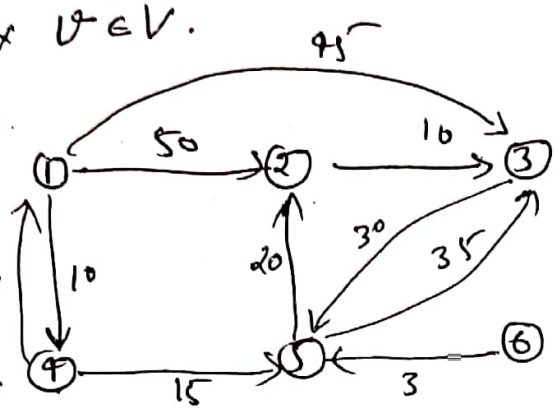


Q = {a, b, c, d}

SINGLE-SOURCE SHORTEST PATH

directed or undirected or mixed

Given a graph $G = (V, E)$, we want to find a shortest path from a given source vertex $s \in V$ to each vertex $v \in V$.



graph

Path	length
1) 1, 4 : (1, 4)	10
2) 1, 2 : (1, 4, 5, 2)	45
3) 1, 3 : (1, 3)	45
4) 1, 5 : (1, 4, 5)	25
5) 1, 6 : —	∞

shortest path from 1

Vertex -
edges
weight eges
→ Positive weight
→ Negative weight

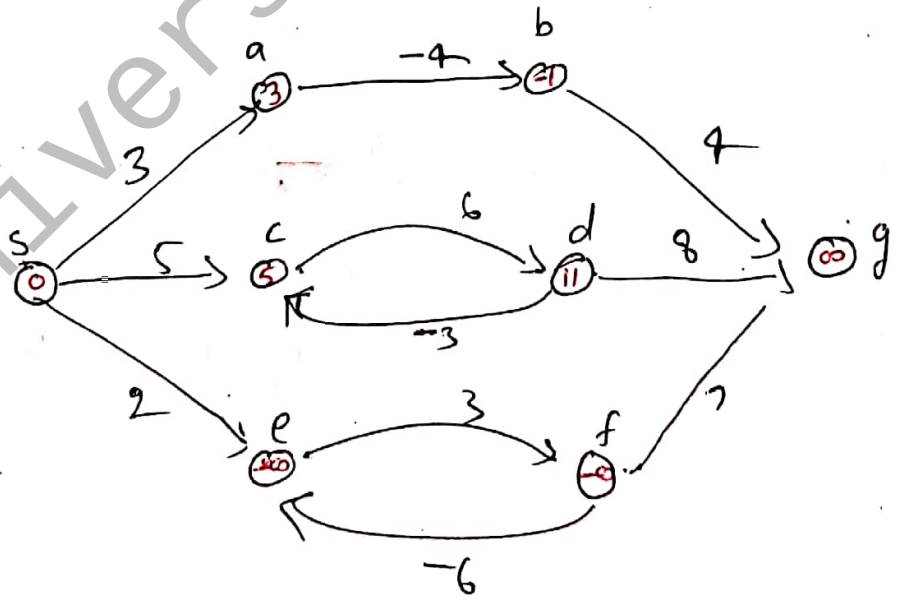
Cycle
→ Positive weight cycle
→ Negative weight cycle

There are to Algorithm to solve single source shortest path.

1- Bellman-Ford Algorithm. (weight may negative)

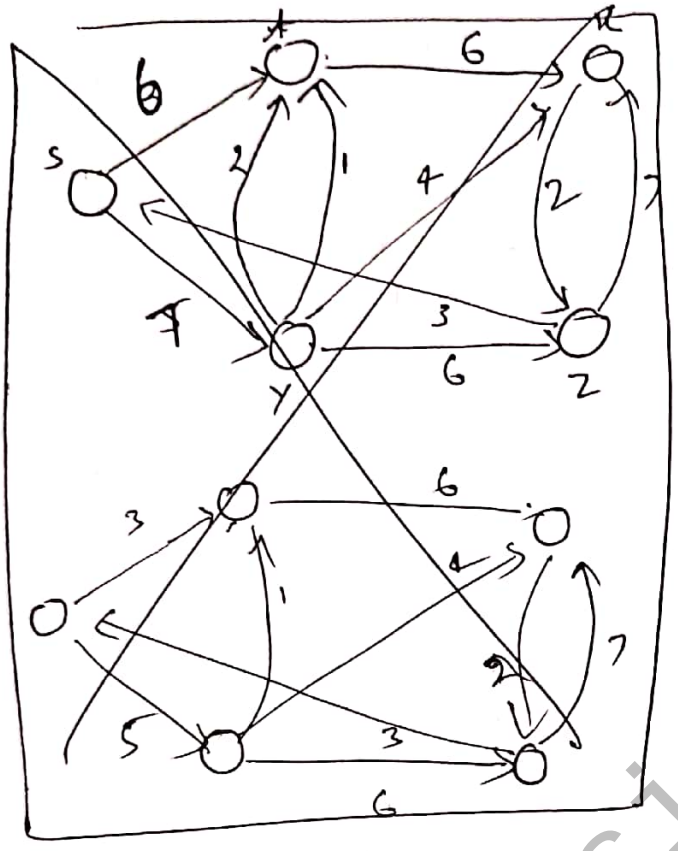
2. Dijkstra Algorithm (non-negative weight).

1- ~~Bellman-Ford Algorithm~~ :-



Bellman- Ford Algorithm

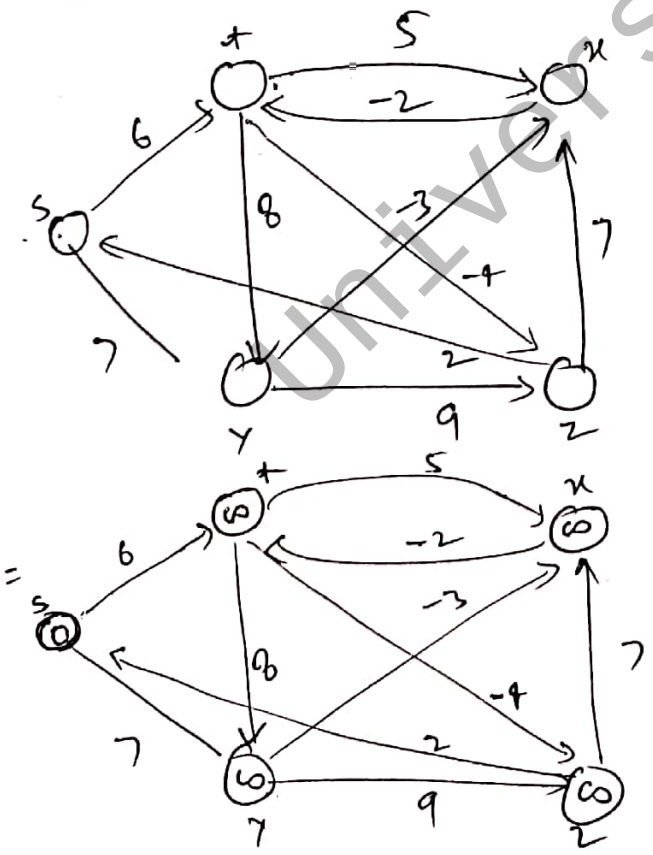
Bellman- Ford algorithm solve the single source shortest path problem in the general case in which weight may be negative but not negative weight cycle.

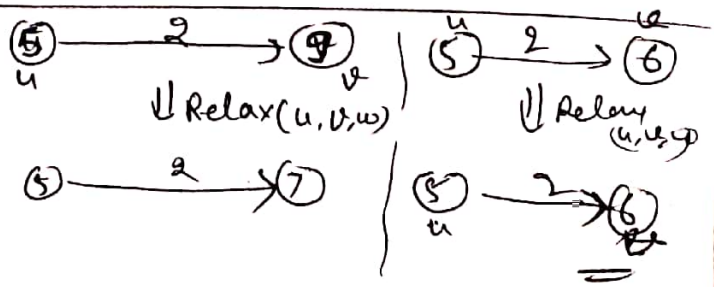



```
Bellman- Ford(G, w, s)
1. Initialize- single source(G,s)
2. for i ← 1 to |V[G]| - 1
3      do for each edge (u,v) ∈ E[G].
4           do Relax (u, v, w)
5. for each edge (u,v) ∈ E[G]
6      do if d[v] > d[u] + w(u,v)
7           then Return FALSE
8. Return TRUE:
```

```
Initialize single source (G,s)
1. for each vertex v ∈ V[G]
2      do d[v] ← ∞
3         π[v] ← NIL
4. d[s] ← 0
```

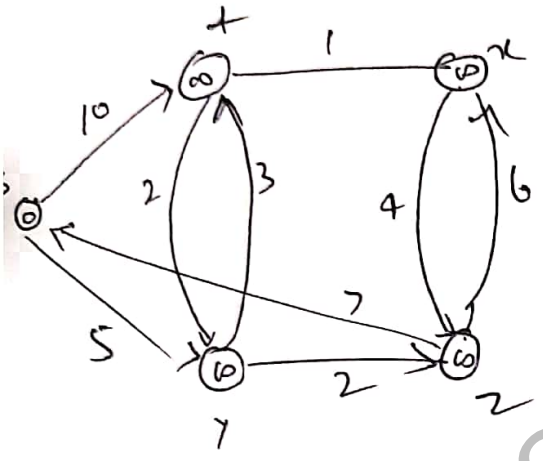
```
RELAX (u, v, w)
1. if d[v] > d[u] + w(u,v)
2      then d[v] ← d[u] + w(u,v)
3           π[v] ← u
```





Dijkstra's Algorithm.

Dijkstra algo solve the single source shortest-path problem on a weighted directed graph $G = (V, E)$ for the case in which all edge weight are non-negative.




```
DIJKSTRA (G, w, s)
1. Initialize-single-source (G, s)
2  S ← ϕ
3. Q = V[G]
4.  while Q ≠ ϕ
5       do u ← Extract-min (Q)
6          S ← S ∪ {u}
7          for each vertex v ∈ Adj[u]
8             do Relax (u, v, w)
```

Dynamic Programming -

Dynamic programming approach is similar to divide and conquer. in breaking down the problem into smaller subproblems. but unlike divide and conquer these subproblems are not solved independently.

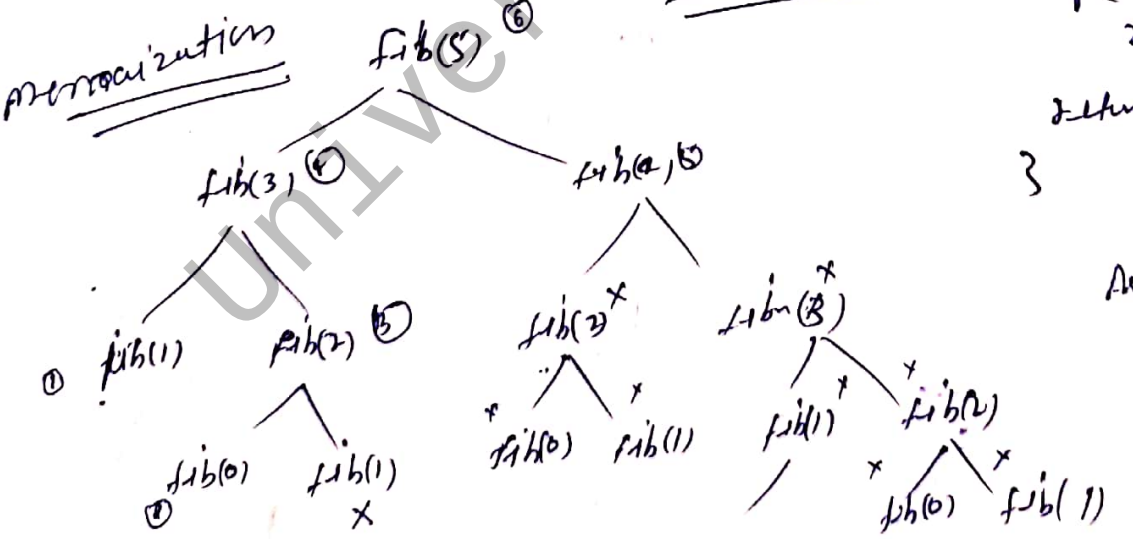
in the dynamic programming approach before solving the subproblem examine the result of previously solved sub problems.

Dynamic problem used in optimization problem.
(minimum / maximum)

Dynamic Programming is technique for solving problem recursively or nonrecursively (iterative) by memorization or tabulation method.

Example fibonacci :-

Memorization



Tabdownapproach.

```
int fib(int n)
{
  if(n<=1)
    return n;
  return fib(n-2)+fib(n-1);
}
```

Assume $(n-2) \cong (n-1)$

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

$= O(2^n)$

15 call

fib(5)

F

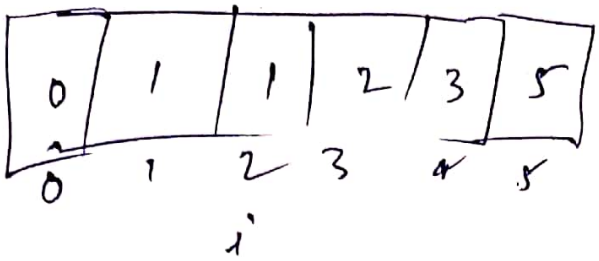
0	1	1	2	3	5
0	1	2	3	4	5

$T(n)$ = n+1 calls

$= O(n)$.

Iterative Tabulation - Bottom up -

```
int fib(int n)
{
  if (n <= 1)
     return n;
  F[0] = 0; F[1] = 1
  for (i = 2; i <= n; i++)
  {
     F[i] = F[i-2] + F[i-1]
  }
  return F[n]
}
```

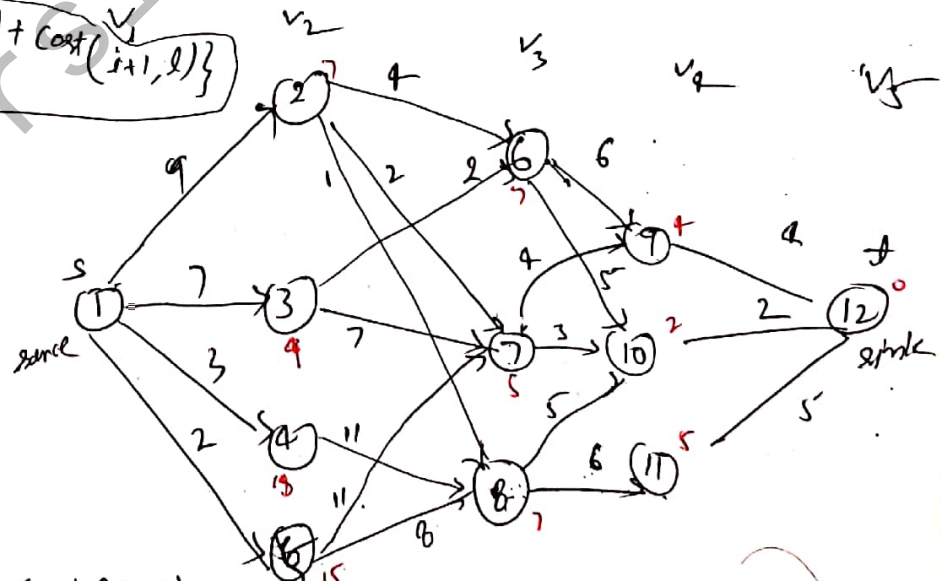


Multistage Graph (Tabulation method)

∴ the multistage graph is to find a minimum cost path from s to t.

(Forward approach)

$$cost(i, s) = \min_{l \in V_{i+1},\ \langle j, l \rangle \in E} \{ c(j, l) + cost(i+1, l) \}$$



stage / vertex

cost(5, 12) = 0

cost(4, 9) = 4

cost(4, 10) = 2

cost(4, 11) = 5

cost(3, 6) = min { 6 + cost(4, 9), 5 + cost(4, 10) }

= {10, 7} = 7

cost(3, 7) =

vertex	1	2	3	4	5	6	7	8	9	10	11	12
cost	16	7	9	18	15	7	5	7	4	2	5	0
d	2/3	7	6	8	8	10	10	10	12	12	12	12

0/1 Knapsack Problem

(1) Tabular method

(2) Sets method.

$m = 8$ $\quad P\{1, 2, 5, 6\}$

$n = 4$ $\quad w\{2, 3, 4, 5\}$

$x_i = 0/1 \; x\{1\ 0\ 0\ \cdots\}$

$\max \sum P_i x_i$

$\sum w_i x_i \le m$

Table

P_i	w	i \ K	0	1	2	3	4	5	6	7	8
		0	0	0	0	0	0	0	0	0	0
1	2	1	0	0	1	1	1	1	1	1	1
2	3	2	0	0	1	2	2	3	3	3	3
5	4	3	0	0	1	2	5	5	6	7	7
6	5	4	0	0	1	2	5	6	6	7	8

$K[i, w] = \max\{K[i-1, w], K[i-1, w - w(i)] + P(i)\}$

$K[4, 7] = \max\{K[3, 7], K[3, 7-5] + 6\}$

$= \max\{0, x\}$

$K[4, 3] = \max\{K[3, 3], K[3, 3-5] + 6\}$

$= \max\{2, x\} = 2$

$K[4, 5] = \max\{K[3, 5], K[3, 5-5] + 6\}$

$= \{5, 6\} = 6$

Set method

$S^0 = \{(0,0)\}$ (weight, profit)

$S_1^0 = \{\{1,2\}\}$ (profit, weight)

$S^1 = \{\{0,0\},\{1,2\}\}$; $S_1^1 = \{\{2,3\},\{3,5\}\}$

$S^2 = \{(0,0),(1,2),(2,3),(3,5)\}$

$S_1^2 = \{(5,4),(6,6),(7,7),(8,9)\}$

$S^3 = \{(0,0)(1,2),(2,3),(3,5)(5,4),(6,6),(7,7)\}(8,9)$

P_K, w_K P_J, w_J dominated

$S_1^3 = \{\{6,5\},\{7,7\},(8,8),$ Prof. weight increase profit increase.

$(11,9),(12,11),(13,14)$ discard lower profit by dominance rule

$S_0^4 = \{(0,0)(1,2)(2,3),(3,5),(5,4),(6,6)(6,5),(7,7),(8,8)\}$

(P_J, w_J) and (P_K, w_K)

$P_J \le P_K$ and $w_J \ge w_K$

discard (P_J, w_J)

Solution

$(8,8) \in S^4$ $x_4 = 1$

$(8,8) \notin S^3$

$(8-6, 8-5)$

$(2,3) \in S^3$

$(2,3) \in S^2$ $x_2 = 1$

$(2,3) \notin S^1$

$(0,0)$ --

$n=3$ $(P_1, P_2, P_3) = (1, 2, 5)$

$m=6$ $(w_1, w_2, w_3) = (2, 3, 4)$

$\rightarrow w$

$i \downarrow$	0	1	2	3	4	5	6
0	0	0	0	0	0	0	0
1	0	0	1	0	0	0	0
2	0	0	1	2	2	3	3
3	0	0	1	2	5	5	6

P	W
1	2
2	3
5	4

TF

$(6,6) \in 3$

$(6,6) \notin 2$ $x_3 = 1$

$(6-5, 6-4) = (1, 2)$

$(1,2) \in 2$

Partial:

$(1,2) \in 1$ $x_1 = 1$

$(1,2) \notin 0$

Set method:

$S^0 = \{(0,0)\}$: $S^0_1 = \{(1,2)\}$.

$S^1 = \{(0,0), (1,2)\}$ $S^1_1 = \{(2,3), (3,5)\}$

$S^2 = \{(0,0), (1,2), (2,3), (3,5)\}$

$S^2_1 = \{(5,4), (6,6), (7,7), (8,9)\}$

$S^3 = \{(0,0), (1,2), (2,3), (3,5), (5,1), (6,6)\}$

(3,5) discarded by dominance rules.

(P_j, w_j) (P_k, w_k)

$P_j \leq P_k$ and $w_j \geq w_k$

then (P_j, w_j) discarded.

$(6,6) \in S^3$ $x_3 = 1$

$(6,6) \notin S^2$ $x_2 = 1$

~~$(6,6) \notin S^1$~~

$(6-5, 6-4) = (1, 2)$

$(1,2) \in S^2$

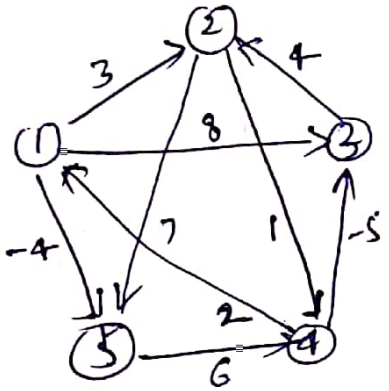
$(1,2) \in S^1$ $x_1 = 1$.

$(1,2) \notin S^0$.

All pair shortest Path.

The All pair shortest path problem is the determining of shortest distance or weight between every pair of vertices int given directed graph. the floyd-warshall algorithm run in $O(V^3)$ time to solve the problem.

this algorithm also work on negatitive weight. but no negative weight cycle are allowed.




```
Floyd-warshall(w)
1  n ← row(w)
2  D(0) ← w
3  for k ← 1 to n
4      do for i ← 1 to n
5          do for j ← 1 to n
6              do d_ij^k ← min(d_ij^(k-1), d_ik^(k-1) + d_kj^(k-1))
7  return D(n)
```

$$D^{(0)} = \begin{bmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & \infty & -5 & 0 & \infty \\ 5 & \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$$D^{(1)} = \begin{bmatrix} & 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 3 & 8 & \infty & -4 \\ 2 & \infty & 0 & \infty & 1 & 7 \\ 3 & \infty & 4 & 0 & \infty & \infty \\ 4 & 2 & 5 & -5 & 0 & -2 \\ 5 & \infty & \infty & \infty & 6 & 0 \end{bmatrix}$$

$d_{ij}^{k} \leftarrow \min(d_{ij}^{k-1}, d_{ik}^{k-1} + d_{kj}^{k-1})$

$d_{4,2}^{(1)} \leftarrow \min(d_{4,2}^{k-1}, d_{4,1}^{k-1} + d_{1,2}^{k-1})$

$\leftarrow \min(\infty, 2+3)$

$\leftarrow 5$

Resource Allocation Problem.

A Resource allocation problem in which m resource are to be allocated to n project... if j resource allocated to i project the profit $P(i, j)$ earned.

The problem is to allocate the resource to n project such way as to maximize total profit

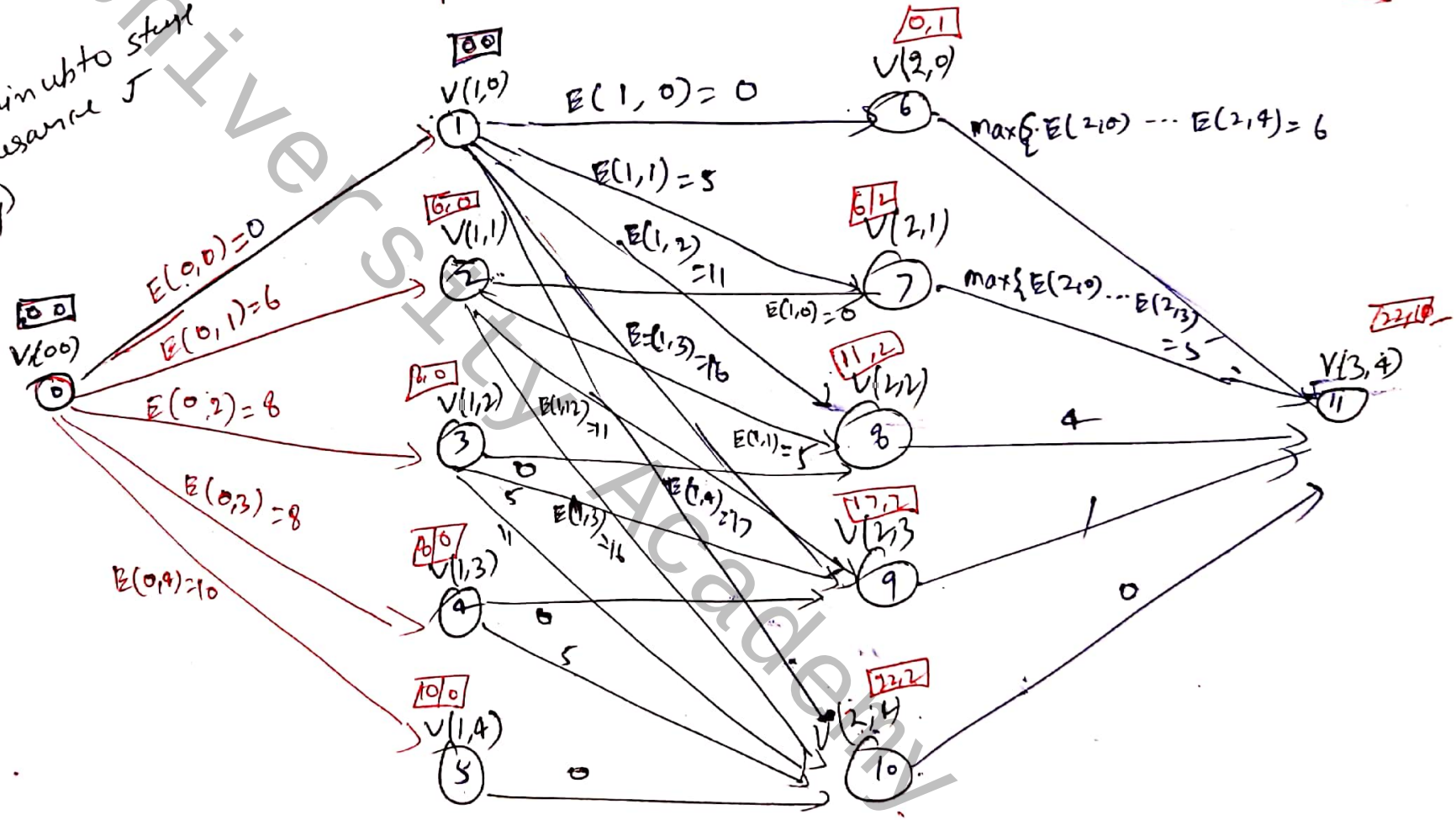
4 Villages to invest
3 Possible projects

(0, 1, 2, 3, 4)

Project \ Investment	0	1	2	3	4
0	0	6	8	8	10
1	0	5	11	16	17
2	0	1	4	5	06

V(i, j)
benefit to obtain upto stage i by using resource j
(But not including)

E(m, n)
value obtained by spending n resource on project m



Matrix chain Multiplication

we are given a sequence of matrices, find the most efficient way to multiply these matrices together.

eg: $\langle A_1, A_2, A_3 \rangle = ((A_1 A_2) A_3) = (A_1 (A_2 A_3))$

$\langle A_1 A_2 A_3 A_4 \rangle = A_1 \cdot A_2 \cdot A_3 \cdot A_4$

$$\frac{2(n-1)C_{n-1}}{n} = \frac{1}{4} * {}^{6}C_3 = \frac{1}{3} * \frac{\lfloor 6}{\lfloor 3 \cdot \lfloor 6-3} = \frac{1}{4} \times \frac{120}{6} = 5$$

$$\frac{2(n-1)C_{n-1}}{n} = \frac{2(2)C_2}{3} = \frac{4C_2}{3} = \frac{1}{3} \cdot \frac{\lfloor 4}{\lfloor 2 \lfloor 4-2} = \frac{12}{3 \times 2} = 2$$

1) $(A_1 (A_2 (A_3 A_4)))$

2) $(A_1 ((A_2 A_3) A_4))$

3) $((A_1 (A_2 A_3)) A_4)$

4. $((A_1 A_2)(A_3 A_4))$

5. $(((A_1 A_2) A_3) A_4)$

Matrix Multiplication.

$A_{p \times q}$ $B_{q \times r}$ total multiplication in $C_{pr} = p \times q \times r$

$$A = \begin{bmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{bmatrix} \quad B = \begin{bmatrix} B_{11} & B_{12} \\ B_{21} & B_{21} \end{bmatrix}$$

$2 \times \boxed{2 \quad 2} \times 2$

$$C = A * B = \begin{bmatrix} A_{11} \times B_{11} + A_{12} \times B_{21} & A_{11} \times B_{12} + A_{12} \times B_{21} \\ A_{21} \times B_{11} + A_{22} \times B_{21} & A_{21} \times B_{12} + A_{22} \times B_{21} \end{bmatrix}$$

total multiplication = 8 $= 2 \times 2 \times 2 = 8$

$A = 2 \times 2$ $C = 2 \times 2$

$B = 2 \times 2$

Example: Consider three matrix $\langle A_1 A_2 A_3 \rangle$ having dimention.

$\underset{A_1}{10 \times 100}$, $\underset{A_2}{100 \times 5}$, $\underset{A_3}{5 \times 50}$

1) $((A_1 A_2) A_3) = 10 \times 100 \times 5 = 5000$

$(A_1 A_2) = 10 \times 100 \times 5 = 5000$

$10 \times 5, 5 \times 50 = 2500$

7500.

2. $(A_1 (A_2 A_3)) =$

$A_2 A_3 = 100 \times 5 \times 50 = 25,000$

$10 \times 100 \quad 100 \times 50 = 10 \times 100 \times 50 = 50,000$

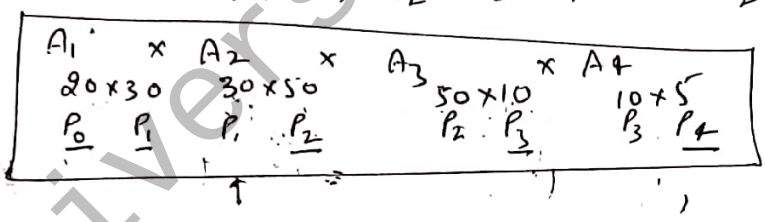
75000

Consider the following Four matrices. Find optimal parenthesis of matrix chain multiplication.

matrix	order
A_1	20×30
A_2	30×50
A_3	50×10
A_4	10×5

order of matrix chain: $(P) = \langle 20, 30, 50, 10, 5 \rangle$ so.

$P[0] = 20$, $P[1] = 30$ $P[2] = 50$ $P[3] = 10$ $P[4] = 5$

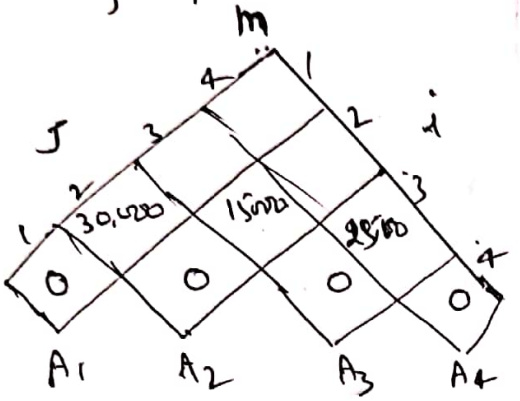


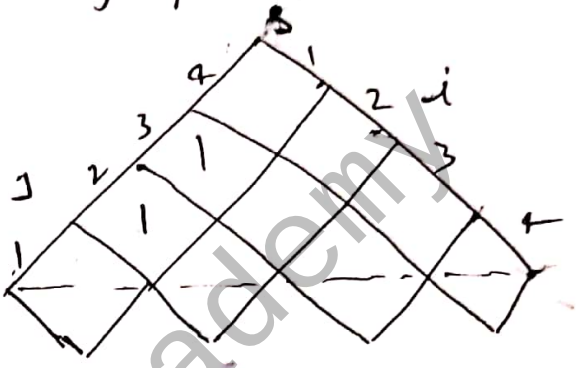
m

i \ j	1	2	3	4
1	0	30000	21,000	13,000
2		0	15,000	18,000
3			0	25000
4				0

s

i \ j	1	2	3	4
1	0	1	1	1
2		0	2	2
3			0	3
4				0





$P\langle 20, 30, 50, 10, 5\rangle$

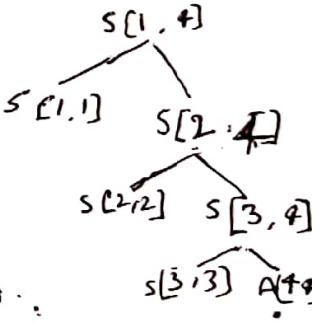
(A1 (A2 A3

$$m[i,j] = \min_{i \le k < j} \{ m[i,k] + m[k+1,j] + P_{i-1} P_k P_j \}$$

$$m[1,2] = \min_{1 \le k < 2} \{ m[1,1] + m[2,2] + P_0 \cdot P_1 \cdot P_2 \}$$

$$= 0 + 0 + 30,000$$

$s[1,2] = 1$ because $k = 1$



similarly. $m[2,3] = 15,000$, $m[3,4] = 2500$.

$$m[1,3] = \min_{1 \le k < 3} \begin{cases} m[1,1] + m[2,3] + P_0 P_1 P_3 \\ m[1,2] + m[3,3] + P_0 P_2 P_3 \end{cases} = \min \begin{cases} 0 + 15000 + 20 \times 30 \times 50 \\ 30,000 + 0 + 20 \times 50 \times 10 \end{cases}$$

$$= \begin{cases} 21,000 \\ 40,000 \end{cases}$$

$$m[2,4] = \min_{2 \le k < 4} \begin{cases} m[2,2] + m[3,4] + P_1 \cdot P_2 \cdot P_4 \\ m[2,3] + m[4,4] + P_1 P_3 \cdot P_4 \end{cases} = \min \begin{cases} 0 + 2500 + 7500 \\ 15,000 + 1500 \end{cases} = 10,000$$

$$m[1,4] = \min_{1 \le k < 4} \begin{cases} m[1,1] + m[2,4] + P_0 \cdot P_1 \cdot P_4 \\ m[1,2] + m[3,4] + P_0 P_2 P_4 \\ m[1,3] + m[4,4] + P_0 P_3 P_4 \end{cases} = \min \begin{cases} 0 + 10000 + 3000 \\ 30,000 + 2500 + 5000 \\ 21000 + 0 + 10,00 \end{cases} = \begin{cases} 13,000 \\ 37,500 \\ 31,000 \end{cases}$$

Longest Common Subsequences (LCS)

We are given two sequence $X = \langle x_1, x_2, x_3 \cdots x_m \rangle$ and $Y = \{ y_1, y_2, y_3 \cdots y_n \}$ and wish to find a maximum length common subsequence of X and Y.

1. Subsequence:

X = IRONMAN

ROMA
ROMAN
IRON
} subsequence in X.

X → IRONMAN

MANI
NRMA
} Not a subsequence

2. Common Subsequence:

Stm 1 a b d a c e

Stm 2 b a b c e

ba ce
ab ce
} longest common subsequence.

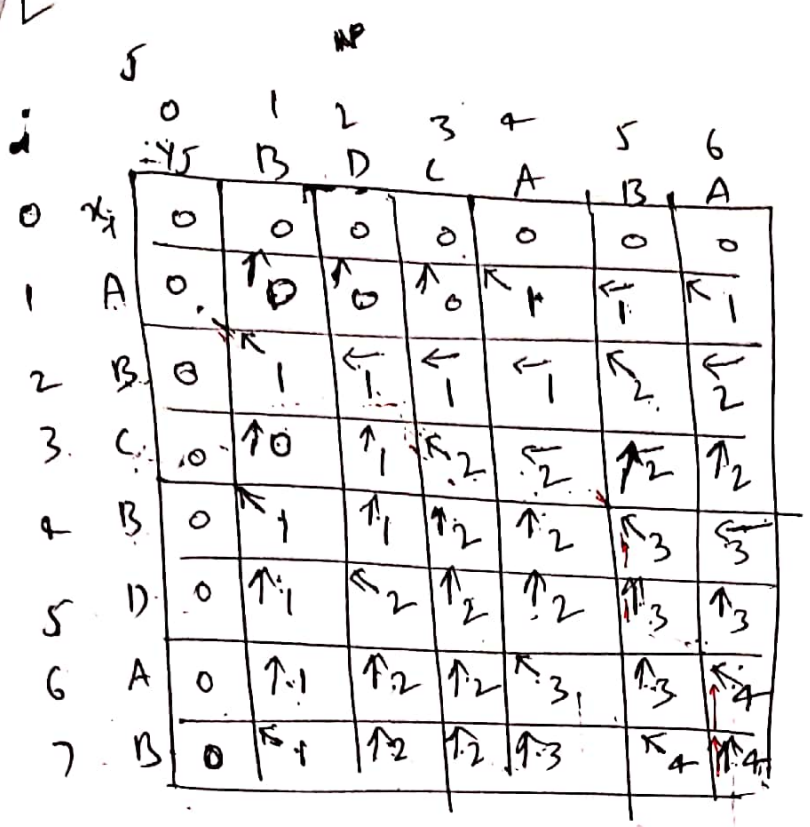
ce
bce
ace
} also common subsequence.

3 longest common subsequence.

$X = \langle A, B, C, B, D, A, B \rangle$

$Y = \langle B, D, C, A, B, A \rangle$

LCS = (B, C, B, A)



B C B A

```
for i ← 1 to m
for j = 1 to n.
{
if (X[i] == Y[j])
    LCS[i,j] = LCS[i-1, j-1] + 1
else B[i][j] ←
    LCS[i,j] = max(LCS[i-1,j], LCS[i,j-1])
}
```

X = ⟨A B C B D A B⟩

Y = ⟨B D C A B A⟩

B C A B

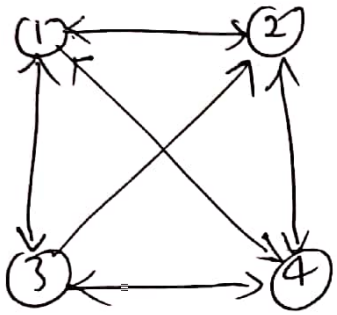
mxn time

```
for(i=1 ; i<=m  i++)
    for( j=1 : j<=n j++)
        if (X[i] == Y[j])
            set L[i,j] = L[(i-1),(j-1)] + 1
            B[i,j] = '↖'
        else
            if L[i-1][j] >= L[i, j-1])
                L[i, j] = L[i-1, j]
                B[i,j] = ↑
            else L[i, j] = L[i, j-1]
                B[i,j] = '←'
}
Return L and B.
```

Travelling Salesperson Problem.

TSP is to obtain the shortest tour from a given set of cities while visiting each city exactly once and returning to city from where we started. with minimal distance. weight.

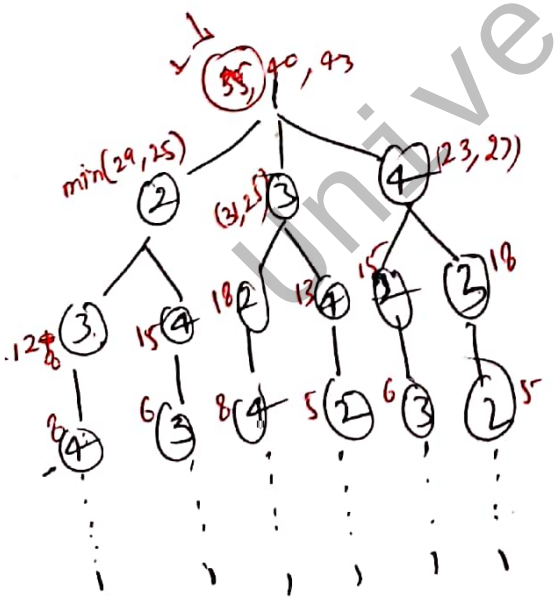
TSP is represented as a weighted graph G(V, E) where V is set of vertices (cities) and E refers edges (path ~~from~~ betⁿ the cities).



	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

$$g(i,S) = \min_{j \in S} \{ c_{ij} + g(j, S-\{j\}) \}$$

g(i,S) is shortest path from starting at vertex ϕ i and going all vertex in S and terminated 1.



$g(2,\phi) = c_{21} = 5$, $g(3,\phi) = c_{31} = 6$, $g(4,\phi) = c_{41} = 8$

$g(2,\{3\}) = c_{23} + g(3,\phi) = 9 + 6 = 15$

$g(2,\{4\}) = c_{24} + g(4,\phi) = 18$

$g(3,\{4\}) = 20$, $g(3,\{2\}) = 18$

$g(4,\{3\}) = 15$, $g(4,\{2\}) = 13$

Branch & Bound.

Branch - Branching using State-space tree maximisation/minimization

Bounding : done by using concept of upper or lower bound.

It is similar to backtracking but ~~B&B find solution~~ it differs from backtracking in the sense that all children of E-node are generated before any other live node can become an E-node. The selection of E-node depends on satisfying the condition of bounding function.

1. First-in - First out → Implemented using Queue (BF-S)
2. Last-in - First out → Implemented by Stack (DFS) D-search
3. Least count(LC) → Implemented by priority queue.

DFS and BFS ——— Blind search

LC ——→ Run on priority.

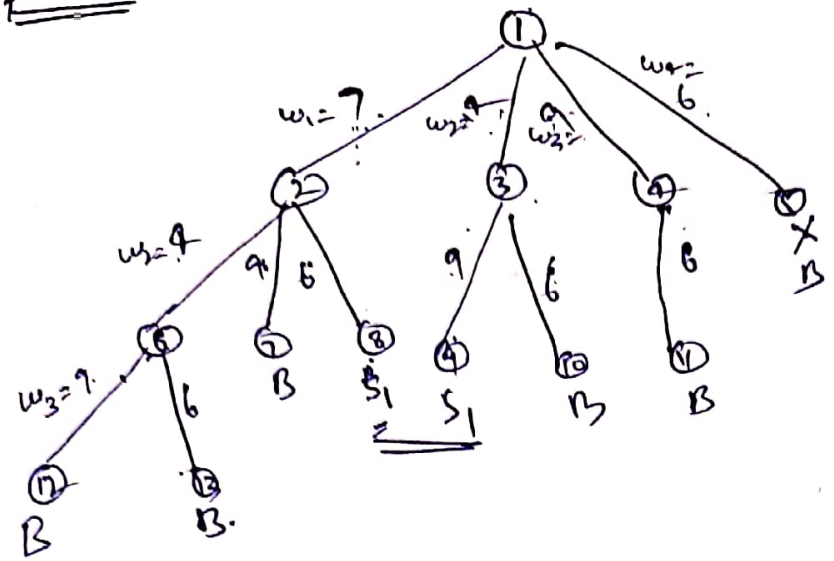
Terms used in Back Tracking

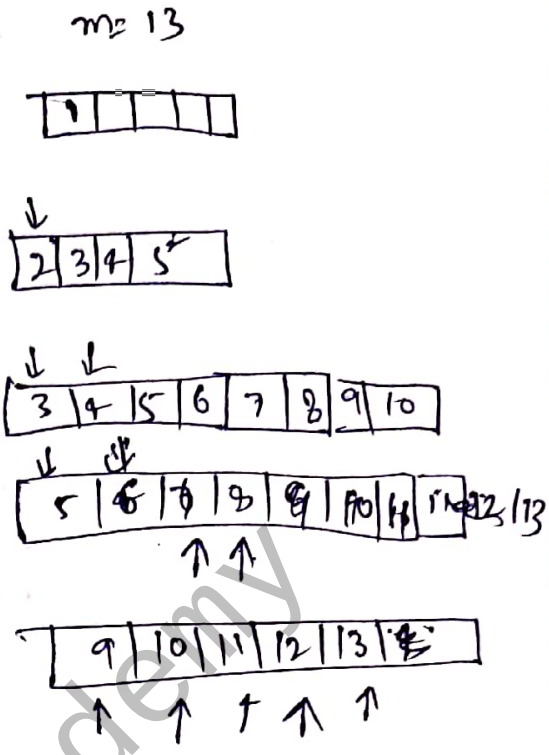
1. Live Node — the node that has been generated but none of it descendants are yet generated
2. Extended node (Enode) : the live node currently being explored.
3. dead Node - Not been expanded further
5. Answer node - node that represent answer of problem.

n=4 ; $w_1 w_2 w_3 w_4 = \{4, 8, 9, 6\}$

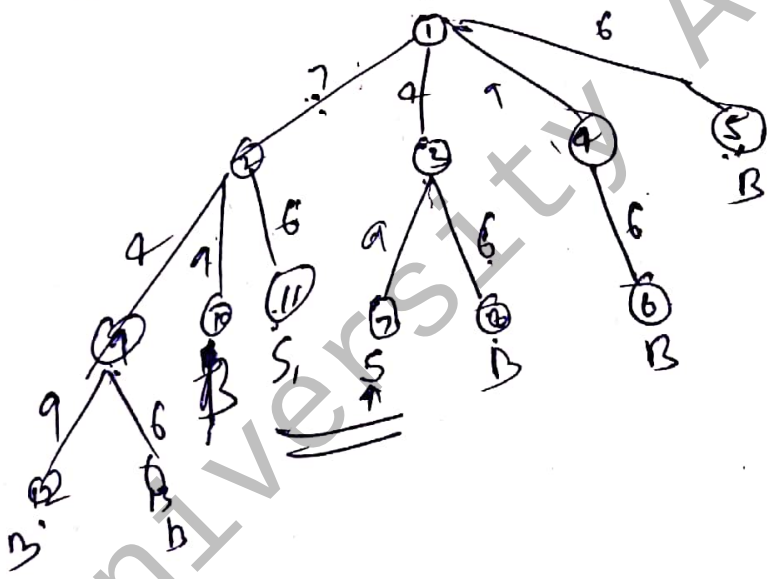
m = 13

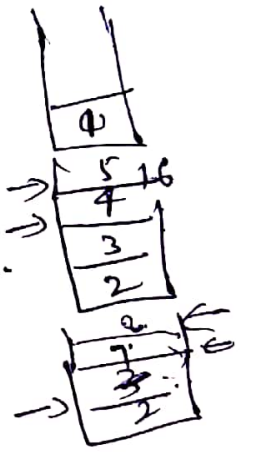
FIFO



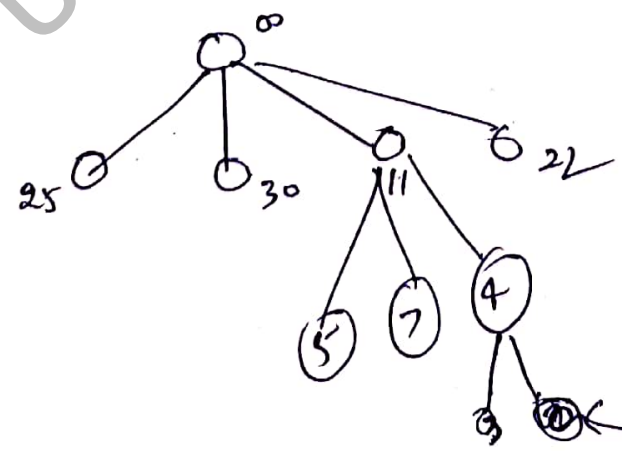


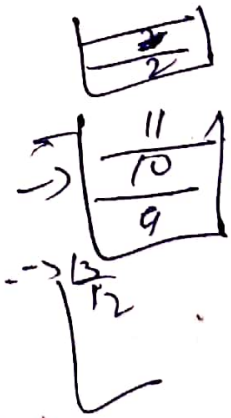
LIFO (stack)





LC





0/1 Knapsack Problem using LCBB

Least cost Branch Bound aim to minimization of ~~problems~~.

So we take the objective function

$$C(x) = -\sum P_i x_i \text{ subjected to } \sum_{i=1}^{n} w_i x_i \le m.$$

In order to convert problem of maximization problem. we ~~one~~ define two Bounding function UB(x) and LB(x). as Bounding function.

Upper Bounding.

$$LB(x) \le c(x) \le UB(x)$$

Example

$(P_1 P_2 P_3 P_4) = (10, 10, 12, 18)$

$(w_1 w_2 w_3 w_4) = (2, 4, 6, 9)$

$m = 15$

$$UB = \sum_{i=1}^{n} P_i x_i$$

$$LB = \sum_{i=1}^{n} P_i x_i \text{ (in fraction)}$$

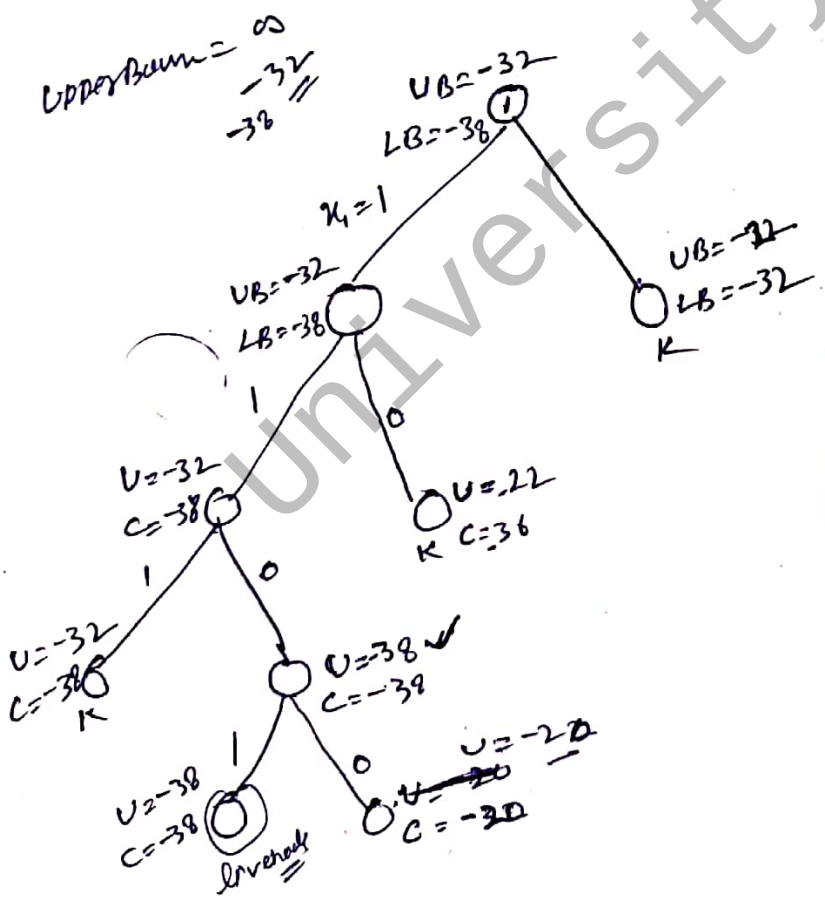
Fixed size solution

e.g. $S = \{1\ 0\ 1\ 0\}$.

$$U.(UB) = 10 + 10 + 12 = 32$$

$$C(LB) = 10 + 10 + 12 + \frac{10}{3} = 38$$

$$2 + 4 + 6 + 3$$



$$U = 10 + 12 + = 32$$

$$C = 10 + 12 + \frac{2}{3} \times \frac{5}{9} = 32$$

$$4 + 6 + \frac{9 \times 5}{9}$$

$$U = 10 + 12 = 22$$

$$C = 10 + 12 + 18 \times \frac{3}{9} = 36$$

$$2 + 6 + 9 \times \frac{3}{9}$$

Backtracking

Brute force Approach.
(Depth First Search)

Backtracking uses Brut force Approce to solve problem.

Brute for Approach say that for any given problem you should try out all possible solution and pick up desired solution.

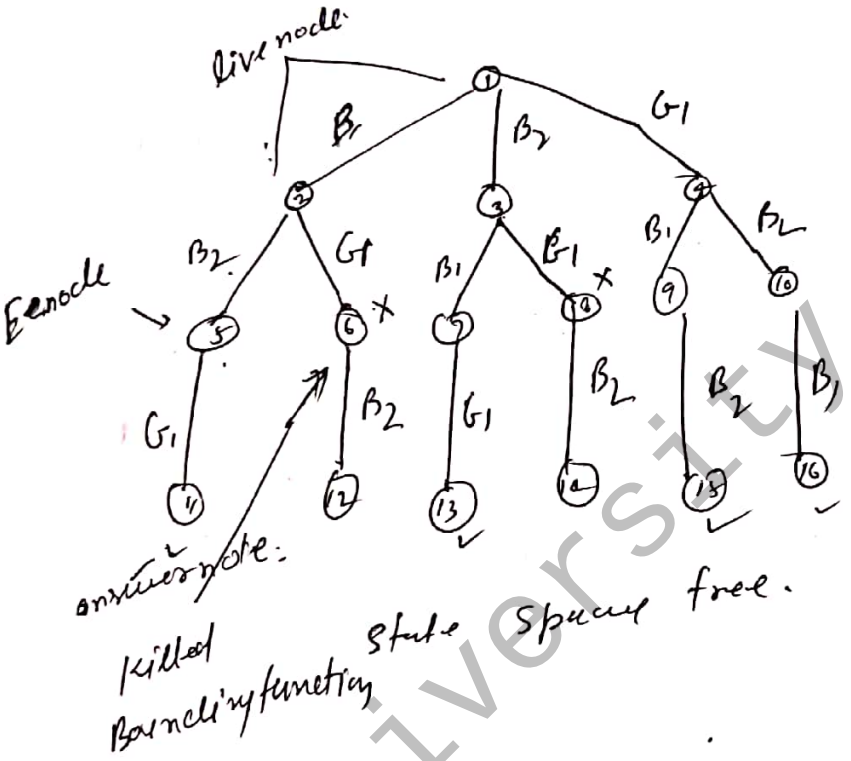
Back tracking used to explore all possible solution

$(x_1, x_2, x_3) = \{B_1, B_2, G_1\}$

char

Explicit

$3! = 6$ way.



Constraint → girl should not sit in the middle

Implicit

4 Solutions

Terms used in Backtracking

1. Backtracking require two set of constraints called explicit and implicit.
2. solution space :→
3. state space tree.
4. Bounding function
5. live node
6. E- node
7. Dead node.

N-Queens Problem

N-queens are to be placed on n×n Chessboard so that no two queens are in attacking to each other
ie. no two queens are in

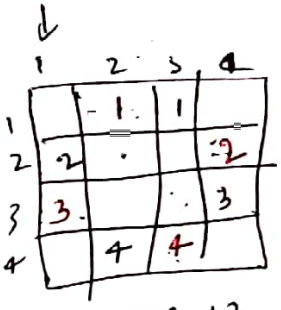
1- same row
2. same column.
3. diagonal.

the solution consist n! leaf node and total

$$1 + \sum_{j=0}^{n-1} \left[\prod_{i=0}^{j} (n-i) \right] \text{ node.}$$

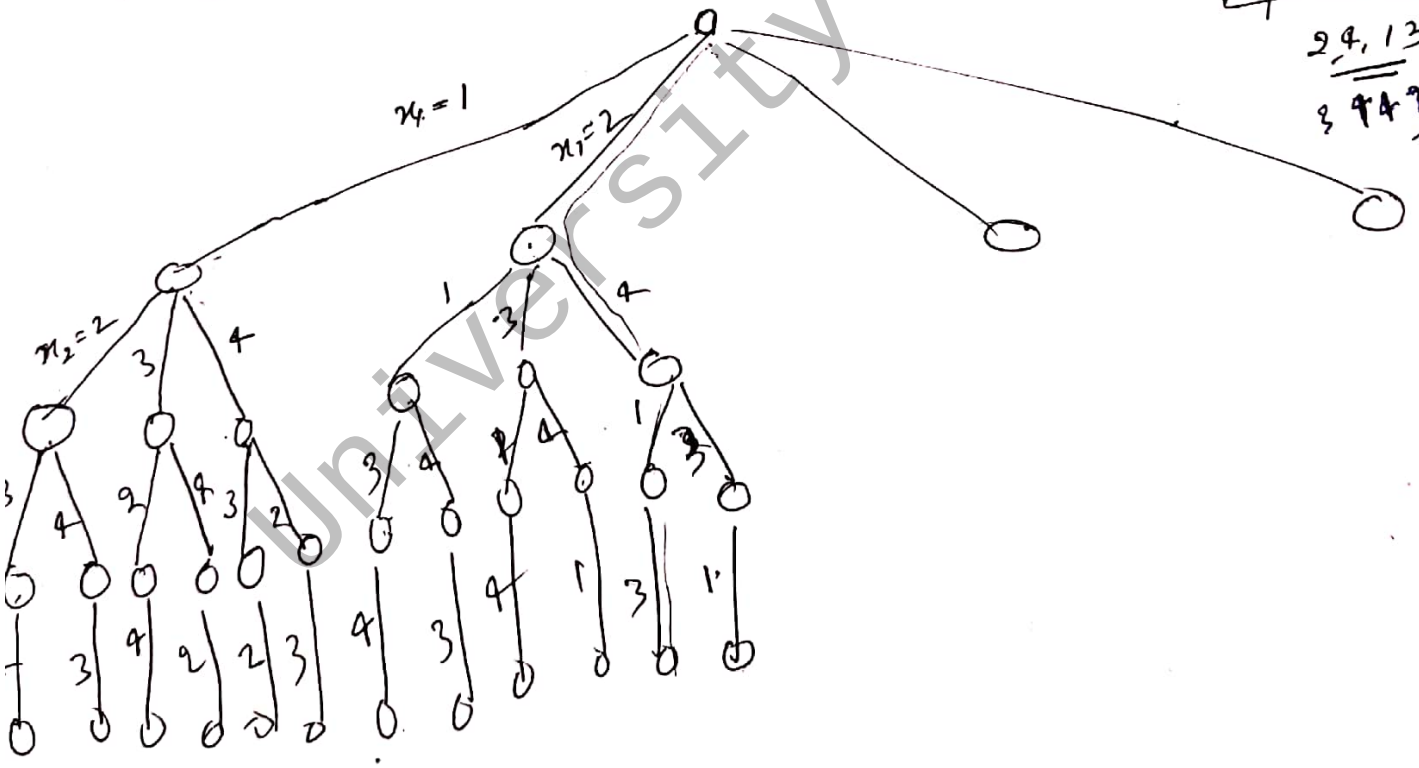
there two varient of n-queens problem.

1- 4-Queens problem
2- 8-Queens problem



2,4,1,3

3,1,4,2



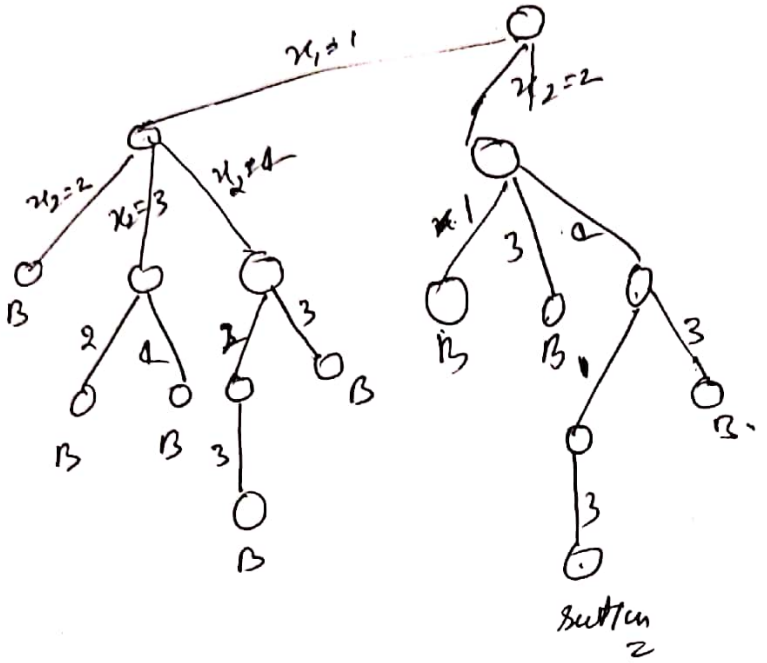
L4 = 24 Leaf node.

$$1 + \sum_{j=0}^{n-1} \left[\prod_{i=0}^{j} (n-i) \right] = 1+4+4\times3+4\times3\times2+4\times3\times2\times1$$

$$= 1+4+12+24+24$$

$$= 65$$

Solution generated during Backtracking.



Sum of Subsets

Suppose we are given distinct positive integers w_i, $1 \le i \le n$ and m. This we desire to find all combinations of these numbers whose sum are m. This is called sum of subset problem.

Problem calls for finding all subsets of the w_i whose sums are m.

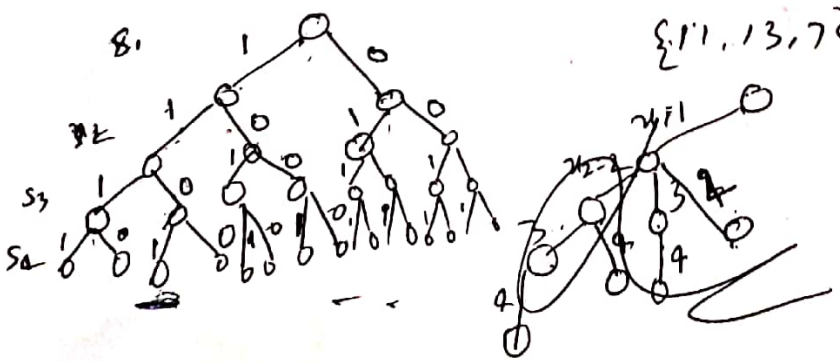
- the explicit constraints: $x_i \in \{ j \mid j \text{ is an integer and } 1 \le j \le n \}$
- Implicit constraints:: ~~no two be the same~~ avoid generating multiple instances of same subset eg. (1,2,4) and (1,4,2). Represent same subset.

~~solution space tree~~:

(1)

Example $n = 4 \{ s_1 s_2 s_3 s_4 \} = \{ 11, 13, 24, 7 \} = m = 31$

$\{11, 13, 7\}$, $\{24, 7\}$.



(1 1 0 1)
(0 0 1 1) Selected

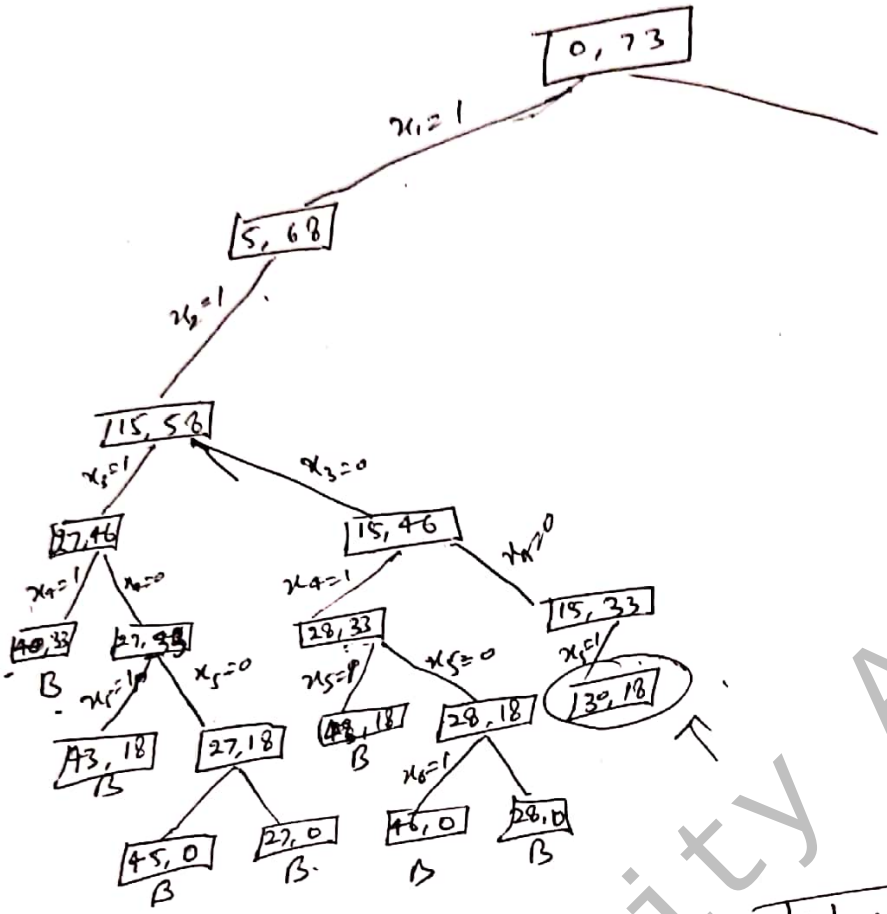
2^n ~~Total~~ Solution Path

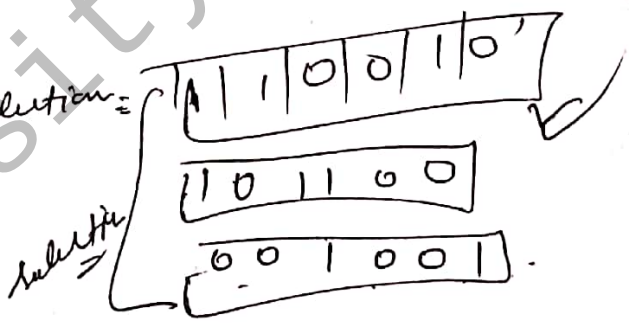
$n = 6, \ m = 30$

$w[1:6] = \{5, 10, 12, 13, 15, 18\}$

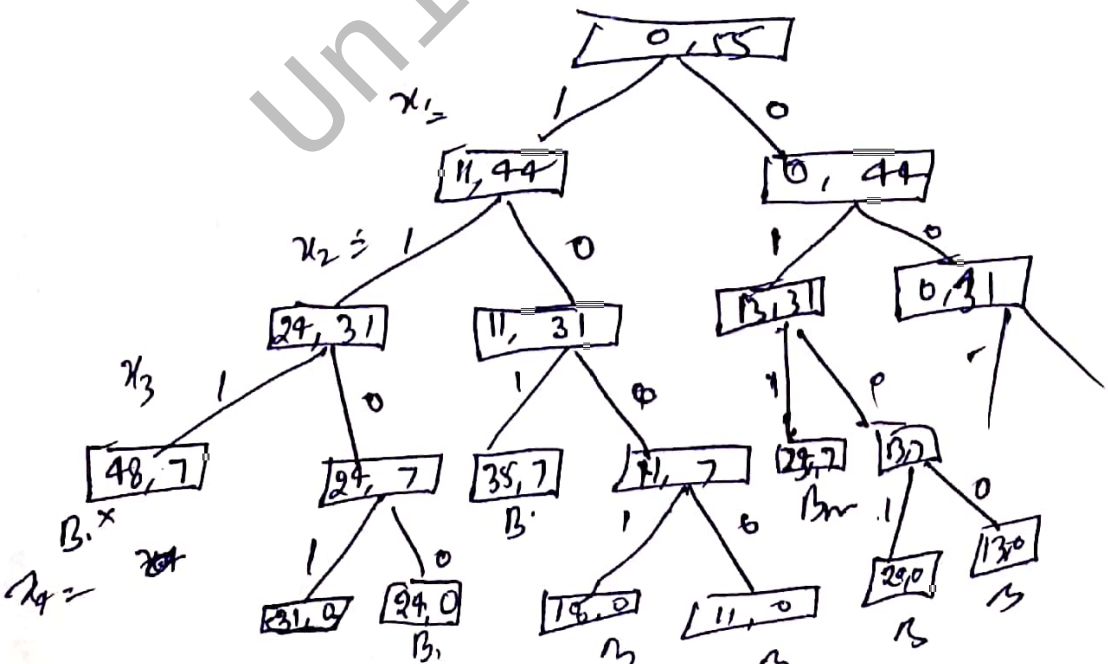
$\sum w_i x_i + w_{k+1} \leq m$

$\sum w_i x_i + \sum w_i > m$





$2^n = 2^6 = 64$

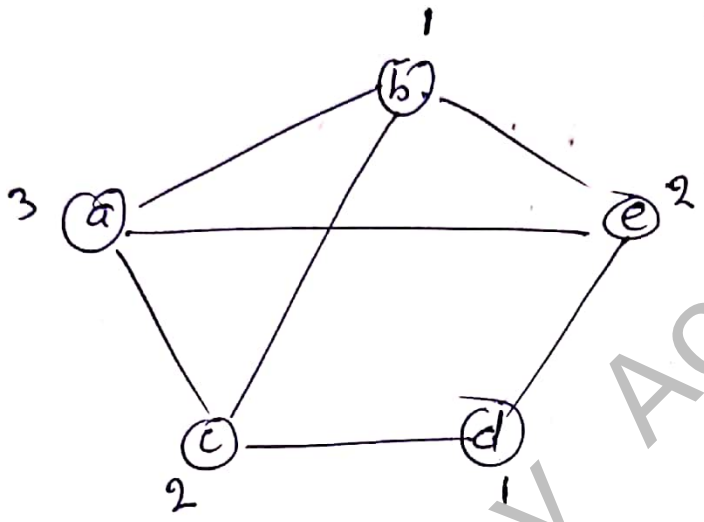


Graph Coloring

Graph coloring is way of labelling graph component such as vertices, edges and regions under some constraints. In graph no two adjacent vertices, edges or regions are colored with same color. the minimum number of color requires to color graph is called cromatic number.

m coloring decision problem
m color optimization problem

Vertex coloring

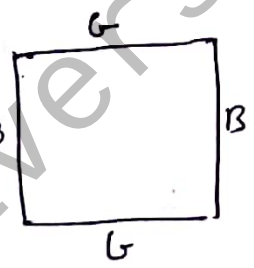


Color = {R, G, B}

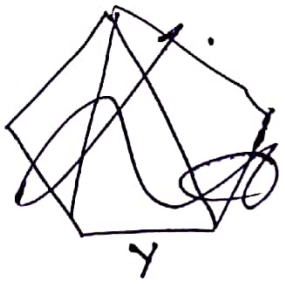
Cromatic number is 3

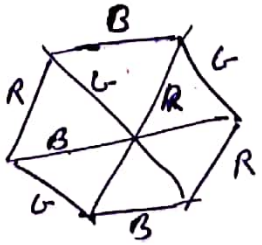
a b c d e
3 1 2 1 2

Edge coloring

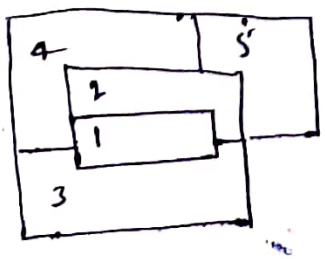


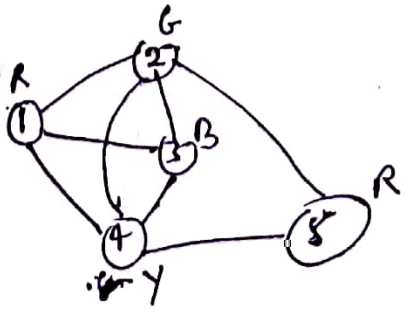
Cromatic Num-2



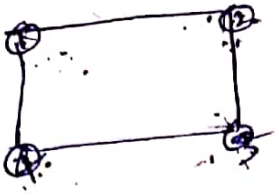


Regions or map coloring

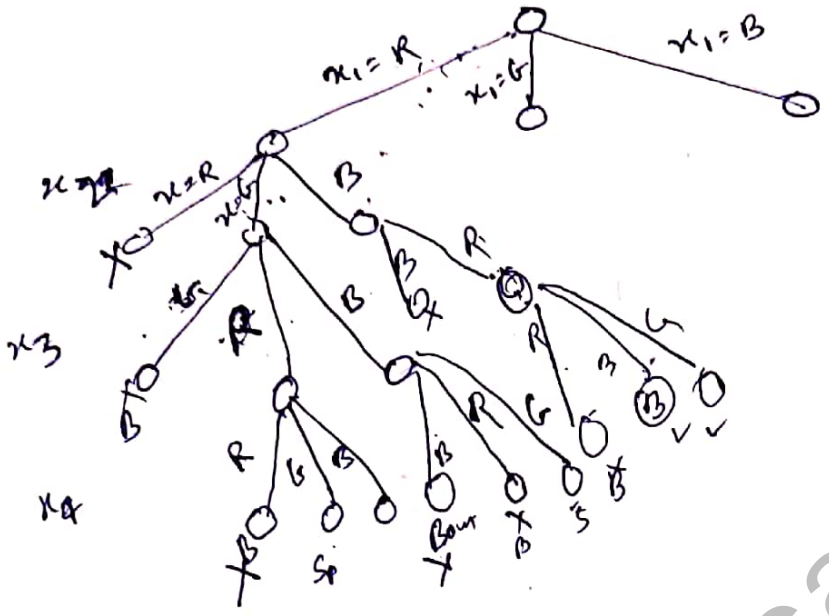




- Applications
  1. Making schedule of TT.
  2. Radio frequency Assign
  3. Sudoku
  4. map coloring.



Color = {R, G, B}

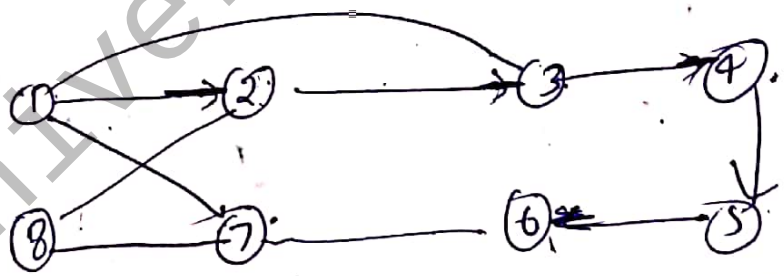


RGRG
RGRB
RGBG
RBRB
RBRG

Chromatic Number = 2

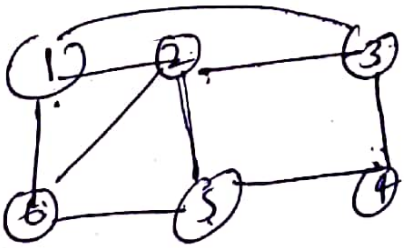
Hamiltonian Cycles

Let G=(V, E) be a connected graph with n vertices A Hamiltonian Cycle is sound-trip path along n edges of G. that visit every vertex once and Return to its starting point.

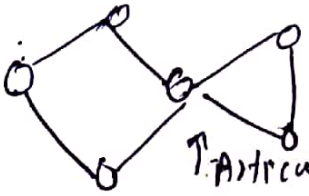


$\frac{(n-1)!}{2}$

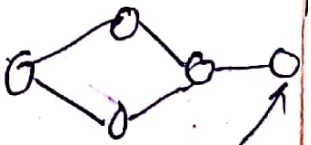
→ 1 3 4 5 6 7 8 2 1
1 2 8 7 6 5 4 3 1
only distinct circuit is there



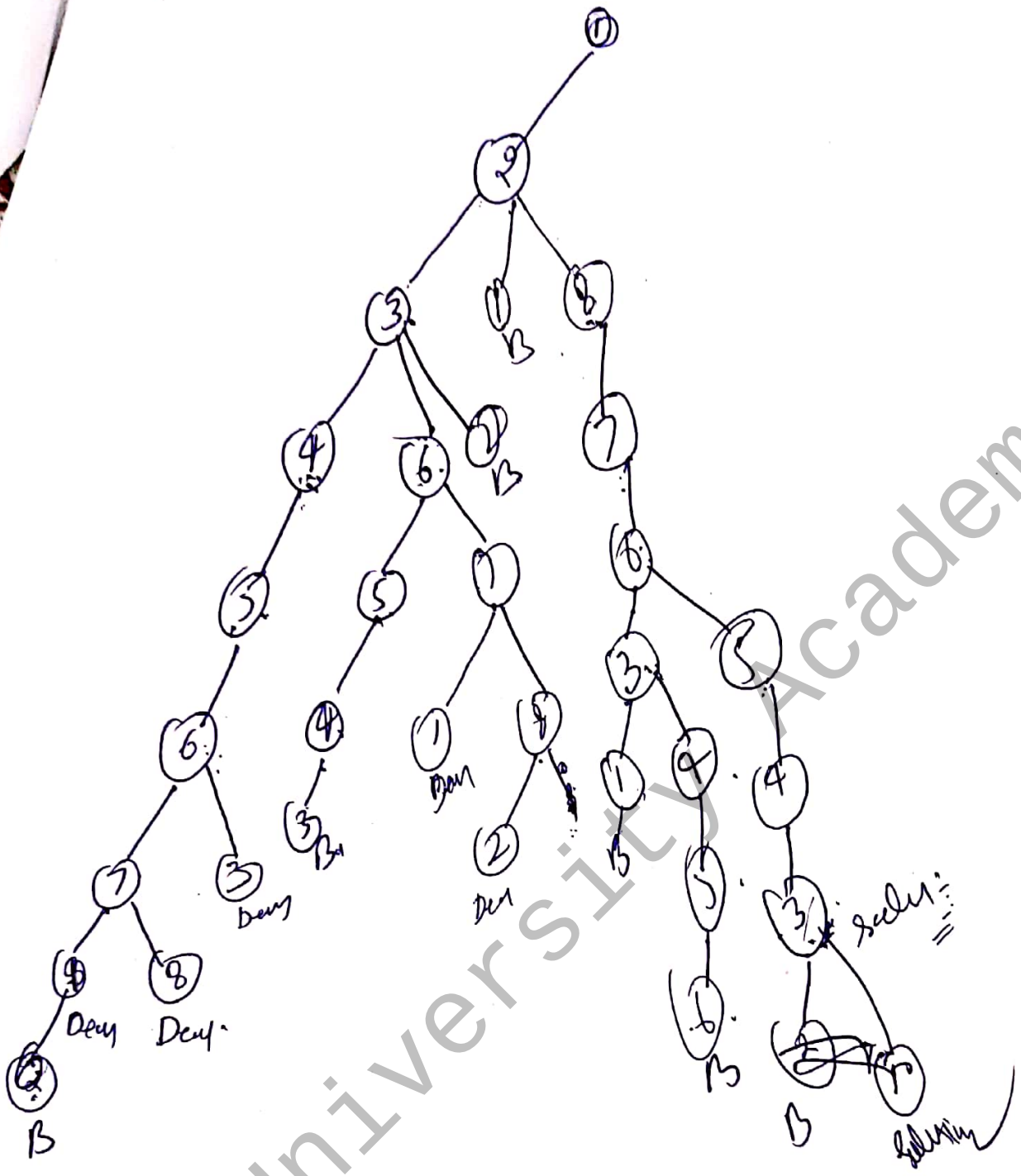
1 2 3 4 5 6 1
1 3 4 5 6 2 1
1 6 2 5 4 3 1
2 3 4 5 6 1 2 ✗
1 2 6 5 4 3 2 1



↑ Articulation point



Pendant Vertex



STRING MATCHING

String Matching Algorithm try to find a place where one or several string (called pattern) are found within a larger string.

we assume that the Text array $T[1.....n]$ and pattern $P[1.....m]$ $m \leq n$ elements of P and T $\in \Sigma$, $\{0,1\}$ / $\{a,b,c.....z\}$.

Ex: T | a | b | c | a | b | a | a | b | c | a | b | a | c |
1 2 3 .

P $\xrightarrow{S=3}$ | a | b | a | a |

		matching time	Processing time
1- Naive	→	$O((n-m+1)m)$	0
2- Rabin karp	→	$O((n-m+1)m)$	$O(m)$
3- Finite Automata		$O(n)$	$O(m)$
4- Knuth-Morri's-Patt (KMP)		$O(n)$	$O(m$

1- Naive String Matching Algorithm

The naive algorithm finds all valid shift using a loop that checks the condition $P[1...m] = T[S+1..... S+m]$ for each of the $n-m+1$ possible value of s.

T | a | c | a | a | b | c | $n-m=3$

P | a | a | b |

① | a | a | a | a | b | c |
S→0 | a | a | b |

| a | a | a | b | c |
S→1 | a | a | b |

| a | c | a | a | b | c |
S→2 | a | a | b |
multiple S→2 | a | a | b |

| a | c | a | a | b | c |
S→3 | a | a | b |

```
NAIVE-SRING-MATCHERS(T, P)
1- n ← length[T]
2. m ← length[P]
3- for s ← 0 to n-m
4-     do if P[1 ....m] = T[s+1.....s+m]
5-         Then print "Pattern occurs with shift" s
```

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

The Rabin-Karp algorithm:

developed by Rabin karb. This algorithm matth the Pattern in the text using Rashing

T = a c a a b a a

P = a a b

a = 1
b = 2
c = 3
d = 4
e = 5
⋮
= 9

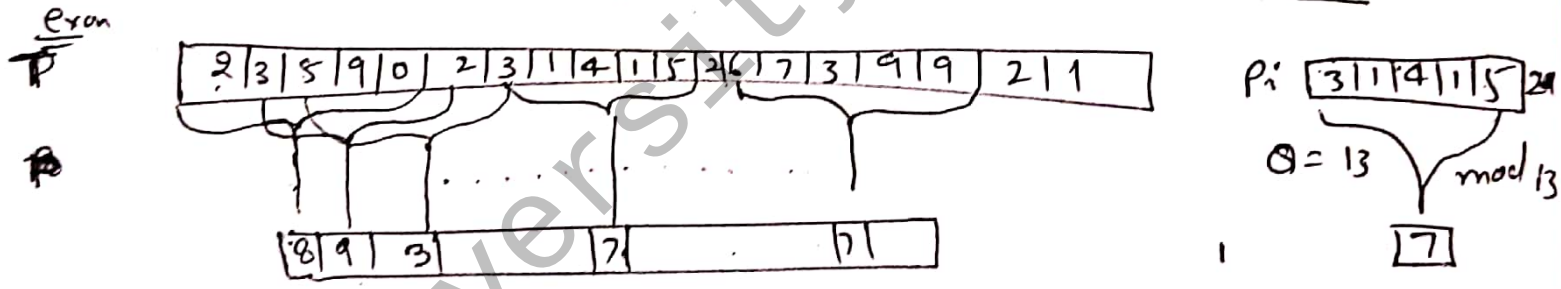
T = 1 3 1 1 2 1 1

P = 1 1 2

1+1+2=4 hashing = sum of digit

valid hit . spurious hit.

Rabin karp suyest a hash function → chase a prime numbers q and cas calculate. P[1....m] mod q.



valid hit

spurious hit:-

T = 31415926535897 9793
9384626433832795

P = 26

Q = 11

$O((n-m+1)m)$

String Matching with finite Automata:

These string matching Automata are very efficient they examine each text character exactly once. hence ~~this~~ time complexity $\theta(n)$.

A Finite state automata is 5-tuple $(Q, q_0, F, \Sigma, \delta)$ where

Q - set of states

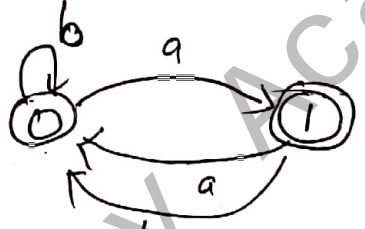
q_0 is start state

$F \subseteq Q$ set of final state

Σ is input alphabet

δ transition mapping

state	input a	input b
0	1	0
1	0	0



abaaq Accepted

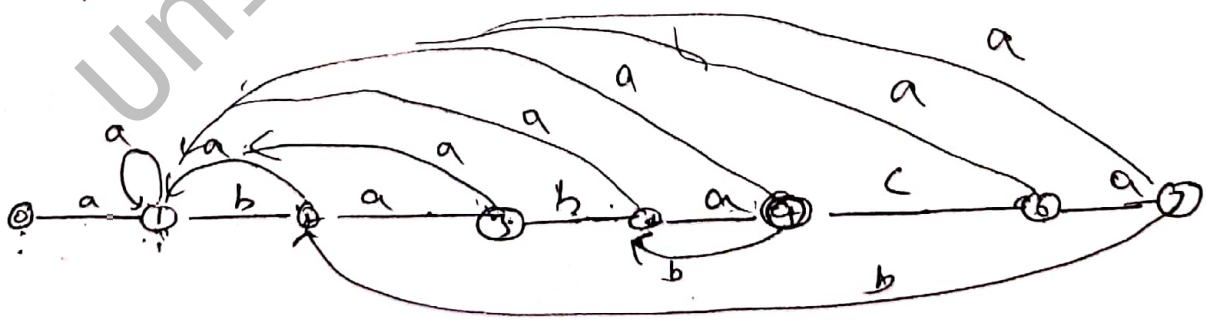
abbaa Rejected.

String Matching

	1	2	3	4	5	6	7
P =	a	b	a	b	a	c	a

$\Sigma = \{a, b, c\}$

$m = 7$



$k \leftarrow \min(7+1, 0+2) = 2$

$k = 2-1 = 1$

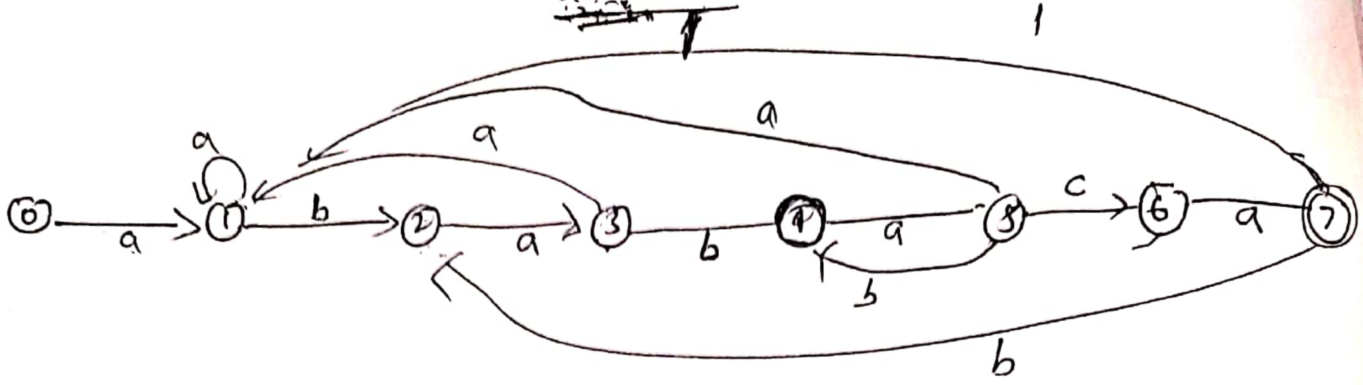
$P_1 \sqsupset P_0 a$

$\delta(q, a) = k = 1$

$k \leftarrow \min(7+1, 0+2$

$P_0 \sqsupset P_0 b$

a b a b a c a



a b a b a b = P_4

longest prefix of P that is suffix of

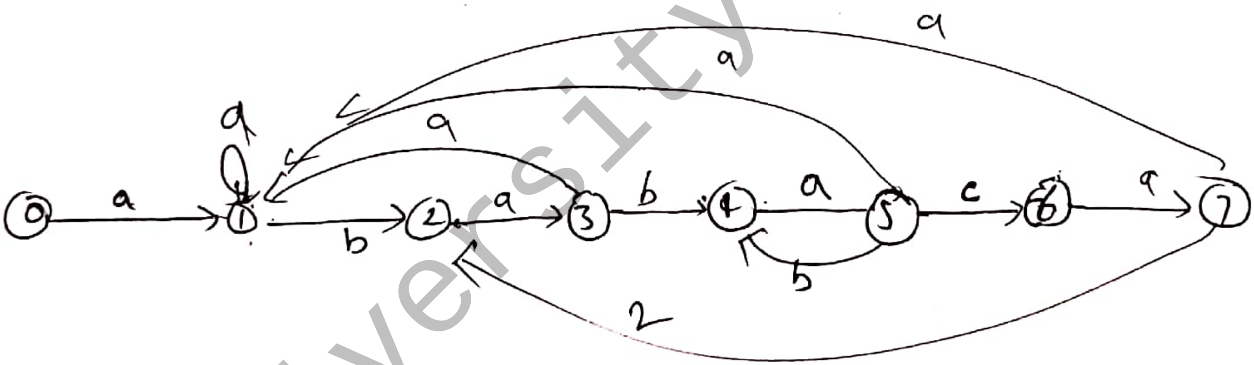
ababacb

longest prefix that of P that suffix

a b a b a c a a

a b a b a c a b 2

a b a b a c a



aa = 1

a c

a b b x

a b c x

a b a a

a b a b

a b a b b x

a b a b

a b a b x

a b a b a a

a b a b a b

a b a b a b

a b a b a c b x

c x

a b a b a c a b

state	input a	input b	input c
0	1	0	0
1	1	2	0
2	3	0	0
3	1	4	0
4	5	0	0
5	1	4	6
6	7	0	0
7	1	2	0

j	1	2	3	4	5	6	7	8	9	10	11
T[i]	a	b	a	b	a	b	a	c	a	b	a
0	1	2	3	4	5	4	5	6	7	2	3

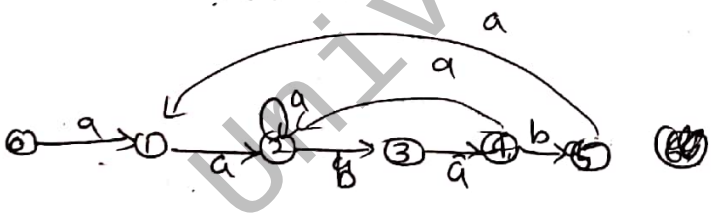
9-7=2

Finite Automata - Matcher(T, δ, m)

1- n ← length[T]
2- q ← 0
3- for i ← 1 to n
4- do q ← δ(q, T[i])
5- if q = m
then print pattern
occurs with
shift i-m

T = a a a b a b a a a b a a b a b a a b

P = a a b a b



6-5 = 1 shift

14-5 = 9

state	input a	input b
0	1	0
1	2	0
2	2	3
3	4	0
4	2	5
5	1	0

i	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17
T[i]	a	a	a	b	a	b	a	a	b	a	a	b	a	b	a	a	b
0	1	2	2	3	4	5	1	2	3	4	2	3	4	5			

a b

a a a

a a b b

a a b a a

a a b a b a

y b

Put [0...4] C

A C A C A C

0 - 4

A C A C A C A

0 - 4

S → C ④

S an A

Put [0...4] A

A C A C A A

0 - 4

suffix function

$\Sigma^* = \{\phi, a, b, c\} \times \{0, 1 \ldots m\}$

$\sigma(x)$ is length of longest prefix of P that is suffix of x

$\sigma(x) = \max\{k : k_k \sqsupset x\}$

state

$\delta(0, a) = \sigma(P_0 a) = a = 1$

$\delta(0, b) = \sigma(P_0 b)\ 0 = b \times$

$\delta(0, c) = \sigma(P_0 c)\ 0 = c \times$

$\delta(1, a) = \sigma(P_1 a) = aa = 1$

$\delta(1, b) = \sigma(P_1 b) = ab = 2$

$\delta(1, c) = \sigma(P_1 c) = a \times$

$\delta(2, b) = \sigma(P_2 b) = abu = 1$

$\delta(2, c)$ skipped

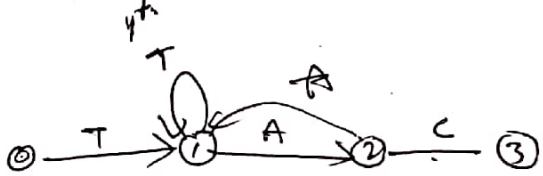
$\delta(2, A) = 1$

$\delta(2, c) = 3$

$\delta(2, G) = 0$

$\delta(2, T) =$

String TAC $\Sigma = \{A, C, G, T\}$



delta(A) =

$\delta(0, A) = \sigma(A) = 0 = \delta(0, c) = \sigma(G)$

delta(0 T) = $\sigma(T) = 1$ string T matches (no) symbol matching so far.

delta(1, A) = 2 string corresponding symbol matches.

$\sigma(1, C) = \sigma(1, G) = 0$ string not matched.

$\sigma(1, T) = 1$ is not correct prefix and Symbol but it is correct

Knuth-Morris-Pratt Algorithm.

Problem with naive algorithms

kmp is linear time string matching algorithm. in $\Theta(n)$ time

Text a b b a b a b a b b a

Pattern a b a b a

$\Theta(mn)$ Naive algo

kmp:

Prefix : start from beginning (LHS)
: a, ab, abc, abcd, ...

for Pattern
~~abcdd~~.
abcdabc

Suffix start from ending (RHS)
c, bc, abc, dabc ....

P_i: (π) table longest prefix same as suffix · LPS

P_1:

	1	2	3	4	5	6	7
	a	b	c	d	a	b	e
	0	0	0	0	1	2	0

	8	9
	a	b
	1	2

P_2:

	1	2	3	4	5	6	7	8	9
	a	b	c	d	e	a	b	f	a
	0	0	0	0	0	1	2	0	1

	10	11
	b	c
	2	3

P_3:

	1	2	3	4	5	6	7	8	9
	b	a	b	a	a	b	b	a	b
	0	0	1	2	1	1	2	3	2

	10
	a
	1

P_4:

	1	2	3	4	5	6	7	8	9
	a	a	a	a	b	a	a	c	d
	0	1	2	3	0	1	2	0	0

=

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

i		1	2	3	4	5	6	7	8	9	10	11	12	13	14
String		a	b	a	b	c	a	b	c	a	b	a	b	a	b	d

J = 0	1	2	3	4	5
Pattern	a	b	a	b	d
	0	0	1	2	0

i is not back tracking

1. take two variables i and J
2. compare i with J+1
3. if match move both i and J to Right
3. if match, if not then move J to the location in Π table
4. if J = 0 move i to the Right

Approximation Algorithm

An approximation algorithm is way of dealing with NP-Completeness for optimization problem. The goal of Approximation algo. is come as close as possible to optimal value in polynomial time.

Definition

C → cost of solution
C^* → cost of optimal solution
$\rho(n)$ → approximation Ratio.
n → input size

maximization $\frac{C^*}{C} \leq \rho(n)$

minimization $\frac{C}{C^*} \leq \rho(n)$ $\quad \therefore \rho(n) \not< 1$

1- Vertex Cover problem. → A vertex cover of a graph is a subset of vertices which cover every edges.

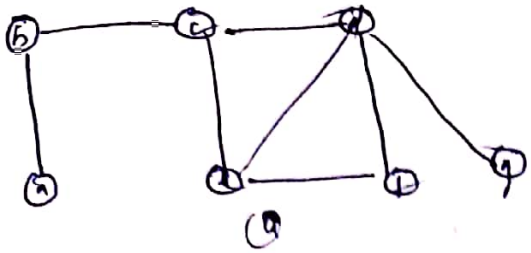
2- TSP problems

3- Set covering problems

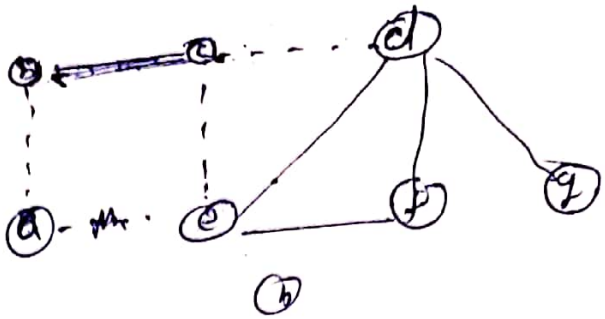
An edges is covered if one of its endpoint is chosen

the vertex cover problem is the minimum size vertex cover in G.

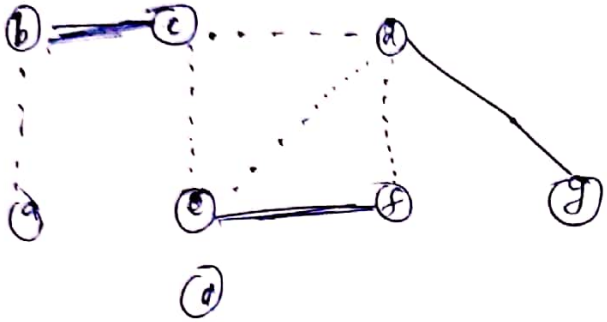
```
Approx-vertex-cover(G)
C ← φ
E' ← E[G]
while E' ≠ φ
    do let (u, v) be an arbitrary edge of E'
        C ← C ∪ {u, v}
        Remove from E' every edge incident on either u or v
return C.
```



(a)



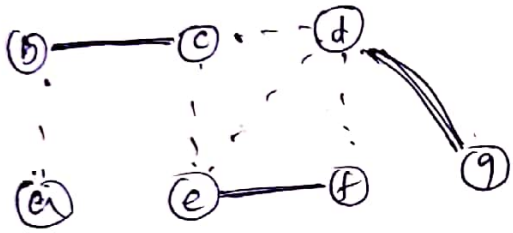
(b)



(c)

$C = [b, c, e, f, d, g]$.

optimal $C^* = [b, d, e]$



$\frac{C}{C^*} \leq P(n)$

$\frac{6}{3} \leq P(n)$

$= 2$

$O(V+E)$

Randomized Algorithms

An Algorithm that uses random numbers to decide what to do next anywhere in its logic called Randomized algorithm.

randomness is used to reduce time or space complexity

e.g.. Randomized Quicksort, approximate median:

deterministic algo-

input → [Algo] → output

the output and running depends on only input.

input → [Random Bit / Algo] → output

the output & Running time depends on input and Random bit number.

Types of Randomized algo.

1- Randomized Las Vegas Algo:

- output is always correct Random variables.
- Execution time depends on ~~output of Randomizer~~.

Ex: Randomized Quicksort

2. Randomized Monte Carlo Algo.

1. output may be incorrect.
2. running time is deterministic.

eg: Randomized approximate median.

identify Repeated Element:

No. of element in Array is 'n'

```
Algo Search Repeat[A]
{
for(i=0; i<n; i++)
   for(j=i+1; j<n; j++)
      if (A[i]==A[j])
         return true.
}
```

$O(n^2)$

```
Algo_LV_search Repeat(A)
{
while(true) do
   i = random() mod n+1
   j = random() mod n+1
if ((i≠j) and A[i]==A[j])
   return true.
}
```

$O(n \log n)$

```
Algo Search (A,a)   no of element=n.
{
for(i=0; i<n-1; i++)
   if (A[i]=a))
   return true.
}
```

$O(n)$

```
Algo MC_Search (A, a, n)   no. of iteration
{ i=0; flag=false
while (i<=n)
{ i = random() mod n+1
  i = i+1
  if (A[i]==a)
     flag = true.
}
return flag
```

```
Array mc-search(A, a, n).
    flag = false
for(i=0 ; i<=n ; i++)
{
    i = random() mod n+1
    if (A[i] = a)
        flag = true.
}
Return flag.
```

Fast Fourier Transform

The straight forward method of addition and multiplication of two polynomials of degree n takes $\Theta(n)$ and $\Theta(n^2)$ time respectively But the fast Fourier transform (FFT) can Reduce the time to multiply polynomial to $\Theta(n \log n)$.

A Fast Fourier Transform (FFT) is an efficient algorithm to compute the discrete Fourier transform (DFT) and its inverse.

A polynomials in the variable x over an algebraic field F is a representation of a function A(x) as a formal sum

$$A(x) = \sum_{j=0}^{n-1} a_j x^j$$

a_j is coefficients

* polynomial A in x is:
$a_0 x^0 + a_1 x^1 + a_2 x^2 + \cdots + a_{n-1} x^{n-1}$
* degree k is highest non zero coeffient is a_k.
* degree Bound is strictly greater than the degree of polynomial.
* the degree of a polynomial of degree bound n may not be any integer betn 0 and n-1.

Examples

$A(x) = x^3 - 2x - 1$ ⇒ A(x) has degree - 3
⇒ A(x) has degreebound, 4 5 6 ....
⇒ A(x) has coffiecient (-1, -2, 0, 1)

$A(x) = x^3 + x^2 + 1$ degree - 3
degree bound 4, 5 6 ...
Coffiecient (1, 0, 1, 1)

Representation of Polynomials

1- Coefficient Representation
2- Point value Representation

1- A coefficient Representation of polynomial $A(x) = \sum_{j=0}^{n-1} a_j x^j$ of degree bound n is a vector of coefficient

$$a = (a_0, a_1, a_2 \cdots a_{n-1})$$

$A(x) = 6x^3 + 2x^2 - 10x + 9 \quad (9, -10, 7, 6)$

$B(x) = -2x^3 + 4x - 5 \quad (-5, 4, 0, -2)$

* Evaluation $A(x)$ at given point x_0 consist of computing the value of $A(x_0)$ . . takes time $\Theta(n)$ by Horner's rule

$$A(x_0) = a_0 + x_0(a_1 + x_0(a_2 + \cdots x_0(a_{n-2} + x_0(a_{n-1}))\cdots))$$

Eg: $a(x) = a_0 x^0 + a_1 x^1 + a_2 x^2 \cdots\cdots a_{n-2} x^{n-2} + a_{n-1} x^{n-1}$

$$= a_0 + x(a_1 + x(a_2 + \cdots\cdots x(a_{n-2} + x(a_{n-1}))\cdots))$$

Horner's rule

2. Point Value Representation:

A point value representation of a polynomial $A(x)$ of degree bound n is a set of n point-value pairs

$((x_0, y_0)(x_1, y_1) \cdots (x_{n-1}, y_{n-1}))$ for all x_k are distinct and $y_k = A(x_k)$ for $k = 0, 1, \cdots n-1$

Eg: $A(x) = x^3 - 2x + 1$

x_k	0	1	2	3
$y_k = A(x_k)$	1	0	5	22

$\{(0,1)(1,0)(2,5)(3,22)\}$

Using Horners method n-point evaluation take $\Theta(n^2)$.

* The Inverse of evaluation is called interpolation based on Lagrange formula.. and take (n^2).

$$A(x) = \sum_{k=0}^{n-1} Y_k \frac{\prod_{j \neq k} (x - x_j)}{\prod_{j \neq k} (x_k - x_j)}$$

Example: Using Lagrange formula we interpolate the point value representation $\{(0,1), (1,0), (2,5), (3,22)\}$

$x_0 \quad x_1 \quad x_2 \quad x_3$

$Y_k = \{1, 0, 5, 22\}$.

$Y_0 \quad Y_1 \quad Y_2 \quad Y_3$

$$\Rightarrow 1 \frac{(x-1)(x-2)(x-3)}{(0-1)(0-2)(0-3)} = \frac{x^3 - 6x^2 + 11x - 6}{-6} = \frac{-x^3 + 6x^2 - 11x + 6}{6}$$

$$\Rightarrow 0 \frac{(x-0)(x-2)(x-3)}{(1-0)(1-2)(1-3)} = 0$$

$$\Rightarrow 5 \frac{(x-0)(x-1)(x-3)}{(2-0)(2-1)(2-3)} = 5 \frac{x^3 - 4x^2 + 3x}{-2} = \frac{-15x^3 + 60x^2 - 45x}{6}$$

$$= 22 \frac{(x-0)(x-1)(x-2)}{(3-0)(3-1)(3-2)} = 22 \frac{x^3 - 3x^2 + 2x}{6} = \frac{22x^3 - 66x^2 + 44x}{6}$$

$$= \frac{6x^3 + 0x^2 - 12x + 6}{6}$$

$$= \boxed{x^3 - 2x + 1}$$

Multiplying polynomials

if $A(x)$ and $B(x)$ are two polynomials of degree bound n the product $C(x) = A(x) \cdot B(x)$ is given by

$$C(x_k) = A(x_k)\, B(x_k) \quad \text{for any point } x_k.$$

if A and B of degree Bound n the C is degree bound $2n$. so extent A and B point value $2n$.

$$A\{(x_0\ y_0)\ (x_1\ y_1) \cdots x_{2n-1}\ y_{2n-1})\}$$

$$B\{(x_0\ y_0')\ (x_1\ y_1') \cdots x_{2n-1}\ y_{2n-1}')\}$$

$$C\{(x_0, y_0 y_0')\ (x_1, y_1 y_1') \cdots (x_{2n-1}\ x_{2n-1}\ y_{2n-1}')\}$$

taken $O(n)$ time.

example:

$$A(x) = x^3 - 2x + 1 \qquad B(x) = x^3 + x^2 + 1$$

$C(x) = A(x) \cdot B(x)$

$x_k = -3\ -2\ -1\ 0\ 1\ 2\ 3$

we need 7 coefficient $(2n-1)$

$2x+1 = 7$

$A: \{(-3,-20)(-2,-3)(-1,2)(0,1)(1,0)(2,5)(3,22)\}$

$B: \{(-3,-17)(-2,-3)(-1,1)(0,1)(1,3)(2,13)(3,37)\}$

$C: \{(-3,340)(-2,9)(-1,2)(0,1)(1,0)(2,65)(3,818)\}$

The Road so far

$a_0\ a_1\ a_2 \cdots a_{n-1}$
$b_0\ b_1 \cdots b_{n-1}$ —ordinary multiply time $\Theta(n^2)$→ $c_0\ c_1 \cdots c_{n-1}$

point evaluation | n^2

$A(x_0)\ B(x_0)$
$A(x_1)\ B(x_1)$
$A(x_{2n-1})\ B(x_{2n-1})$ —Point wise multiplication (n)→ $C(x_0)$ $C(x_1)$ (x_{2n-1})

Interpolation time $\Theta(n^2)$

$27, +9+1-1-$ $-8+4+1$ $(-3)^2 - (-3)+1$

$-27+6+1 = -20$

$-1+2+1 = 2$

$n^2 + n + n^2 = \Theta(n^2)$

using Fast Fourier transform and its inverse we can do evaluation and Interpolation in time $\Theta(n \log n)$.

Fast multiplication of polynomials

* Using complex root of unity.

* Evaluation by taking Discrete Fourier transform (DFT) of coefficient vector.
* Interpolation by taking the inverse DFT of point value pair
* Fast Fourier transform (FFT) can perform DFT and inverse DFT in $(n \log n)$ time

Algo
1- add n higher order zero coefficients to A(x) and B(x)
2- Evaluate A(x) and B(x) using FFT for 2n points.
3- Pointwise multiplication of point value form.
4- Interpolation C(x) using FFT to compute Inverse DFT.

Complex root of unity: A complex nth root unity is a complex number w such that

$$w^n = 1$$

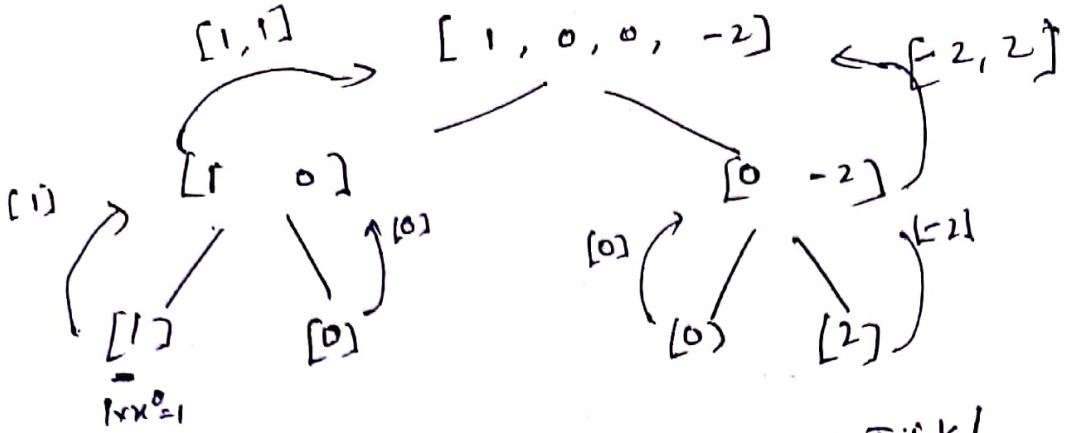
there are exactly n complex n^{th} root of unity

$$e^{2\pi ik/n} \text{ for } k = 0, 1, \ldots, n-1$$

$$e^{iu} = \cos(u) + i \sin(u).$$

the value $w_n = e^{2\pi i/n}$ is called principle nth root of unity.

$-2x^3+1 = 1\times x^0 + 0\times x^1 + 0\times x^2 - 2x^3$



$\omega_n = e^{2\pi i k/n}$

$k = 0, 1, 2 \ldots n-1$

$n=1$

$\omega_n = e^{2\pi i k/n}$	ω	k	a_0	Y_0
$= e^{2\pi i 0/n}$	1	0	[1]	$1\times x^0$
$= e^0 = 1$				$= 1\times(1)^0$
				$= 1$

$n=2$

$\omega_n = e^{2\pi i k/n}$	ω	k	a_0 a_1	Y_0 Y_1
$= e^{2\pi i k/n}$	1	0	[1 0]	$1\times x^0 + 0\times x^1$
$= e^0 = 1$				$= 1+1+0+1$
				$= 1$
$= e^{\pi i}$	-1	1		$1\times x^0 + 0\times x^1$
				$1\times(-1)^0 + 0\times -1$
				$=$
				$= 1$

[1 1]

$n=2$

$\omega_n = e^{2\pi i k/n}$	ω	k	a_0 a_1	Y_0 Y_1
	1	0	[0 -2]	$= -2$
	-1	1		$= 2$

$n=4$ $\quad \omega_n = e^{2\pi i k/n}$

	ω	k	a_0 a_1	Y_0 Y_1 Y_2 Y_3
$\begin{bmatrix} -1 \\ 1+2i \\ 3 \\ 1-2i \end{bmatrix}$	1	0	$\begin{bmatrix}1\\1\end{bmatrix}$ $\begin{bmatrix}-2\\2\end{bmatrix}$	-1 $\quad$ $1+2i$ $\quad$ 3 $\quad$ $1-2i$
	$+i$	1		
	-1	2		
	$-i$	3		