ALGORITHMS

Contents

CH1 : Analyzing the complexity of Algorithms

Algorithm: A finite sequence of instructions for solving a problem.

** *Two-Reasons to choose an Algorithm:*

1. Human Reason :
   To understand and implement the algorithm.

2. Machine Reason :
   Time , Space.

<u>Cost of algorithm :</u>

Cost = (Cost of understanding and programming + Cost of running the program)
= Software Cost + Hardware Cost

Hardware Cost = (Cost of running program once) * number of execution

(*Choosing the algorithm is dependent on the cost of the algorithm*)

** *How do you compare the efficiency of two algorithms (for one problem) ?*

1. Compare the execution time.
2. Compare the size of program or (algorithm)
   Size of program or algorithm is dependent on :
   - The number of lines.
   - The number of instructions.

** *It's better *

to measure the complexity of algorithm, that means (count number of basic operations).

<u>Type of Basic Operations :</u>

1. By Searching (Comparing or logical operations)….[< , <= , ≠ , > , >= , ==]
2. By arithmetic operations [+ , - , * , / , % , ++ , -- , ^ , *= , ….]

Three types to determine cost of algorithm :-

1. Worst case complexity
2. Best case complexity
3. Average case complexity

Suppose n >= 0 (The size of input) :

1. <u>Best case complexity :</u>
   $n >= 0$, $\forall$ I (instances of problem) define :
   B(n) = the minimum value of T(I) ,
   where T(I) the number of basic operations for instance I.

   (B(n) the Best case complexity of the algorithm

2. <u>Worst case complexity :</u>

   $n >= 0$, $\forall$ I (instances of problem) define :

   W(n) = the maximum value of T(I) ,

   where T(I) the number of basic operations for instance I.

   (W(n) the worst case complexity of the algorithm)

3. <u>Average complexity :</u>

   $A(n) = \sum P(I) * T(I)$

   Where P(I) is the probability that the instance I will occur and

   T(I) the number of basic operations for instance I.

➔ Two ways to define the notation of complexity of an algorithm to solve a class of problems (Worst or Average case copmlexity) :

Examples :

Example1 - Meaning of Instance I :

Suppose we have a one dim array length 10 containing different int keys

19	22	13	45	34	31	100	90	75	60
1	2	3	4	5	6	7	8	9	10

Problem : Searching a given key

Algorithm : Sequential Searching

➔ There are 11 Instances of this problem :

I_1 : Find first ➔ $T(I_1)$ = 1 B.O.

I_2 : Find second ➔ $T(I_2)$ = 2 B.O.

I_3 : Find third ➔ $T(I_3)$ = 3 B.O.

…

….

I_{10} : Find Last ➔ $T(I_{10})$ = 10 B.O.

I_{11} : Not found ➔ $T(I_{11})$ = 10 B.O.

Best case complexity = B(n) = B(10) = 1 B.O.

Worst case complexity = W(n) = W(10) = 10 B.O.

Average case complexity =

$A(n) = A(10) = \sum_{j=1}^{11} P(Ij) * T(Ij)$ = (1/11)*1 + (1/11)*2 + ...+(1/11)*10 + (1/11)*10 = ?

Example2 :

Analyze and find the worst case complexity of the following algorithm :

```
t = 1 ;
while ( t <= n )      // n is the input size
   {
     for ( int  i = 1  ;  i < n  ;  ++i )
        {
           Add  +=  i % t ;
           for ( int j = n  ;  j  >  1  ;  --j )
             P = P *3  ;
        }
     if (  X > 2)
          S.O.P(Y - 1);
     t  =  t + 1;

   }
```

Basic Op	Count
<=	**N**
<	n^2
++	n^2
+=	n^2
%	n^2
>	n^3
--	n^3
*	n^3
>	**N**
-	<= **n**
+	**N**

➔ $W(n) = 4n + 4n^2 + 3n^3$

Example3 :

Given following :

Problem with input size $n \to \infty$?

solved

Algorithm$_1$ (P_1) Algorithm$_2$ (P_2)

$W_1(n) = 100n$ $W_2(n) = 4n^2$

Two algorithms P_1, P_2 for solving the same problem with W_1 , W_2 as worst case complexities of both algorithms :

P1 : $W_1(n) = 100n$

P2 : $W_2(n) = 4n^2$

1. suppose the input size : $n < 25$

$n= 1 \Rightarrow W_1(n) = 100n = 100$ B.O. , $W_2(n) = 4n^2 = 4$ B.O.

$\Rightarrow$ In this case it's better to use P_2

$n= 2 \Rightarrow W_1(n) = 100n = 200$ B.O. , $W_2(n) = 4n^2 = 16$ B.O.

$\Rightarrow$ In this case it's better to use P_2 than P_1

$n= 3 \Rightarrow W_1(n) = 100n = 300$ B.O. , $W_2(n) = 4n^2 = 36$ B.O.

$\Rightarrow$ In this case it's better to use P_2 than P_1

……..

…….

$n = 24 \Rightarrow W_1(n) = 100n = 2400$ B.O. , $W_2(n) = 4n^2 = 2304$ B.O.

$\Rightarrow$ In this case it's better to use P_2 than P_1

2. Suppose the input size : $n = 25$ ➔

$W_1(n) = 100n = 2500$ B.O. , $W_2(n) = 4n^2 = 2500$ B.O.

$\Rightarrow$ In this case P_1 and P_2 are same

3. suppose the input size : $n > 25$

$n = 26 \Rightarrow W_1(n) = 100n = 2600$ B.O. , $W_2(n) = 4n^2 = 2704$ B.O.

$\Rightarrow W_2(n) > W_1(n)$

$\Rightarrow$ In this case it's better to use P_1 than P_2

In general :

$\Rightarrow \forall\, n : P_1$ better than P_2 , (using the same computer)

<u>Definitions :</u>

$f, g : N^+ \rightarrow R^+\backslash$ (Two positive real valued functions) Then :

1. $g(n)$ is $O(f(n))$ (*<u>read:</u> g(n) is big O of f(n)*)
   $\Leftrightarrow \exists\, k \in R\backslash\{0\}, n_0 \in N^+$ such that
   $g(n) \le k * f(n) \quad \forall\, n \ge n_0$

2. $g(n)$ is $\Omega(f(n))$ (*<u>read:</u> g(n) is big Omega of f(n)*)
   $\Leftrightarrow \exists\, k \in R\backslash\{0\}, n_0 \in N^+$ such that
   $g(n) \ge k * f(n) \quad \forall\, n \ge n_0$

3. $g(n)$ is $\theta(f(n))$ (*<u>read:</u> g(n) is big Theta of f(n)*)
   $\Leftrightarrow \exists\, k_1, k_2 \in R\backslash\{0\}, n_0 \in N^+$ such that
   $k_1 * f(n) \le g(n) \le k_2 * f(n) \quad \forall\, n \ge n_0$

** if $g(n)$ is $O(f(n))$ *but* $f(n)$ is not $O(g(n)) \Rightarrow O(g(n))$ better than $O(f(n))$.

$\rightarrow$

<u>That means</u> an algorithm with worst case complexity $g(n)$ runs faster than one with worst case complexity $f(n)$

$\Rightarrow$ an algorithm is <u>efficient</u> $\Leftrightarrow$ $W(n)$ is $O(n^k)$, where $k \in N\backslash\{0\}$.

Examples :

Ex1 :

Given two positive real functions $W_1(n)$ and $W_2(n)$, where

$W_1(n) = 100n$ and $W_2(n) = 4n^2$

Question : 1- $W_1(n)$ is $O(W_2(n))$? or
2- $W_2(n)$ is $O(W_1(n))$?

1- $W_1(n)$ is $O(W_2(n))$?

Solution :

Suppose $k = 1$ and $n_0 = 25$, $g(n) = W_1(n)$ and $f(n) = W_2(n)$ ➔

$g(n) \le k*f(n)$? $\forall\, n \ge n_0$

$W_1(n) \le 1*W_2(n)$? $\forall\, n \ge 25$

$100n \le 1*4n^2$? $\forall\, n \ge 25$

$25 \le n$? ➔ Yes $\forall\, n \ge 25$

➔ $W_1(n)$ is $O(W_2(n))$

2- $W_2(n)$ is $O(W_1(n))$?

Solution :

Suppose $k = 1$ and $n_0 = 25$, $g(n) = W_2(n)$ and $f(n) = W_1(n)$ ➔

$g(n) \le k*f(n)$? $\forall\, n \ge n_0$

$W_2(n) \le 1*W_1(n)$? $\forall\, n \ge 25$

$4n^2 \le 100n$? $\forall\, n \ge 25$

$n \le 25$? ➔ ***<u>NO</u>*** $\forall\, n \ge 25$

➔ $W_2(n)$ is ***<u>NOT</u>*** $O(W_1(n))$ ➔ $W_2(n)$ is **Ω** $(W_1(n))$

Ex2 :

$W_2(n)$ is **Ω** $(W_1(n))$?

Solution :

Suppose $k = 1$ and $n_0 = 25$, $g(n) = W_2(n)$ and $f(n) = W_1(n)$ ➔

$g(n) \ge k*f(n)$? $\forall\, n \ge n_0$

$W_2(n) \ge 1*W_1(n)$? $\forall\, n \ge 25$

$4n^2 \ge 100n$? $\forall\, n \ge 25$

$n \ge 25$? ➔ ***<u>YES</u>*** $\forall\, n \ge 25$ ➔ $W_2(n)$ is **Ω** $(W_1(n))$

Try with other values for k and n_0 , e.g. $k = 1/2$ and $n_0 = 50$ ….

Other Definitions (using limit) :

$f , g : N^+ \rightarrow R^+$ (Two positive real valued functions) Then :

1. g(n) is O(f(n)) $\Leftrightarrow \lim_{n\to\infty} g(n)/f(n) = c$,
   where $c \geq 0$, c nonnegative real number $c \in R^+$

2. g(n) is Ω(f(n)) $\Leftrightarrow \lim_{n\to\infty} g(n)/f(n) = c$,
   where c>0 , strictly positive real number
   OR $\lim_{n\to\infty} g(n)/f(n) = \infty$

3. g(n) is θ(f(n)) $\Leftrightarrow \lim_{n\to\infty} g(n)/f(n) = c$
   where $0 < c < \infty$, c positive real number

** If lim g(n) / f(n) = c , c >0 positive real number :
$\Rightarrow$ lim f(n)/ g(n) = 1/c , 1/c > 0
$\Rightarrow$ g(n) is O(f(n)) and f(n) is O(g(n))

** If lim g(n) / f(n) = 0
$\Rightarrow$ g(n) is O(f(n)) *but* f(n) is not O(g(n))
$\Rightarrow$ g(n) is better than f(n)

Examples : (classes of positive real functions)

1. Infinite constant functions
   (like g(n) = 1/2 , = 1/5 , = 7.5 , = 10000 , 10^{20} , ….)

2. Infinite log functions
   (Like g(n) = $\log_2 n$, 3.5*$\log_2 n$, 10000*$\log_2 n$, 5*$\log_2 n$+1000 -9.5 , …)

3. Infinite linear functions
   (Like g(n) = 100*n , …..)

4. Infinite linear log functions
   (Like g(n) = 7*n*$\log_2 n$, ….)

5. Infinite polynomial functions
   (like g(n) = n^2 , n^3 , n^5 , 15*n^4 – n*$\log_2 n$, …..)

6. Infinite expontial functions
   (Like g(n) = 2^n , 3^n , ….)

➔ 1. $\lim$ (any constant function) / $\log_2(n) = 0$,
likes $g(n) = 1/2$,$g(n) = 1000$, $g(n) = 10^{200}$
2. $\lim (\log_2 (n) / n) = 0$
3. $\lim (n / (n*\log_2 (n))) = 0$
4. $\lim ((n*\log_2 (n)) / n^2) = 0$
5. $\lim (n^p / n^q) = 0$ … If ($p < q$) and $p , q >= 3$
6. $\lim (n^p / 2^n) = 0$ … $\forall$ positive integer indices p

Efficiency of algorithms :

1. $O(1)$ [**constant functions**] is better than $O(\log_2 n)$
2. $O(\log_2 n)$ **[log functions]** is better than $O(n)$
3. $O(n)$ [**linear functions**] is better than $O(n \log_2 n)$
4. $O(n \log_2 n)$ [**log linear functions**]is better than $O(n^2)$
5. $O(n^p)$ [**polynomial functions**] is better than $O(n^q)$ …
if ($p < q$) and $p,q >= 2$
6. $O(n^p)$ [**polynomial functions**] is better than $O(2^n)$ …
$\forall$ positive integer indices $p >=0$

OR : \
$O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^p) \ (p > 2) < O(2^n)$
.

Example : **(Calculation the complexity of algorithms)**

Given following :

Problem with input size $n \to \infty$?

solved

Algorithm$_1$ (P_1) Algorithm$_2$ (P_2)

$W_1(n) = 100n$ $W_2(n) = 4n^2$

Two algorithms P_1, P_2 for solving the same problem with W_1 , W_2 as worst case complexities of both algorithms :

P_1 : $W_1(n) = 100n$
P_2 : $W_2(n) = 4n^2$

Which algorithm is better to use than the other ?

Solution :

P_1 : $W_1(n) = 100n$

$\Rightarrow \lim W_1(n)/n = \lim 100n/n = 100 \neq \infty \Rightarrow W_1(n)$ is $O(n)$

P_2 : $W_2(n) = 4n^2$

$\Rightarrow \lim W_2(n)/n^2 = \lim 4n^2/n^2 = 4 \neq \infty \Rightarrow W_2(n)$ is $O(n^2)$

Problem with input size $n \rightarrow \infty$?

↓

solved

↙ ↘

Algorithm$_1$ (P_1)	Algorithm$_2$ (P_2)
↓	↓
$W_1(n) = 100n$	$W_2(n) = 4n^2$
↓	↓
$W_1(n)$ is $O(n)$	$W_2(n)$ is $O(n^2)$

$\Rightarrow$ It is better to use P_1 than P_2 for all cases

Example :

Suppose we have an algorithm P with worst case complexity W(n) ,
Every basic operation costs τ times (the Algorithm written as Program runs on a machine)
T the used time to run the algorithm for the input n ,

$\Rightarrow T = W(n)*\tau$

when we solve the equation , we can know the maximum input size , which can be handled in T time .

Examples :

Ex1 :

Suppose τ

$\tau = 1$ ms ,
$W(n) = n^2$ ،
$T = 1$ hour

$\Rightarrow T = W(n)*\tau$

$\Rightarrow 60*60*sec = n^2 * 10^{-3}*sec$

$\Rightarrow$ $n^2 = 6 * 10^5$

$\Rightarrow$ $n = 600 * \sqrt{10} \approx 1897$ input size

Ex2 :

Given an algorithm with $W(n) = 2^n$ runs on two different machines so that the time for execution a basic operation for the first machine equal to τ and for the other one equal to τ/k , where $k >= 2$.
Calculate n_1 , n_2 maximum input size for two machines which can be handled in T time (Same time interval)

Problem?

$\downarrow$

Solved

$\downarrow$

Algorithm

$\downarrow$

$W(n) = 2^n$

$\downarrow$

Write a program

$\downarrow$

Runs on two different machines (Computers) using the same time (T)

$comp_1$ with speed = τ	$comp_2$ with speed = τ/k , $k > 1$
$\downarrow$	$\downarrow$
calculate n_1 ?	claculate n_2 ?

<u>Solution :</u>

Using the equation $T = W(n)*\tau$

$T = W(n_1)*\tau$ // running on $comp_1$

$T = \tau/k*W(n_2)$ // running on com_2

$\Rightarrow$

$W(n_1)*\tau = T = \tau/k*W(n_2)$

$\Rightarrow$ $W(n_2) = k*W(n_1)$

Now we have the complexity $W(n) = 2^n$

$W(n_2) = k*W(n_1)$

$\Rightarrow$ $2^{n2} = k*2^{n1}$ | using $\log_2$

$\Rightarrow$ $\log_2(2^{n2}) = \log_2(k*2^{n1})$

$\Rightarrow$ $n_2 = \log_2 k + n_1$

$\Rightarrow$ $n_2 > n_1$

CH2 : SORTING ALGORITHMS

Two types for sorting algorithms :

1. Internal sorting algorithms
2. External sorting algorithms

Declarations :

K = any ordered data type ;

T = a group of objects with key $\in K$

E a collection of T (like array or file of T)

$\Rightarrow$ If E small enough to fit into internal memory (algorithm called internal sorting algorithm).

Otherwise E too large $\Rightarrow$ sorting the elements of E in a file saved in external memory like hard disk,… (algorithm called external sorting algorithm).

Type of internal sorting algorithms :

1. Bubble sort
2. Insertion sort
3. Selection sort
4. Quick sort
5. Heap sort
6. Merge sort
7. Two Radix ……………….> The keys saved in a Queue

The keys saved in one dim Array A

Type of external sorting algorithms :

1. Balanced merge sort
2. Natural merge sort
3. Polyphase sorting

The keys saved in a file

** Searching in sorted elements costs $O(\log_2 n)$.

** Searching in unsorted elements costs O(n).

INTERNAL SORTING ALGORITHMS

Declaration :

*N = … (the size of the array **A** to be sorted)*

Index = 1… N;

Bubble sort :

The elements in the array will be sorted in (N-1) passes beginning with i = 2, in the first pass comparing A[N] with A[N-1] , if (A[N].key < A[N-1]) ⇒ swapping until we reach the comparing of A[i] with A[i-1].

Body of algorithm :

```
For i = 2 to N do
 {
   for j := N down to i do
    if A[j].key < A[j-1] then
      Swap( A[j-1], A[j]) ;
  }
```

```
void bubble ( int A[ ] , int n )
{ int tmp ;
  for ( int i = 2 ; i <= n ; ++i )
  { for ( int j = n ; j >= i : --j )
      if ( A[j] < A[j-1] )
        { tmp = A[j] ;
          A[j] = A[j-1];     SWAPPING
          A[j-1] = tmp;
        }
  }
}
```

Example:

Sort the following array of integers using Bubble sort.

1	2	3	4
8	2	6	4

1. Round (Outer loop) :

i = 2 ⇒ j = N to 2

j = 4 ⇒ swap (A[j],A[j-1])

1	2	3	4
8	2	**4**	**6**

j = 3 ⇒ nothing to do

1	2	3	4
8	2	4	6

j = 2 ⇒ swap (A[j],A[j-1])

1	2	3	4
2	**8**	4	6

2. Round (Outer loop) :

$\underline{i = 3}$ $\Rightarrow$ j = 4 to 3

j = 4 $\Rightarrow$ nothing to do

2	8	4	6
1	2	3	4

j = 3 $\Rightarrow$ swap (A[j],A[j-1])

2	**4**	**8**	6
1	2	3	4

3. Round (Outer loop) :

$\underline{i = 4}$ $\Rightarrow$ j = 4 to 4

j = 4 $\Rightarrow$ swap (A[j],A[j-1])

2	4	**6**	**8**
1	2	3	4

Complexity of bubble sort :

Value of i = 2 , 3 , 4 , … , N-1 , N

……….

No of comparisons for each round : N-1 , N-2 , N-3 … , 2 , 1

Comparisons in i-th round : (N - i + 1)

The number of comparisons = (N-1) + (N-2)+… + 3 + 2 + 1 = 1/2N(N-1)

$\Rightarrow$ worst case complexity of bubble sort W(N) = 1/2N(N-1) , is O(?)

lim $W(N)/N^2$ = lim $[1/2N(N-1)]/N^2$ = 1/2*lim (N-1)/N = 1/2 $\neq \infty$

$\Rightarrow$ W(N) = 1/2N(N-1) is $O(N^2)$

Insertion sort:

Idea :

We begin with *for i = 2 to N (N the number of elements)*
Comparing i-th element with the preceding entries with index (i-1) , (i-2) ,…,2 , 1 in the array until either we reach a smaller element or reach the left hand end of the array .

Body of algorithm :

```
void insertion ( int A[ ] , int n )
{  int i , j , x ;
   for (  i = 2 ; i <=  n ; ++i )
    {    x = A[i] ;
         A[0] = x ;
         j = i - 1 ;
         while  ( A[j] > x )
           {  A[j+1] = A[j] ;
              A[j] = x ;
              j - 1 ;
           }
    }
}
```

Example :

Sort the following array of integers using insertion sort.

8	2	6	4
1	2	3	4

1. Round :

i = 2

x = A[2] = 2

A[0] = 2

2	8	2	6	4
0	1	2	3	4

j = i-1 = 1

while : A[j] > x

8 > 2 → YES

→ A[j+1] = A[j]

A[2] = A[1] = 8

A[1]=2.

2	**2**	**8**	6	4
0	1	2	3	4

j := j - 1 = 0 ,

A[0] comparing with x → A[0] = x out of the loop.

2. Round :

i = 3

x = A[3] = 6

A[0] = 6

6	2	8	6	4
0	1	2	3	4

j = i -1 = 2

while : A[j] > x

8 > 6 → YES

→ A[j+1] = A[j]

A[3] = A[2] = 8 ,

A[2] = 6.

6	2	**6**	**8**	4
0	1	2	3	4

j = j -1 = 1;

j = 1

A[1] > x →

2 > 6 → **NO**

→ out of the loop

3. Round :

i = 4

x = A[4] = 4

A[0] := 4

4	2	6	8	4
0	1	2	3	4

j = i – 1 = 3

while : A[j] > x

8 > 4 → YES

→ A[j+1] = A[j]

A[4] = A[3] = 8

A[3] = 4

4	2	6	**4**	**8**
0	1	2	3	4

j = j – 1 = 2

while : A[j] > x

6 > 4 → YES

→ A[j+1] = A[j]

A[3] = A[2] = 6

A[2] = 4

4	2	**4**	**6**	8
0	1	2	3	4

$j = j - 1 = 1$

while : $A[j] > x$

$2 > 4 \rightarrow$ NO

$\rightarrow$ Out of while (STOP)

2	4	6	8
1	2	3	4

Complexity of insertion sort :

Value of i = 2 , 3 , 4 , … , N-1 , N

No of comparisons for each round : 1 , 2 , 3 , … , N-2 , N-1

Comparisons in i-th round : (i - 1)

The number of comparisons = (N-1) + (N-2)+… + 3 + 2 + 1 = 1/2N(N-1)

$\Rightarrow$ worst case complexity of insertion sort W(N) = 1/2N(N-1) is $O(N^2)$

Selection sort :

Idea :

For i = 1 to N-1 :

In the i_th round comparing the key in i_th position with the keys in the (i+1)_th, (i+2)_th ,…, N_th positions, each time we find a key less than the key in i_th position (swap).

Body of algorithm :

```
void selection ( int A[ ] , int n )
{  int i = 1 , k  ;
   int tmp , x  ;

  for (   ; i <=  n - 1 ;  ++i  )
    {   k = i ;
         x = A[i] ;
        for ( int j = i + 1 ; j <= n ; ++j )
          if ( A[j] < x )
           { k = j ;
              x = A[j] ;
           }
        tmp = A[k] ;
        A[k] = A[i] ;      swap(A[i] , A[k])
        A[i] = tmp ;
    }
}
```

Example:

Sort the following array of integers using selection sort.

8	2	6	4
1	2	3	4

1. Round :

i = 1

k = 1

x = A[1] = 8

inner loop :

j = i + 1 = 2 → A[2] < x → 2 < 8 → **YES** → k = j = 2

x = A[2] = 2

j = 3 → A[3] < x → 6 < 2 → **NO** (nothing to do)

j = 4 → A[4] < x → 4 < 2 → **NO** (noting to do)

swap(A[i] , A[k]) = swap(A[1],A[2]) :

2	**8**	6	4
1	2	3	4

2. Round :

i = 2

k = 2
x = A[2] = 8

inner loop :
j = i + 1 = 3 → A[3] < x → 6 < 8 → **<u>YES</u>**
→ k = j = 3
x = A[3] = 6

j = 4 → A[4] < x → 4 < 6 → **<u>YES</u>**
→ k = j = 4
x = A[4] = 4

swap(A[i] , A[k]) = swap(A[2],A[4]) :

2	**<u>4</u>**	6	**<u>8</u>**
1	2	3	4

<u>3. Round :</u>
<u>i = 3</u>
k = 3
x = A[3] = 6

inner loop :
j = i + 1 = 4 → A[4] < x → 8 < 6 → **<u>NO</u>**
→ (Nothing to do)

swap(A[i] , A[k]) = swap(A[3],A[3]) :

2	**<u>4</u>**	6	**<u>8</u>**
1	2	3	4

<u>Complexity of selection sort :</u>

Value of i = 1 , 2 , 3 , … , N-2 , N-1

No of comparisons for each round : N-1 , N- 2 , N - 3 , … , 2 , 1

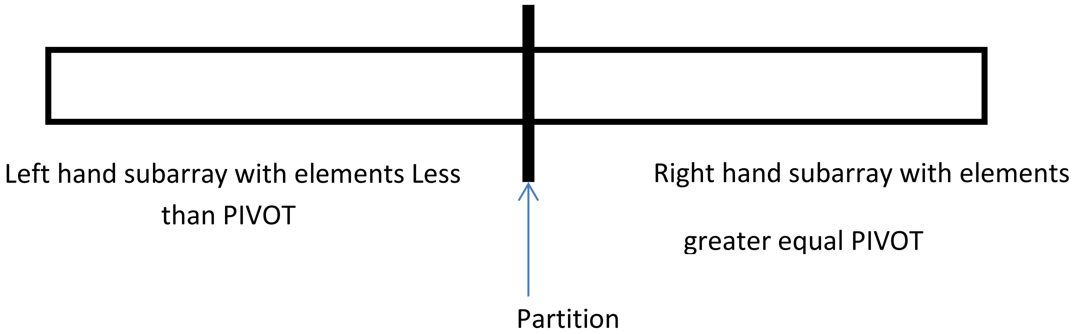
Comparisons in i-th round : (N - i)

The number of comparisons = (N-1) + (N-2)+… + 3 + 2 + 1 = 1/2N(N-1)

⇒ worst case complexity of insertion sort W(N) = 1/2N(N-1) is $O(N^2)$

Quick sort : (RECURSION)

Choose a key (***Pivot***) from given array then *:*



Idea :

Two stages :

1. Function ***find_pivot(i , j) ;*** // ***i*** is the index of the first element in the array and ***j*** is the last element in the array

2. Method : ***Partition (i , j , p , k);***
   the Method returns the index of the first element of the right hand subarray

3. Apply stage 1 and stage 2 recursively

Body of the algorithm :

```
quick( i , j )
 { p , k    // local
  if  i < j  then
   {
      p = find_Pivot (i,j);
      partition(i,j,p,k)
     quick(i,k-1);
     quick(k,j);
   }
 }
```

Tecniques to find Pivot :

1. *Random* to find ***m*** as positive of pivot in i..j ,
   →find_Pivot(i,j)→A[m]
2. define Pivot as *Middle element* in the array ,
   → find_Pivot(i,j) → A[(i+j)/2].
3. Small sample of elements in subarray then define find_Pivot(i,j)→ median of the keys .

……..

……..

1. First Stage : How to Find Pivot

Comparing the elements A[i],..,A[j] until we find two elements with different keys → choosing the larger of these as pivot .

Problem :
- The array contains only one element ?
- The keys of the array are the same ?

➔ We do nothing , because the array is sorted

```
public int pivotIndex( int A[] , int i , int j )
    { int z , p , q ;
      boolean found = false ;
      p = i-1;
      q = I ;
      do
       {  p = p + 1;
          q = q+1;
          if(A[p] != A[q])
             found = true;
         if(A[p] < A[q])
           z = q;
         else
            z = p;
         } while ( p!=j-1) && ( !found));
     if (!found) z = 0;
  return z;
 }
```

2. Second Stage : Partitioning the array

Idea of the partitioning :

1. Define two pointers left and right : ***left = i and right = j*** .
2. Right moving of left pointer , while A[left] **<** pivot
3. Left moving of right pointer while A[right] ***>=*** pivot
4. if left < right → swapping (A[left] , A[right]) .
5. No crossing by left and right pointers that means (right is still greater then left) → goto step 2 else goto step 6
6. k = left : means the left position of right subarray is equal to the value of left.

```
public int partition ( int A[] , int i , int j , int p)
   { int left , right , tmp;
     left = I;
    right = j;
    do
     { while ( A[left] < p)
          Left = left+1;
        while (A[right] >= p)
          right = right-1;
        if (left<right)
         { tmp = A[left];
           A[left] = A[right];
           A[right] = tmp;
          }
    } while(left > right);
 return left;
}
```

Apply stage 1 and stage 2 recursively

```
public void quick ( int A[] , int i , int j)
  { int p , k , n;
     n = pivotIndex(A,i,j);
     if ( ( n!=0)&&(j>i))
      {
        p = A[n];
        k = partition(A,i,j,p);
        quick(A,i,k-1);
        quick(A,k,j);
       }
  }
```

```
class QuickSort
 {
     public int pivotIndex( int A[] , int i , int j )
      { int z , p , q ;
         boolean found = false ;
         p = i-1;
         q = i ;
         do
          {  p = p + 1;
              q = q+1;
              if(A[p] != A[q])
                 found = true;
             if(A[p] < A[q])
               z = q;
             else
                z = p;
             } while ( p!=j-1) && ( !found));
       if (!found) z = 0;
   return z;
   }

 public int partition ( int A[] , int i , int j , int p)
   { int left , right , tmp;
      left = I;
     right = j;
     do
      { while ( A[left] < p)
             Left = left+1;
         while (A[right] >= p)
            right = right-1;
         if (left<right)
          { tmp = A[left];
             A[left] = A[right];
            A[right] = tmp;
           }
    } while(left > right);
 return left;
}

  public void quick ( int A[] , int i , int j)
   { int p , k , n;
     n = pivotIndex(A,i,j);
     if( ( n!=0)&&(j>i))
      {
         P = A[n];
         k = partition(A,i,j,p);
         quick(A,i,k-1);
         quick(A,k,j);
       }
  }
 }
```

Example :

left right

beg	Car	sup	and	the	bee	sum	pie
1	2	3	4	5	6	7	8

n = 2
p = A[n] = A[2] = car
k = partition(A,i,j,p) = partition (A, 1 , 8 , car)

partition (A, 1 , 8 , car) :

1- left = 1 , right = 8
2- while (A[left] < car)
beg < car ➔ yes ➔ left = left +1 =2
car < car ➔ no ➔ stop
3- while (A[right] >= car)
pie >= car ➔ yes ➔ right = right -1 = 7
sum >= car ➔ yes ➔ right = 7 - 1 = 6
bee >= car ➔ no ➔ stop
4- if (left < right)
2 < 6 ➔ yes ➔ swap (A[left] , A[right]) = swap(car , bee)

left right

beg	**Bee**	sup	and	the	**car**	sum	pie
1	2	3	4	5	6	7	8

5- while (left < right) [means : No Crossing] ➔ yes ➔ goto step 2

Again :

2- while (A[left] < car)
bee < car ➔ yes ➔ left = left +1 =3
sup < car ➔ no ➔ stop
3- while (A[right] >= car)
car >= car ➔ yes ➔ right = right -1 = 5
the >= car ➔ yes ➔ right = 5 - 1 = 4
and >= car ➔ no ➔ stop
4- if (left < right)
3 < 4 ➔ yes ➔ swap (A[left] , A[right]) = swap(sup , and)

left right

beg	Bee	**And**	**sup**	the	car	sum	pie
1	2	3	4	5	6	7	8

5- while (left < right) [means : No Crossing] → yes → goto step 2

<u>Again :</u>

2- while (A[left] < car)
    and < car → yes → left = left +1 =4
    sup < car → no → stop
3- while (A[right] >= car)
    sup >= car → yes → right = right -1 = 3
    and >= car → no → stop
4- if (left < right)
    4 < 3 → no → NO swap

left   right

beg	Bee	and	sup	the	car	sum	pie
1	2	3	4	5	6	7	8

5- while (left < right) [means : No Crossing]
    4 < 3 → no → → there is Crossing → goto step 6

6- k = left = 4 →

beg	Bee	and	sup	the	car	sum	pie
1	2	3	4	5	6	7	8

<u>Recursion :</u>

beg	Bee	and
1	2	3

Quick(A[] , 1 , 3)

n = 1
p = A[n] = A[1] = beg
k = partition(A,i,j,p) = partition (A, 1 , 3 , beg)
…….
……

sup	the	car	sum	pie
4	5	6	7	8

quick(A[] , 4 ,8)

n = 5
p = A[n] = A[5] = the
k = partition(A,i,j,p) = partition (A, 4 , 8 , the)
…….
….

Complexity of quick sort

1.Worst case Complexity (General):

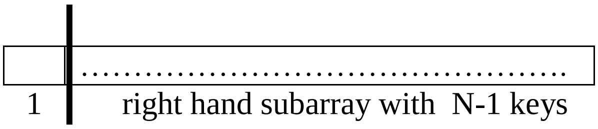
The number of comparisons **[** needed to partition an array of length N **] :**
is either N (if pivot is **NOT** one of the entries in the array)
or N-1 (if pivot is one of the entries in the array)

First Instance :

Apply quick sort using an array with following properties :

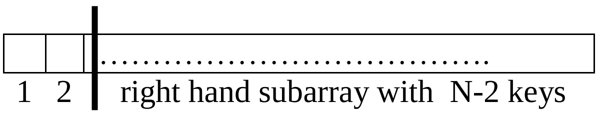
- Number of keys is equal to N
- Sorted keys.
- Different keys.
- The pivot is larger of the first two entries.

After the 1. Partition :



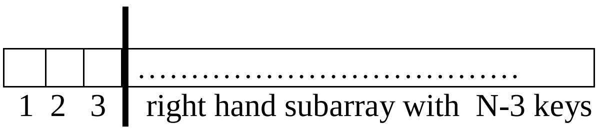
→ The number of comparsions =N-1

After the 2. Partition :



→ The number of comparsions = N-2

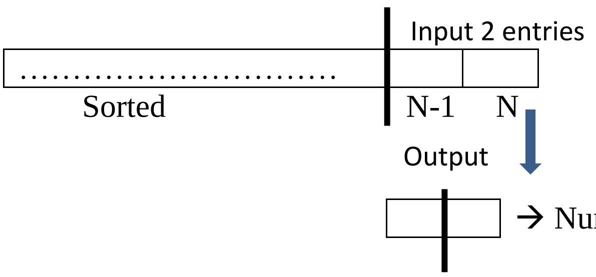
After the 3. Partition :



→ The number of comparsions = N-3

…………
………..

After the Last Partition :



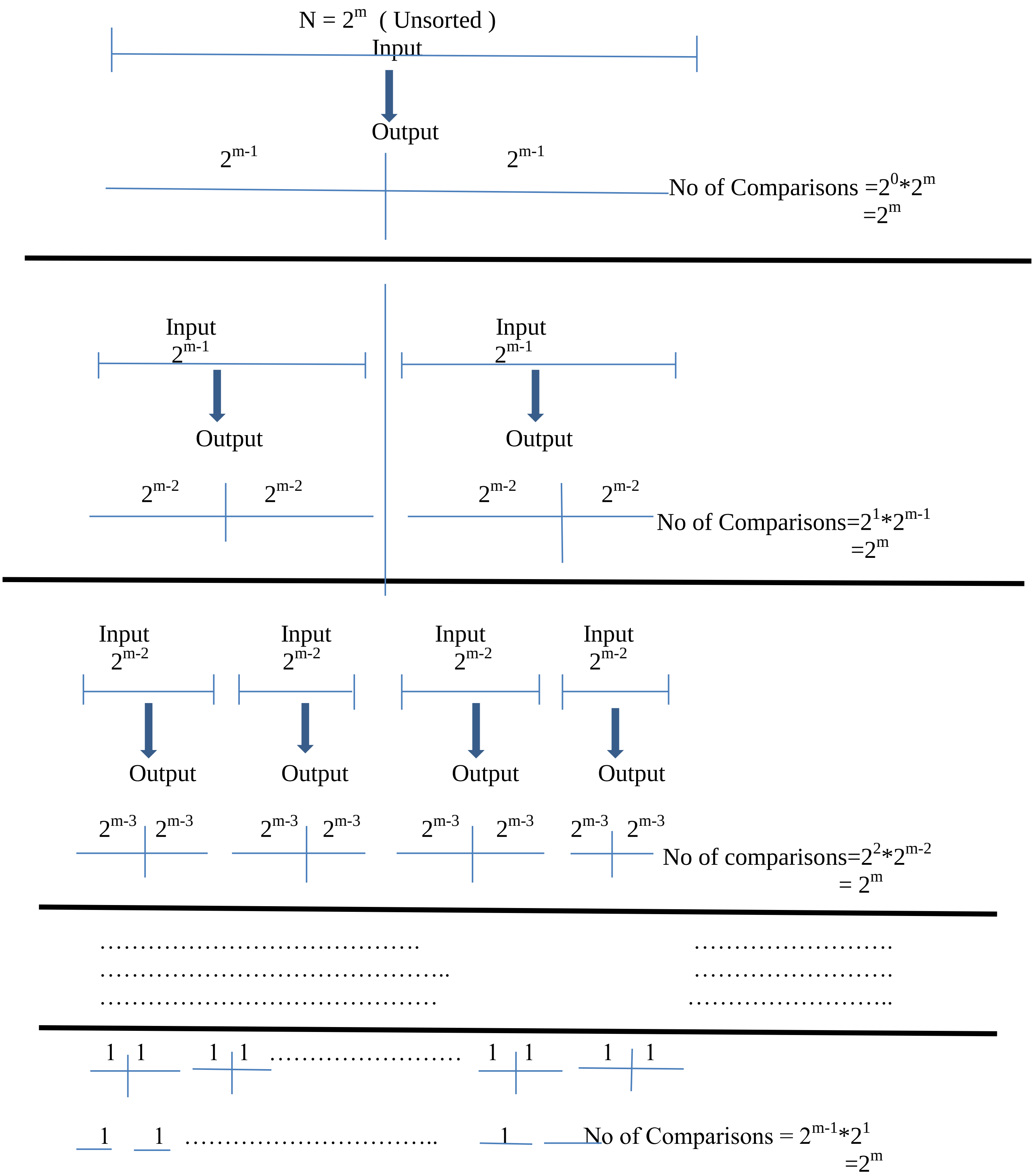
→ Number of comparisopns = 1

$\Rightarrow$ Worst case complexity of quick sort = (N-1)+(N-2)+…+2+1 = 1/2N(N-1)
$\Rightarrow$ the worst case complexity of quick sort is $O(N^2)$

Second Instance :

Apply quick sort using an array with following properties :

- Number of keys is equal to $N = 2^m$, where m >=1
- Unsorted keys.
- After each parition, the array will be divided into exactly equal parts.
- pivot is not one of the elements.



→ Worst case complexity $= \underbrace{2^m + 2^m + \ldots\ldots + 2^m}_{\text{m times}} = m*2^m = N*\log_2 N$

$N = 2^m$

➔ $m = \log_2 N$

➔ $w(N) = N*\log_2 N$ is $O(N\log_2 N)$

2. Average complexity of quick sort :

Suppose we have an unsorted array with different items and the pivot is one of its entries.

→ left subarray consists 1 or 2 or 3 ,…, N-1

Let A(N) average complexity of array length N :

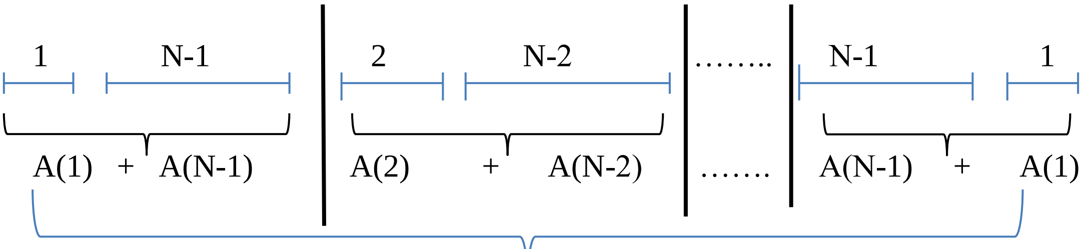
$A(N) = \sum P(I) * T(I)$

Where P(I) is the probability that the instance I will occur and

T(I) the number of basic operations for instance I

N = 1 ➔ A(1) = 0

N > 1 :



Number of Instances = N -1

➔ P(I) = 1/(N-1) // The probability

A(N) = (probability that left subarray consist 1 elements) * (A(1)+A(N-1)) +
(probability that left subarray consist 2 elements) * (A(2)+A(N-2)) +
………………………..
………………………..
………………………..
(probability that left subarray consist N-1 elements) * (A(N-1)+A(1))
+ (N-1) *// comparisons of non_pivot with pivot*

A(N) = 1/(N-1)*[A(1)+A(N-1)] + 1/(N-1)*[A(2)+A(N-2)] + … +
1/(N-1)[A(N-1)+A(1)] + (N-1)

= (N-1) + [A(1)+A(N-1) + A(2) + A(N-2)+… + A(N-1)+A(1)] / (N-1)

= (N-1) + [2*A(1) +… + 2*A(N-1)] / (N-1) …………………….. **(1)**

replacing N in **(1)** by N-1 :

A(N-1) = (N-2) + [2*A(1)+ … + 2*A(N-2)] / (N-2) ………………….. **(2)**

multiply **(1)** by (N-1) : ➔

$A(N)(N-1) = (N-1)^2 + 2[A(1) + \ldots + A(N-1)]$

$= N^2 - 2N + 1 + 2[A(1) + \ldots + A(N-1)]$ …………………….. **(3)**

multiply **(2)** by (N-2) : ➔

$A(N-1)(N-2) = N^2 - 4N + 4 + 2[A(1) + \ldots + A(N-2)]$…………………. **(4)**

(3) - (4) $\Rightarrow$

$A(N)(N-1) - \underline{A(N-1)(N-2)} = N^2 - 2N + 1 + 2A(1) + \ldots + 2A(N-2) + 2A(N-1)$

$- N^2 + 4N - 4 - 2A(1) + \ldots + 2A(N-2)$

$= 2N - 3 + 2A(N-1)$…………………………. **(5)**

➔ $A(N)(N-1) - A(N-1)*N + 2A(N-1) = 2N - 3 + 2A(N-1)$

➔ $A(N)(N-1) - N*A(N-1) = 2N - 3$ …………………… ……… **(6)**

Divide **(6)** by (N-1)*N :

$\Rightarrow$ $A(N)/N - A(N-1)/(N-1) = (2N-3)/[N*(N-1)]$

$\Rightarrow$ $A(N)/N - A(N-1)/(N-1) = 3/N - 1/(N-1)$ *(partition fraction)* …...**(7)**

Suppose $B(k) = A(k)/k$ *(k = 1…N)* , *B(1) = A(1)/1 = 0/1 = 0*

Replace A(N)/N by B(N) and A(N-1)(N-1) by B(N-1) in equation **(7)** :

$\Rightarrow$ $B(N) - B(N-1) = 3/N - 1/(N-1)$ ……………………………………..**(8)**

In equation 8 : Replacing N in this equation by (N-1) , N-2 , .. , 2 and add up :

<u>N-1</u> ➔ $B(N-1) - \cancel{B(N-2)} = 3/(N-1) - 1/(N-2)$

<u>N-2</u> ➔ $\cancel{B(N-2)} - \cancel{B(N-3)} = 3/(N-2) - 1/(N-3)$

……..…

……….

<u>3</u> ➔ $\cancel{B(3)} - \cancel{B(2)} = 3/3 - 1/2$

<u>2</u> ➔ $\cancel{B(2)} - B(1) = 3/2 - 1$

$\Rightarrow$ $B(N-1) - B(1) = 3/(N-1) - 1/(N-2) + 3/(N-2) - 1/(N-3) + \ldots + 3/3 - 1/2 + 3/2 - 1$.**(9)**

In **(9)** Replacing (N-1) → N :

→ $B(N) = \underline{\mathbf{3/N}} - 1/(N-1) + \underline{\mathbf{3/(N-1)}} - 1/(N-2) + \ldots + \underline{\mathbf{3/3}} - 1/2 + \underline{\mathbf{3/2}} - 1$

$= \mathbf{3(1/2 + 1/3 + \ldots + 1/(N-1) + 1/N)} - (1+1/2+1/3+\ldots+1/(N-2)+1/(N-1))$

$\mathbf{+\ 1/N - 1/N + 3 - 3}$

$= \mathbf{3(1+1/2 + 1/3 + \ldots + 1/(N-1) + 1/N)} -$

$(1+1/2+1/3+\ldots+1/(N-2)+1/(N-1)+\mathbf{1/N})$

$\mathbf{+\ 1/N - 3}$

$= 2(1+1/2+\ldots+1/N) + \mathbf{1/N - 3}$ ....**(10)**

$LN(N) = 1/N + 1/(N-1) + \ldots + 1/2 + 1$ (definition of logarithm)
→ $LN(N) = 0.693*\log_2 N$

Using LN(N) in **(10)** :

⇒ $B(N) = 1.4*\log_2 N + 1/N - 3$ ($B(N) = A(N)/N$)

⇒ $A(N)/N = 1.4*\log_2 N + 1/N - 3$

⇒ $A(N) = 1.4N*\log_2 N + 1 - 3N$

⇒ ***Average complexity of quick sort is $O(N*\log_2 N)$***

Heap sort:

Definition (Heap array) :

A Heap Array H is a one dimensional array with length N. ***(refer to the definition in Data Structure)***

For any index i : 1…..N

H[i/2] > H[i] > max (H[2*i] , H[2*i+1])

Example :

Heap Array indexed by 1..15

H	96	90	70	80	75	42	60	17	44	10	72	14			
	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Idea of Heap Sort:

1. Make heap array .
2. SWAP (first element , last element).
3. Make heap array of the remains (N-1) elements.
4. Goto to step (2).
5. Make heap array of the remains (N-2) elements.
6. Goto to step (2).
7. And so on until the array is sorted.

●**Make Heap (Heapifying)**

Idea :

1- Calculate N : No of keys in the array.

2- Starting at index i = N/2 : ***(Outer loop)***

A- if (H[i] < max (H[2*i] , H[2*i+1])

➔ ***Trickling down*** : swap (H[i] , max (H[2*i] , H[2*i+1]))

B- Calculate new index j of H[i] : **(if H[j] has any children) [inner loop]**

if (H[j] < max (H[2*j] , H[2*j+1])

➔ ***Trickling down*** : swap (H[j] , max (H[2*j] , H[2*j+1]))

C- Calculate i = i -1

3- while (true \\ i >= 1) goto step 2 else

4- Stop

Example:

Sort the following array of integers using Heap sort.

5	10	27	60	59	62	14	73
1	2	3	4	5	6	7	8

Make heap array →

73	60	62	10	59	27	14	5
1	2	3	4	5	6	7	8

Swap (first with last)

5	60	62	10	59	27	14	73
1	2	3	4	5	6	7	8

Make heap array for (N-1) elements →

62	60	27	10	59	5	14	73
1	2	3	4	5	6	7	8

Swap (first with last)

14	60	27	10	59	5	62	73
1	2	3	4	5	6	7	8

Make heap array for (N-2) elements →

60	59	27	10	14	5	62	73
1	2	3	4	5	6	7	8

Swap (first with last)

5	59	27	10	14	60	62	73
1	2	3	4	5	6	7	8

…

…

…

→

Then the array is sorted :

5	10	14	27	59	60	62	73
1	2	3	4	5	6	7	8

Complexity of Heap Sort :

1. Heaping costs 2.5*N
2. TrickleDown (1, j, A) cost $2\log_2 j$ each time
3. The number of executions trickle is

$$2\log_2(N-1) + 2\log_2(N-2) + \ldots + 2\log_2 1 = 2\log_2(N-1)!$$
$$\approx N*\log_2 N+3N$$

$\Rightarrow$ *Worst case complexity Of heap sort is $O(N*\log_2 N)$.*

Merge sort :

Idea:

1. A , B two sorted arrays with length N , M
2. Define an array C of length N + M
3. Comparing each element of A with each element of B :

```
        if A[i] ≤ B[j] then
         {
           C[k] = A[i];
           i = i +1;
          }
        else
         {
           C[k] = B[j];
           j = j +1;
        }
     k=k+1;
```

Where i = 1 to M , j = 1 to N and k = 1 to N+M

4. A empty and B not empty → copy the remains of B to C
   else B empty and A not empty → copy remains of A to C

A

1	2	..	M

B

1	2	…….	N

C

1	2	…….	…….	……	M+N

Examples :

A

11	13	15
1	2	3

B

12	14	16
1	2	3

C

11	12	13	14	15	16
1	2	3	4	5	5

A

10	14	16	20	21
1	2	3	4	5

B

14	14	25	26
1	2	3	4

C

10	14	14	14	16	20	21	25	26
1	2	3	4	5	6	7	8	9

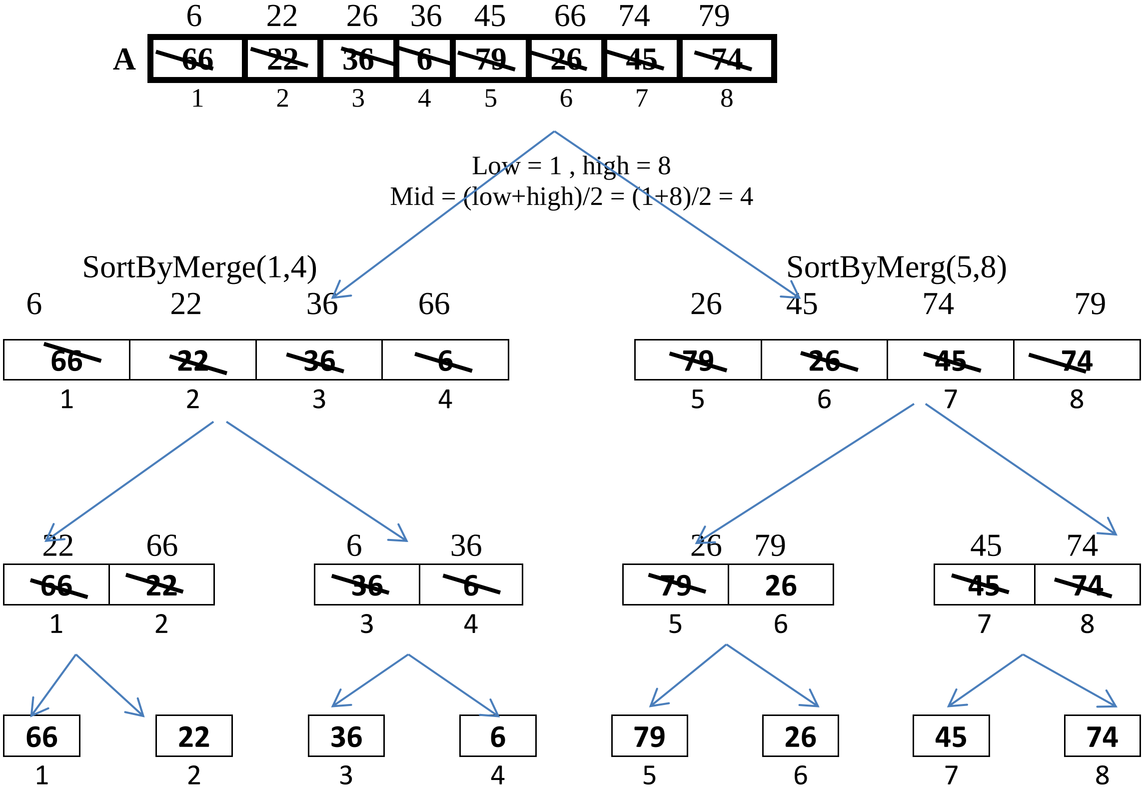
// Algorithm

```
void merge (A , B , C ) // merge A and B into C
{ i : 1..M;
  j : 1..N;
  k : 1..N+M;
  int l ;

  i = 1; j = 1 ; k = 1;
  while ( (i ≤ M) and (j ≤ N) )
   { if ( A[i] ≤ B[j] )
         { C[k] = A[i];
           i = i + 1;
         }
     else
        { C[k] := B[j];
          j = j+1; }
     k = k+1;
   }

 if ( i > M )
    for l = j to N do
      {
        C[k] = B[l];
        k = k+1; }

if ( j > N)
    for l := i to M do
      {
        C[k] := A[l];
        k := k+1; }
}
```

```
void sortByMerge( low, high : Index)
 { mid : low..high;
    B  : array[low..high] of T;

    if ( low < high )
     { mid = (low+high)/2 ;
       sortByMerge ( low , mid ) ;
       sortByMerge ( mid+1 , high ) ;
       merge (A[low..mid] , A[mid+1..high]) , B );
       copy B to A[low.. high]
     }
 }
```

Example :

Sort the following array using merge sort

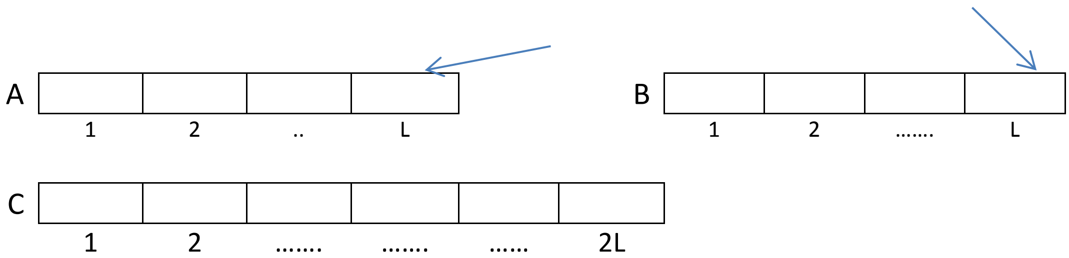


Complexity of Merge Sort :

By merge two sorted arrays length L1 , L2 $\Rightarrow$ cost is proportional to L1 + L2

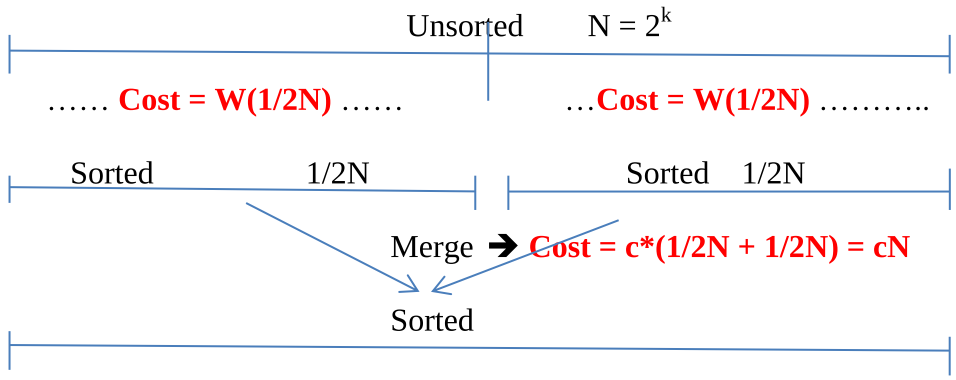
➔ c(L1+L2) or*

2L-1 (exactly) , if A and B same length L.



Suppose N = 1 ➔ W(1) = a , to sort an array length 1

N > 1, N = 2^k , where k >=1 :



➔ $W(N) = W(1/2N) + W(1/2N) + cN$, constant

$W(N) = 2^{1} * \underline{\mathbf{W(1/2^{1}N)}} + 1cN$

To calculate $W(1/2N)$:

Unsorted 1/2N

…… **Cost = W(1/4N)** …… …**Cost = W(1/4N)** ………..

Sorted 1/4N Sorted 1/4N

Merge ➔ **Cost = c*(1/4N + 1/4N) = c*1/2N**

Sorted

➔ $W(1/2N) = W(1/4N) + W(1/4N) + c*1/2N = \underline{2W(1/4N) + 1/2cN}$

Replacing

$W(N) = 2*\underline{\mathbf{W(1/2N)}} + cN$

➔

$W(N) = 2\,[2W(1/4N) + 1/2cN] + cN$

$= 2^{2}\, W(1/4N) + cN + cN$

$= 2^{2}\, \underline{W(1/2^{2}N)} + 2cN$

$= 2^{2}\, \underline{[2W(1/8N) + 1/4cN\,]} + 2cN$

$= 2^{3}\, W(1/2^{3}N) + 3cN$

…

…

…

$= 2^{k} W(1/\underline{2^{k}} * N) + k*cN$, $N = 2^{k}$

$= 2^{k} \underline{W(1)} + \underline{k}*cN$, $W(1) = a$, $k = \log_2 N$

$= \underline{2^{k}}\,\mathbf{a} + \mathbf{k}*cN = aN + cN*\log_2 N$

⇒ *Worst case complexity of merge sort is $O(N*\log_2 N)$*

OTHER INTERNAL SORTING ALGORITHM

** Sorting the keys of a queue with values between 0..99

Two Pass Radix Algorithm :

First pass:
Test the key by MOD function, then enqueuing this key in **Qu** indexed by the least significant digit of its key,where Qu is defined as one array length 10 containing 10 queues.

`

```
While ( !empty(Q) )
{
    dequeue (Q , x) ;
    j =  x % 10;
    engueue (x ,Qu[j]) ;
}
```

➔ Concatenate the queues Qu[0] , Qu[1],…, Qu[9] to the queue Q

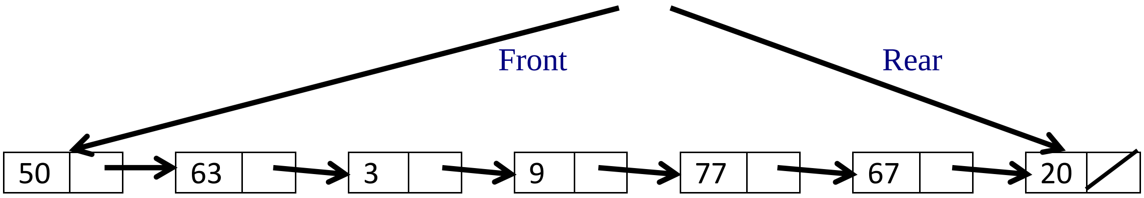
Second pass :
Test the key by DIV function, then enqueuing this key in **Qu** indexed by the most significant digit of its key .

```
While ( !empty(Q) )
{
    dequeue (Q , x) ;
    j =  x / 10;
    engueue (x ,Qu[j]) ;
}
```

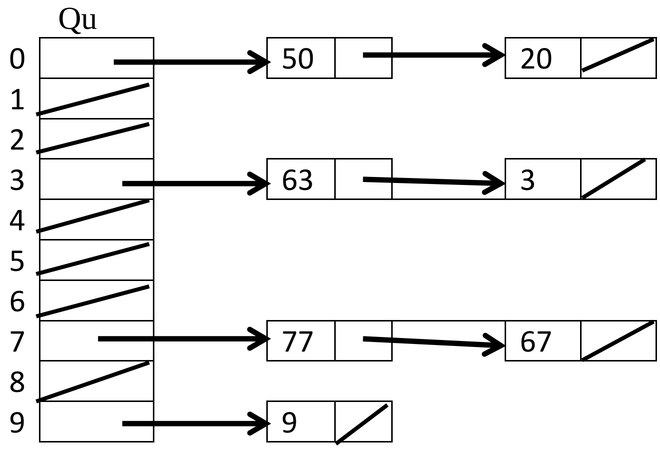
➔ Concatenate the queues Qu[0] , Qu[1],…, Qu[9] to one queue Q

Example :

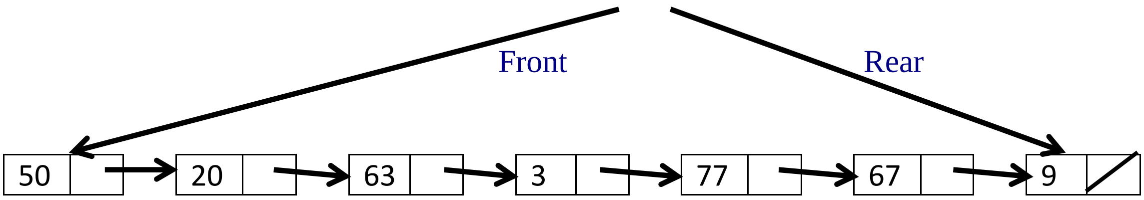
50 / 63 / 03 / 09 / 77 / 67 / 20



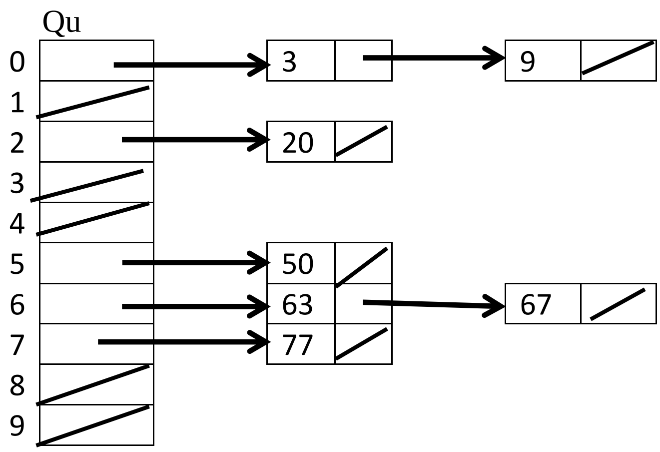
First Pass :



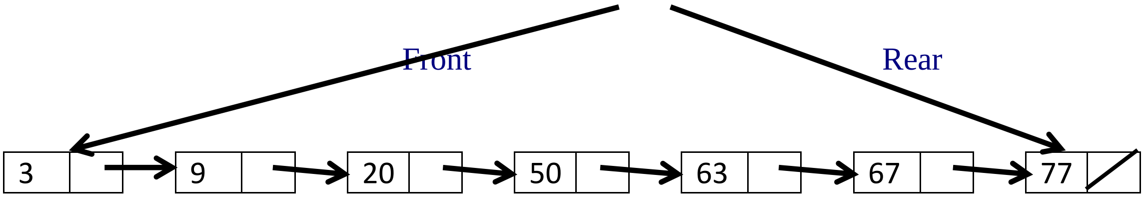
➔ Concatenation the 10 queues Qu[0] , Qu[1] , … , Qu[9] to Q ➔



Second Pass :



➔ Concatenation the 10 queues Qu[0] , Qu[1] , … , Qu[9] to Q ➔



<u>Complexity :</u>

1- Extraction of one of the digits of keys ➔ 2*N
2- Enqueuing each element with proper place ➔ cost is proportional to N (after each pass).
3- Concatenation cost (only concatenating the next field of Qu[i] to Qu[i+1]

➔ *Worst case complexity of Two Pass Radix algorithm is O(N) .*

EXTERNAL SORTING ALGORITHMS

1) Blanaced Merge Sort algorithm :

Idea :

- Define 5 files :
  F as master file containing the keys, and
  F1 , F2 , F3 , F4 as help files
- Number of keys in **F** is equal to **N**
- Define one dim array (called **RUN**) length **M** , where **M << N**
- Define **r = N/M** , number of RUNS
- Use any internal Sort Algorithms (heapSort….., with worst case in $O(N\log_2 N)$)

Body of B.M.S. Algorithm :

Containing two Stages :

1- Distribution stage
2- Merge Stage

1- Distribution stage :

1- Open F read only , open F1 , F2 rewriting : **RESET(F) ; REWRITE(F1 , F2)**
2- Read from F , M keys in one dim array **(RUN)** in the internal memory
3- Sort the keys in this array (using any internal algorithm, like heapSort, …)
4- Write the sorted keys in this array in F1 or F2 (alternately)
5- Goto step **(2)** until **F** eof.

2- Merge Stage :

1- Open F1,F2 read only , rewriting F3 , F4 : **RESET (F1, F2) ; REWRITE(F3 , F4).**
2- While not eof(F1) and not eof(F2) do
  - Read 2 RUNS of F1 and F2
  - Merge these and write the new RUN to F3 or F4 (using F3 , F4 files alternately)
  - Goto step (2)
  - If one of the files (F1,F2) empty and the other still contains the last RUN
    ➔ Read this RUN from the file (F1 or F2) then write it to (F3 or F4)
3- **Again :** Using (F1,F2) and (F3,F4) alternately for reading and rewriting until
    there is only on RUN in one file of (F1,F2,F3,F4)
    ➔ One of these files contains the sorted keys .

Example :

F: 20 , 8 , 5 , 17 , 21 , 9 , 3 , 11 , 2 , 18 , 15 , 23 , 14 , 6 , 15 , 24 , 10 ,21 ,13 , 16 ,19 , 15 , 22 , 5 ,18 , 20 , 8 , 5 , 12 , 1 , 26 , 25 , 4 , 15 , 7

N = 35
Define **M = 4 :** r = N/M = 9

RUN | 20 | 8 | 5 | 17 | Sorting ➔ RUN | 5 | 8 | 17 | 20 |

Using any internal algorithm

And so on …..

DISTRIBUTION STAGE :

RESET (F) , REWRITE(F1) , REWRITE(F2)

F1 : 5 8 17 20 / 2 15 18 23 / 10 13 16 21 / 5 8 18 20 / 4 7 15

F2 : 3 9 11 21 / 6 14 15 24 / 5 15 19 22 / 1 12 25 26

MERGE STAGE :

First round :
RESET (F1 , F2) , REWRITE (F3 , F4)

F3 : 3 5 8 9 11 17 20 21 / 5 10 13 15 16 19 21 22 / 4 7 15

F4 : 2 6 14 15 15 18 23 24 / 1 5 8 12 18 20 25 26

Second round :
RESET (F3 , F4) , REWRITE(F1 , F2)

F1 : 2 3 5 6 8 9 11 14 15 15 17 18 20 21 23 24 / 4 7 15
F2 : 1 5 5 8 10 12 13 15 16 18 19 20 21 22 25 26

Third round :
RESET (F1 , F2) , REWRITE(F3 , F4)

F3 : 1 2 3 5 5 5 6 8 8 9 10 11 12 13 14 15 15 15 16 17 18 18 19 20 20 21 21 22 23 24 25 26
F4 : 4 7 15

Fourth round :
RESET (F3 , F4) , REWRITE(F1 , F2)

F1 : 1 2 3 4 5 5 5 6 7 8 8 9 10 11 12 13 14 15 15 15 15 16 17 18 18 19 20 20 21 21 22 23 24 25 26
F2 : empty

Complexity of Balanced Merge sort :

N number of keys in F to be sorted
M length of RUNS in distribution stage
➔ N/M = r number of RUNS at the distribution stage
suppose $r = 2^k$

After each round of merge stage
1- Length of RUN doubled
2- Number of RUNS halved

1- Worst Case Complexity of Distr. Stage :

Cost for sorting one RUN (using internal sorted algorithm, like HeapSort) is equal to : $\mathbf{A*M*log_2M}$, where A constant .

➔ Total Cost of RUNS is equal to : $\mathbf{r*(A*M*log_2M)}$

2- Worst Case Complexity of Merge Stage :

Remember :

By merge two sorted arrays length L1 , L2 $\Rightarrow$ cost is proportional to L1 + L2
➔ *c*(L1+L2) or*
***2L-1** (exactly) , if A and B same length L.*

1. Round :
Merge $r/2^1$ pairs of RUNS length 2^0M ➔

Output : 1- RUNS length 2M
2- Number of RUNS $r/2^1$
➔ **Cost** = $r/2^1(2^1M – 1)$

2. Round :
Merge $r/2^2$ pairs of RUNS length 2^1M ➔

Output : 1- RUNS length 2^2M
2- Number of RUNS $r/2^2$
➔ **Cost** = $r/2^2(2^2M – 1)$

3. Round :
Merge $r/2^3$ pairs of RUNS length 2^2M ➔

Output : 1- RUNS length 2^3M
2- Number of RUNS $r/2^3$
➔ **Cost** = $r/2^3(2^3M – 1)$

....
....

k. Round :
Merge $r/2^k$ pairs of RUNS length $2^{k-1}M$ ➔

Output : 1- RUNS length 2^kM
2- Number of RUNS $r/2^k$
➔ **Cost** = $r/2^k(2^kM – 1)$

⇒ Complexity by Merge Stage **:**

$r/2^{1}(2^{1}M - 1) + r/2^{2}(2^{2}M - 1) + r/2^{3}(2^{3}M - 1) + \ldots\ldots + r/2^{k}(2^{k}M - 1)$
$= \underline{\mathbf{r*M}} - r/2^{1} + \underline{\mathbf{r*M}} - r/2^{2} + \underline{\mathbf{r*M}} - r/2^{3} + \ldots\ldots + \underline{\mathbf{r*M}} - r/2^{k}$
$= \underline{\mathbf{k*r*M}} - r*(1/2+1/4+\ldots+1/2^{k})$

$< \mathbf{k*r*M} - r*(1/2+1/4+\ldots+1/2^{k}) + \mathbf{r*(1/2+1/4+\ldots+1/2^{k})}$
$= \mathbf{k*r*M}$ Replace r*M by N and $k = \log_2 r$, where $r = 2^{k}$
$= N*\log_2 r$

Multiply with A ⇒
Cost of Merge Stage = $\mathbf{A*N*\log_2 r}$

➔ The complexity of Balanced Merge sort =
the complexity of Distr. Stage + the complexity of Merge stage
$= A*M*r*\log_2 M + A*N*\log_2 r$
$= A*N*\log_2 M + A*N*\log_2 r$
$= A*N (\log_2 M + \log_2 r)$
$= A*N * \log_2(M*r)$
$= A*N \log_2 N$

➔ *The worst case complexity of B.M.S is* $O(N*\log_2 N)$

2) Polyphase Sorting algorithm :

Fib number :

$$Fib : N^+ \longrightarrow N^+$$

Fib(n) = n ,   if   n = 0 or n = 1
Fib(n) = Fib(n-1) + Fib(n-2)   n>= 2

Fib(0) = 0 .
Fib(1) = 1 .
Fib(2) = F(1) + F(0) = 1 .
…………..
………….
F(n)= F(n-1) + F(n-2).

N	0	1	2	3	4	5	6	7	8	9	10	….
FIB(n)	0	1	1	2	3	5	8	13	21	34	55	….

PreCondition :

Suppose T a file contains r sorted RUNS (using any internal algorithm), where r any fib number **(r = FIB(n) = FIB(n-1) + FIB(n-2)) :**

Body of Algorithm (Distribution and Merge Stages) :

1- Create 3 writing files T1 , T2 , T3 , choose two of them for rewriting **REWRITE(T1) , REWRITE(T2)** , and open T read only : **RESET(T)**
2- Read FIB(n-1) RUNS from T then write to T1 and read FIB(n-2) RUNS then write to T2
3- REWRITE (T3) , RESET(T1) , RESET(T2)
4- Merge FIB(n-2) Pairs of RUNS from T1 , T2 writing to T3 ➔ T2 empty , T1 still contains FIB(n-1)-FIB(n-2) = FIB(n-3) RUNS
5- REWRITE (T2) , RESET(T3) and so on until all RUNS in one file sorted

Semi Example :

Let T contains r = FIB(8) = 21 RUNS (after sorting the record in T using any internal sort algorithm)

Distribution Stage :

REWRITE (T1) , REWRITE(T2) , RESET(T)
T1 : FIB(7) = 13 RUNS
T2 : FIB(6) = 8 RUNS

Merge Stage :

1.Round :

RESET(T1) , RESET(T2) , REWRITE(T3)
Merge FIB(6) = 8 RUNS into T3 $\Rightarrow$
T1 : FIB(5) = 5 RUNS
T2 : empty
T3 : FIB(6) = 8 RUNS

2.Round

RESET(T3) , RESET(T1) , REWRITE(T2)
Merge FIB(5) = 5 RUNS into T2 $\Rightarrow$
T1 : empty
T2 : FIB(5) = 5 RUNS
T3 : FIB(4) = 3 RUNS

3.Round

RESET(T2) , RESET(T3) , REWRITE(T1)
Merge FIB(4) = 3 RUNS into T1 $\Rightarrow$
T1 : FIB(4) = 3 RUNS
T2 : FIB(3) = 2 RUNS
T3 : empty

4.Round

RESET(T1) , RESET(T2) , REWRITE(T3)
Merge FIB(3) = 2 RUNS into T3 $\Rightarrow$
T1 : FIB(2) = 1 RUNS
T2 : empty
T3 : FIB(3) = 2 RUNS

5.Round

RESET(T1) , RESET(T3) , REWRITE(T2)
Merge FIB(2) = 1 RUNS into T2 $\Rightarrow$
T1 : empty
T2 : FIB(2) = 1 RUNS
T3 : FIB(1) = 1 RUNS

6.Round

RESET(T2) , RESET(T3) , REWRITE(T1)
Merge FIB(1) = 1 RUNS into T1 $\Rightarrow$
T1 : FIB(1) =1 RUN
T2 : empty
T3 : empty

Example :

Given following File T containing following keys :

T : 20-8-5-17-21-9-3-11-18-15-23-22-14-6-15-24-10-21-8-15-18-13-16- 6-6-25-24 -11-5

Pre-Calculations :

N = 29
M = ? (needs algorithm)
r = N/M = any Fib() number (depends on M)

How to find M :

M = 2 → r = N/M = 29/2 = 15 is not a Fib number
M = 3 → r = N/M = 29/3 = 10 is not a Fib number
M = 4 → r = N/M = 29/4 = 8 is a Fib number → 8= Fib(6) = Fib(5) + Fib(4)

T : 5-8-17-20/3-9-11-21/15-18-22-23/6-14-15-24/8-10-15-21/6-13-16-18/6-11-24-25/5
(Runwise sorted using any internal algorithm)

Distribution Stage :

RESET(T) , REWRITE(T1,T2)
T1 : 5-8-17-20/3-9-11-21/15-18-22-23/6-14-15-24/8-10-15-21/ Fib(5) = 5 RUNS
T2 : 6-13-16-18/6-11-24-25/5 Fib(4) = 3 RUNS

Merge Stage :

1.Round :

RESET(T1,T2); REWRITE(T3)
T1 : 6-14-15-24/8-10-15-21/

T2 : EMPTY

T3 : 5-6-8-13-16-17-18-20/3-6-9-11-11-21-24-25/5-15-18-22-23/

2.Round :

RESET(T1,T3), REWRITE(T2)

T1 : EMPTY
T2 :5-6-6-8-13-14-15-16-17-18-20-24/3-6-8-9-10-11-11-15-21-21-24-25/
T3 : 5-15-18-22-23/

<u>3.Round :</u>

RESET(T2,T3) , REWRITE(T1)

T1 :5-5-6-6-8-13-14-15-15-16-17-18-18-20-22-23-24/

T2 : 3-6-8-9-10-11-11-15-21-21-24-25/

T3 : EMPTY

<u>4.Round :</u>

RESET(T1,T2) , REWRITE(T3)

T1 : EMPTY

T2 : EMPTY

T3 :3-5-5-6-6-6-8-8-9-10-11-11-13-14-15-15-15-16-17-18-18-20-21-21-22-23-24-24-25

CH3 : Graph Algorithms (Shortest Path Algorithms)

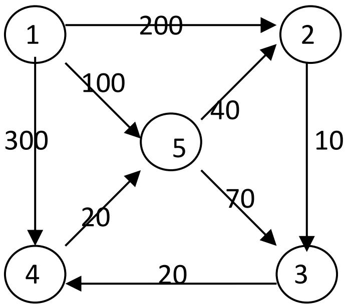
Shorted path algorithms:

Let G =(V,E), where V={ $v_0 \ldots v_n$} set of vertices
E set of edges
Suppose **a , b** $\in$ **V** a k-edges path between **a** and **b** defined as
P : v_0 = **a** , v_1 , v_2 , .. , v_k = **b** with (k+1) vertices
A cycle is a k-edge path such that $v_0 = v_k$, k >= 2

*** Type of graphs:*

1- Undirected graph , { v_i, v_{i+1} } $\in$ E   symmetric
2- Directed graph , (v_i, v_{i+1}) $\in$ E not symmetric
3- Undirected weighted graph
4- Directed weighted graph

Example :



Weight :
W : E → R (or any other informations)

W({1,4}) = 300 = W({4,1})
W((1,4)) = 300 $\neq$ W((4,1))= ∞
W((1,1)) = 0
W({1,1}) = 0

- w(p) = w({ v_0, v_1}) + w({ v_1, v_2}) + ……….+ w({ v_{k-1}, v_k})   (undirected)
- w(p) = w((v_0, v_1)) + w((v_1, v_2)) + ……….+ w((v_{k-1}, v_k))   (directed)

shorted path **p** from **a** to **b** is a path such that for all **ṕ** from **a** to **b : w(p) <= w(ṕ)**

Four shortest path problems :

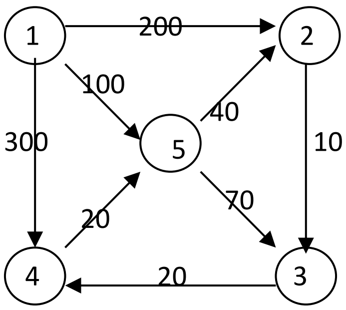
1- Single pair problem : Find a shortest path from one given vertex **a** to another vertex **b**
2- Single source problem : Given a source vertex **a** , find for every vertex **v** a shortest path from **a** to **v**
3- Single sink problem : Given a sink vertex **b** , find for every vertex **v** a shortest path from **v** to **b**
4- All pairs problem : For every orderd pair **(a,b)** of vertices find a shortest path from **a** to **b**

(1) Dijkstra's Algorithm : (For single source problem)

1- Building up a set S of vertices , initialized to source vertex
2- Adding new vertex to S until all vertices of the graph in S
3- Define an array **d** indexed by vertices ***(without the source vertex a***) contains the weights initialized with :
d[x] = { w((a,x)) , if (a,x) $\in$ E (Directed), // w({a,x}) , if {a,x} $\in$ E (Undirected)
{ ∞ other wise .
This array is defined to insert a new vertex to S.

4- *for i = 1 to n-1 do*
{
choose a vertex ***v*** *not in* ***S*** *, for which* ***d[v]*** *is least*
➔ ***S = S U {v} ;***
for each vertex ***x*** *not in* ***S*** *(inner loop)*
➔ ***d[x] = min { d[x],d[v]+w((v,x))} // if (d[v]+w((v,x))< d[x])***
➔ ***d[x] = d[v]+w((v,x))***
}

Example :



1- Initialized suppose a = 1 is the source vertex
➔S ={1}
2- d[2]= 200
d[3] = ∞
d[4] = 300
d[5] = **100** ***unchanged***
➔ d[5] = min { d[2] , d[3] , d[4] ,d[5] } = min {200 , ∞ , 300 , 100} ➔ v = 5
3- Join vertex 5 to S ➔ S = ***S:= S U {v}*** = { 1 , 5}

4- x = 2 , 3 , 4 all are not in S (inner loop)

d[x] = min {d[x],d[5]+w(5,x))}

→ d[2] = min {200,100+40} = **140** ***changed***
d[3] = min {∞ ,100+70}= **170** ***changed***
d[4] = min {300 , 100+∞ } = 300 ***unchanged***

5- Again with outer loop : for all v not in S :
→ d[2] = min { d[2] , d[3] , d[4] } = min { 140, 170, 300} → v = 2
→ S = S U {2} ➔ S = {1,5,2}

6- Again with step (4) : x = 3 , 4 all are not in S (inner loop)
   d[x] = min {d[x],d[2]+w(2,x))}
   → d[3] = min {170,140+10}=**150** ***changed***
   d[4] = min { 300,140+∞ } = 300 ***unchanged***

7- Again with outer loop : for all v not in S :
   → d[3] = min { d[3] , d[4] } = min { 150 , 300} → v = 3
   → S = S U {3} → S = {1 , 5 , 2 , 3}

8- Again with step (4) : x = 4 all are not in S (inner loop)
   d[x] = min {d[x],d[3]+w(3,x))}
   d[4] = min {300,150+20} =**170** ***changed***
   → S = S U {4}→ S ={1,5,2,3,4}

The weight of the shortest path from 1 to the vertex 2 is d[2]=140 ,
Path : 1→5→2
The weight of the shortest path from 1 to the vertex 3 is d[3]=150 ,
Path : 1→5→2→3
The weight of the shortest path from 1 to the vertex 4 is d[4]= 170 ,
Path : 1→5→2→3→4
The weight of the shortest path from 1 to the vertex 5 is d[5]=100 ,
Path : 1→5

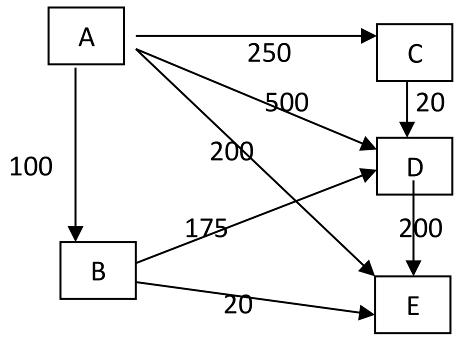
(2) Greedy's Algorithm : (For single source problem)

1- Building up a set S of vertices , initialized to source vertex
2- Adding new vertex to S until all vertices of the graph in S
3- Define one dim. array **d** indexed by vertices (**without the source vertex a**) contains the weights initialized with
   d[x] = { w((a,x)) , if (a,x) ∈ E , // w({a,x}) , if {a,x} ∈ E (Undirected)
   { ∞ other wise .
   This array is defined to insert a new vertex to S.
4- Introduce one dim. array **p** indexed by vertices other than the source, the entries of this array are vertices
5- Initialize **p** with source vertex

6- Body of algorithm

```
For i = 1 to n-1 do
 {
  Choose a vertex v not S , for which d[v] is least;
   S = S U {v} ;
   for each vertex x not in S do    // d[x] = min {d [x] , d[v]+w((v,x))}
     if (d[v] + w((v,x)) < d[x] )
      {
        d[x] = d[v] + w((v,x));
        P[x] = v;
      }
 }
```

Example :



Start vertex : **A**

1- Init S ={A};

d[B] = **100** **unchanged**

d[C] = 250

d[D] =500

d[E] =200

2- Init **p** with source **A**

p[B] = A

p[C] = A

p[D] = A

p[E] = A

Body of algorithm :

Outer loop : for all verices v not in S :

d[B] = min { d[B] , d[C] , d[D] , d[E] } = min {100 , 250 , 500 , 200 }= 100

→ v = B

→ S = S U {B} = {A,B}

Inner loop : for all vertices x not in S : x = C, D, E

d[x] = min { d[x], d[B]+ w((B,x))}

x = C : d[C] = min {250,100+ ∞ } = 250 **unchanged**

x = D : d[D] = min {500,100+175 } = 275 **changed**

x = E : d[E] = min { 200,100+20 } = 120 **changed**

⇒

p[C] := A **unchanged**

p[D] := B **changed**

p[E] := B **changed**

Again Outer loop : for all vertices v not in S :

d[E] = min { d[C] , d[D] , d[E] } = min { 250 , 275 , 120 }= 120 → v = E

→ S = S U {E} ={A,B,E }

Again inner loop : for all vertices x not in S : x = C, D

d[x] = min { d[x], d[E]+ w((E,x))}

x = C : d[C] = min {250,120+ ∞ } = 250 **unchanged**

x = D : d[D] = min {275,120+ ∞} = 275 **unchanged**

⇒

p[C] = A **unchanged**

p[D] = B **unchanged**

Again Outer loop : for all vertices v not in S :

d[C] = min { d[C] , d[D] } = min { 250 , 275 }= 250 → v = C

→ S =S U {C} = { A, B, E, C }

Again inner loop : for all vertices x not in S : x = D

d[x] = min { d[x], d[C]+ w((C,x))}

x = D : d[D] = min {275, 250+20 } = 270 **changed**

⇒

p[D] := C **changed**

Again Outer loop : for all vertices v not in S :

d[D] = min { d[D] } = min { 270 }= 270 → v = D

→ S =S U {D} = { A, B, E, C , D}

Stop

⇒ Output : S ={ A , B , E , C , D }

d[B] = 100	p[B] = A
d[C] = 250	p[C] = A
d[D] = 270	p[D] = C
d[E] = 120	p[E] = B

The weight of the shortest path from A to the vertex B is d[B]=100 ,
Path : A→B

The weight of the shortest path from A to the vertex C is d[C]=250 ,
Path : A→C

The weight of the shortest path from A to the vertex D is d[D]=270 ,
Path : A→C→D

The weight of the shortest path from A to the vertex E is d[E]=120 ,
Path : A→B→E

Single source problem :

(1) Dijkstra's Algorithm
(2) Greedy’s Algorithm

<u>All pairs problem :</u>

<u>(1) Folyd's Algorithm :</u>

G(V,E) , V ={1,…,n}

1- Construct Adjacent matrix initialized with :

$$D[i,j] = \begin{cases} w((i,j)), & \text{if the edge } (i,j) \in E \\ \infty, & \text{other wise} \end{cases}$$

2- Construct a sequence of matrices D_0 , D_1 ,.. , D_n
For k = 1 , 2 ,…, n construct D_k as follows
$D_k[i , j] = \min \{ D_{k-1}[i , j] , D_{k-1}[i , k] + D_{k-1} [k , j]\}= D_{k-1}[i , k]$

3- *float D[nxn] ;*

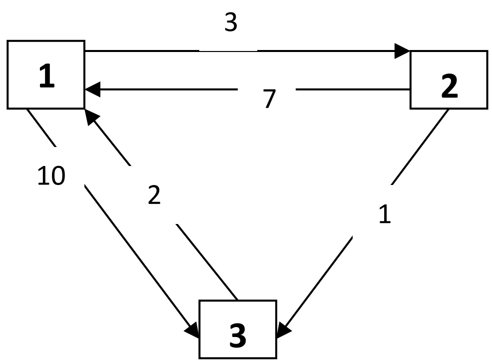
<u>Body of Algorthim :</u>

```
For k = 1 to n do
  For i = 1 to n do
    For j = 1 to n do
      D [i , j] = min { D[i , j],D[i , k]+D[k , j]};
```

<u>OR</u>

```
For k = 1 to n do
   For i = 1 to n do
     For j = 1 to n do
         if  (D[i , k]+D[k , j]  <  D [i , j])
            then   D[i , j] = D[i , k]+D[k , j];
```

<u>Example :</u>



Initialization :

$D_0 = D$

j

i	1	2	3
1	0	3	10
2	7	0	1
3	2	∞	0

Body of Algorithm :

Outer loop : k =1

* keep the 1. Row , 1. Column and the diagonal of D_0 to D_1 ***unchanged***

* Use statement : D[i , j] = min { D[i , j] , D[i , k] + D[k , j] } to calculate

1- D[2 , 3] = min { D[2 , 3] , D[2 , 1] + D[1 , 3] }
= min { 1 , 7 +10 } = 1

2- D[3 , 2] = min { D[3 , 2] , D[3 , 1] + D[1 , 2] }
= min { ∞ , 2 + 3 } = 5

$\Rightarrow$ **D_1 =**

	1	2	3
1	0	3	10
2	7	0	**<u>1</u>**
3	2	**<u>5</u>**	0

Outer loop : k =2

* keep the 2. Row , 2. Column and the diagonal of D_1 to D_2 ***unchanged***

* Use statement : D[i , j] = min { D[i , j] , D[i , k] + D[k , j] } to calculate

1- D[1 , 3] = min { D[1 , 3] , D[1 , 2] + D[2 , 3] }
= min { 10 , 3 + 1 } = 4

2- D[3 , 1] = min { D[3 , 1] , D[3 , 2] + D[2 , 1] }
= min { 2 , 5 + 7 } = 2

$\Rightarrow$ **D_2 =**

	1	2	3
1	0	3	**<u>4</u>**
2	7	0	1
3	**<u>2</u>**	5	0

Outer loop : k = 3

* keep the 3. Row , 3. Column and the diagonal of D_2 to D_3 ***unchanged***

* Use statement : D[i , j] = min { D[i , j] , D[i , k] + D[k , j] } to calculate

1- D[1 , 2] = min { D[1 , 2] , D[1 , 3] + D[3 , 2] }
= min { 3 , 4 + 5 } = 3

2- D[2 , 1] = min { D[2 , 1] , D[2 , 3] + D[3 , 1] }
= min { 7 , 1 + 2 } = 3

$\Rightarrow$ **D_3 =**

	1	2	3
1	0	**<u>3</u>**	4
2	**<u>3</u>**	0	1
3	2	5	0

⇒

All pairs of vertices	The **weights** of the shortest path using the matrix D	The shortest path using the matrix P
1,1	0	?
1,2	3	?
1,3	4	?
2,1	3	?
2,2	0	?
2,3	1	?
3,1	2	?
3,2	5	?
3,3	0	?

(2) Modify Floyd's algorithm: (ALL PAIRS PROBLEM)

Idea :

Modify Floyd's algorithm : producing a matrix **P** indexed (1..n,1..n) and the entries of this matrix are 0 or vertices

Given a graph G(V,E) , V ={1,…,n} :

1- Construct Adjacent matrix initialized with :

D[i,j] = { w((i , j)) , if the edge (i , j) $\in$ E
{ ∞ , otherwise

2- construct a matrix P_{nxn} : Initialize **P[i , j]** = 0 , i = 1,……..,n and j = 1,………..,n

3- Construct a sequence of matrices D_0 = D , D_1 ,.. , D_n
For k = 1 , 2 ,…, n construct D_k as follows
$D_k[i , j] = \min \{ D_{k-1}[i , j] , D_{k-1}[i , k] + D_{k-1} [k , j]\}= D_{k-1}[i , k]$

4- Body of Algorithm :

For k = 1 to n do
For i = 1 to n do
For j = 1 to n do
If (D[i , k] + D[k , j] < D[i , j])
{
D[i , j] = d[i , k]+ d[k , j] ;
P[i , j] = k
}

5- P[i , j] = 0 → there is a shortest path (i , j) direct between i and j
Other wise → using a **procedure path(i, j)** to define all intermediate vertices between i
and j

```
path (i , j )
{
 x ∈ 0..n;

 x = p[i , j] ;
 if (x <> 0)
   {
      Path (i , x);
      S.O.P (x);
      Path (x , j);
   }
 }
```

Example :

Suppose we have the following Adjacent matrix for a directed graph :

D_0 = D

0	90	100	70
40	0	5	10
7	∞	0	4
20	10	7	0

P =

0	0	0	0
0	0	0	0
0	0	0	0
0	0	0	0

k = 1 : find D[i,j] = min { D[i,j] , D[i,k] + D[k,j] } , if changed P[i,j] = k

D_1 =

0	90	100	70
40	0	**5**	**10**
7	**97**	0	**4**
20	**10**	**7**	0

P =

0	0	0	0
0	0	**0**	**0**
0	**1**	0	**0**
0	**0**	**0**	0

k = 2 : find D[i,j] = min { D[i,j] , D[i,k] + D[k,j] } , if changed P[i,j] = k

D_2 =

0	90	**95**	**70**
40	0	5	10
7	97	0	**4**
20	10	**7**	0

P =

0	0	**2**	**0**
0	0	0	0
0	1	0	**0**
0	0	**0**	0

k = 3 : find D[i,j] = min { D[i,j] , D[i,k] + D[k,j] } , if changed P[i,j] = k

D_3 =

0	**90**	95	**70**
12	0	5	**9**
7	97	0	4
14	**10**	7	0

P =

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

k = 4 : find D[i,j] = min { D[i,j] , D[i,k] + D[k,j] } , if changed P[i,j] = k

D_4 =

0	**80**	**77**	70
12	0	**5**	9
7	**14**	0	4
14	10	7	0

P =

0	**4**	**4**	0
3	0	**0**	3
0	**4**	0	0
3	0	0	0

$\Rightarrow$

All pairs of vertices	The **weights** of the shortest path using the matrix D	The shortest path using the matrix P
1,1	0	1-1
1,2	80	1-4-2
1,3	77	1-4-3
1,4	70	1-4
2,1	12	2-3-1
2,2	0	2-2
2,3	5	2-3
2,4	9	2-3-4
3,1	7	3-1
3,2	14	3-4-2
3,3	0	3-3
3,4	4	3-4
4,1	14	4-3-1
4,2	10	4-2
4,3	7	4-3
4,4	0	4-4

By example : The shortest path between 1 and 2 is 1 , 4 , 2 with weight = 80
………….
…………

CH4 : Spanning Tree Algorithms

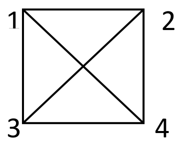
$G = (V,E)$, $V=\{ v_1 \dots v_n\}$
P : $v_1 , v_2 , \dots\dots\dots , v_k$ a path from v_1 to v_k

Defintions :

1- **Simple path** : if all intermediate vertices between $\boldsymbol{v_1}$ and $\boldsymbol{v_k}$ are distinct.
2- Cycle Path : if source $v_1 = v_k$ sink and there are at least 3 different vertices.
3- Simple Cycle : if the path is simple + cycle .

4- **Connected graph :** for all v , w $\in$ V there is at least a simple path from v to w.
5- **Tree graph** **:** is a connected graph with
   a- If for all v, w distinct vertices
      → there is a unique simple path from v and w
   b- A graph with n vertices has n-1 edges
6- Spanning tree :
   G = (V, E) connected graph
   A spanning tree defined as $G^{\sim} = (V , E^{\sim})$ where $E^{\sim}$ subset of E such that $E^{\sim}$ has enough edges to form a tree .
7- **Min Spanning tree :** a spanning tree with least weight of edges.

Example : connected graph



Simple path : 1 , 2 , 3 , 4 **<u>or</u>**
1 , 3 , 4 , 2

Not simple path : 1 , 2 , 3 ,4 , 3 , 2 , 4

Cycle : 1 , 2 , 4 , 1 **<u>or</u>**
4 , 3 , 1 , 2 , 4 **<u>or</u>**
1 , 4 , 2 , 3 , 4 , 1

Simple cycle : 1 , 2 , 4 , 1 **<u>or</u>**
4 , 3 , 1 , 2 , 4

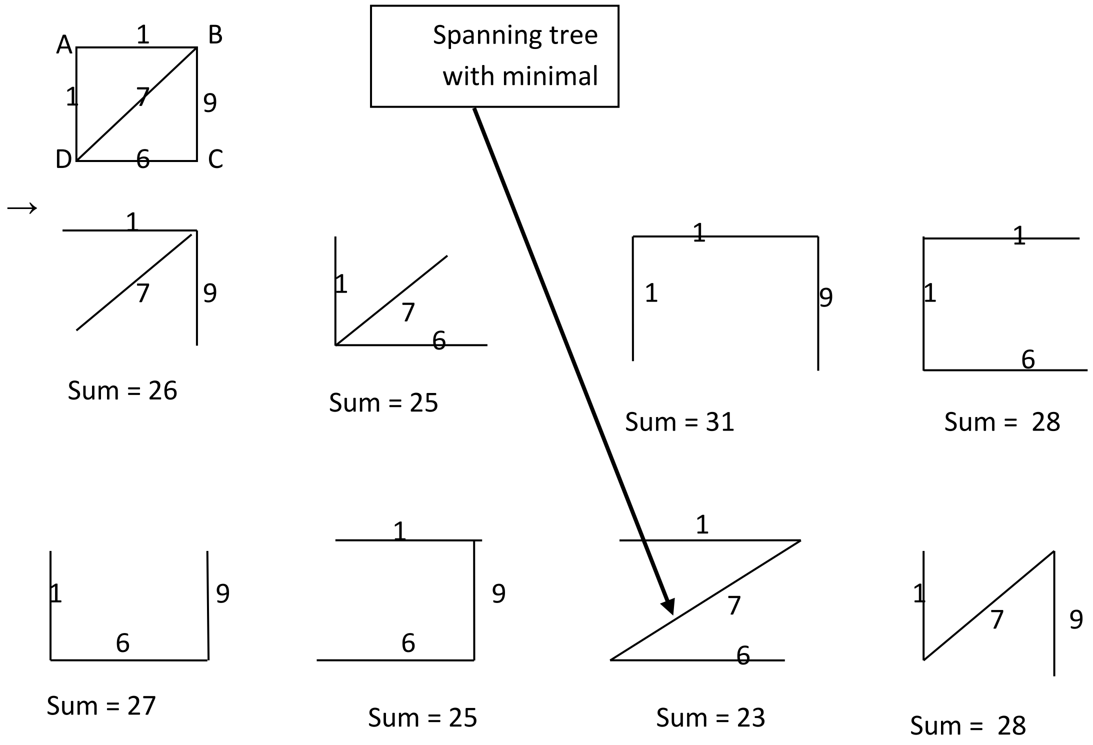
Not Simple Cycle : 1 , 4 , 2 , 3 , 4 , 1 **or**
1 , 2 , 1

Making min spanning tree :

Idea :

Given a connected graph by a cycle remove one edge such that the graph still have a connected graph repeating this procedure until no cycle exists.

Example :



Algorithms to construct a min spanning tree :

IDEA :

Colouring the edges :

2 sets of E one contains the **blue** edges the other contains **red** edges

→ Min spanning tree of G which includes all the blue edges none of the red edges.

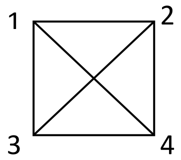
Construct two Procedures :

1- Blue rule procedure
2- Red rule procedure

Definition :   X subset of V , e={ v,w} Є E

e= { v,w} ***protruded*** from X if one end of ***e*** is in X the other is not

Example : (protruded edges)



E = { {1,2} , {1,3} , {1,4} , {2,3} , {2,4} , {3,4} }

Edges e	**e prodruded from X = {4}**	**e prodruded from X = {1,4}**	**e prodruded from X = {1,3,4}**
{1,2}	**NO**	**YES**	**YES**
{1,3}	**NO**	**YES**	**NO**
{1,4}	**YES**	**NO**	**NO**
{2,3}	**NO**	**NO**	**YES**
{2,4}	**YES**	**YES**	**YES**
{3,4}	**YES**	**YES**	**NO**

Blue rule :

1- Choose a non empty subset X of V .
2- Among the uncoloured edges protruded from X choose one of minimum weight and colour it blue

Red rule :

1- Choose a simple cycle K which includes no red edges
2- Among the uncoloured edges of K choose one of maximum weight and colour it red

Stop until n-1 edges coloured blue

→ ***3 algorithms for constructing minimum spanning tree :***

- ***Boruvka's algorithm***
- ***Kruskal's algorithm***
- ***Prim's algorithm***

1- Boruvka's Algorithm :

G = (V, E) connected weighted graph all edges distinct

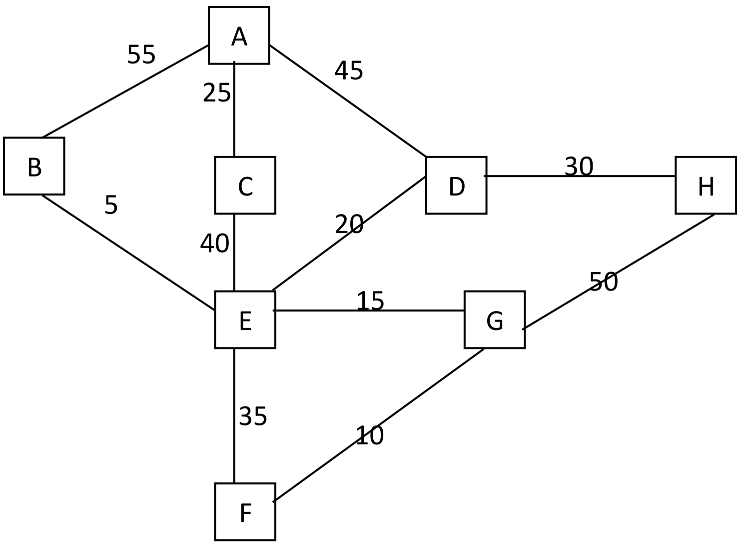
Idea :

- Construct a collection F of blue trees initialized with n single vertex trees.
- Repeating until F consist a single blue tree (with n - 1 edges).

Body of algorithm :

- Choose a set F1 of F containing enough trees of F with different min weighted edges (protruded from these trees)
- Colouring these edges blue → F is a forest
- Repeat until F contain single blue tree with n – 1 edges

Example :



Initialize :

F = { ({A},∅) ,({B},∅) ,({C},∅) ,({D},∅) ,({E},∅) ,({F},∅) ,({G},∅) , ({H}, ∅) }

X = {A}

⟶ AC(25),AB(55),AD(45) prodruded form X

⟶ Min edge from {{A},∅ } is AC(25)

X = {B} .. ……………………………… …… Min edge from {{B}, ∅ } is BE(5)
X = {C}……………………………………. Min edge from {{C}, ∅ } is CA(25)
X = {D}…………………………………..…Min edge from {{D}, ∅ } is DE(20)
X = {E}………………………………...….. Min edge from {{E}, ∅ } is EB(5)
X = {F}……….. …………………………... Min edge from {{F}, ∅ } is FG(10)
X = {G}………………………………….... Min edge from {{G}, ∅ } is GF(10)
X = {H}………………………………….... .Min edge from {{H}, ∅ } is HD(30)

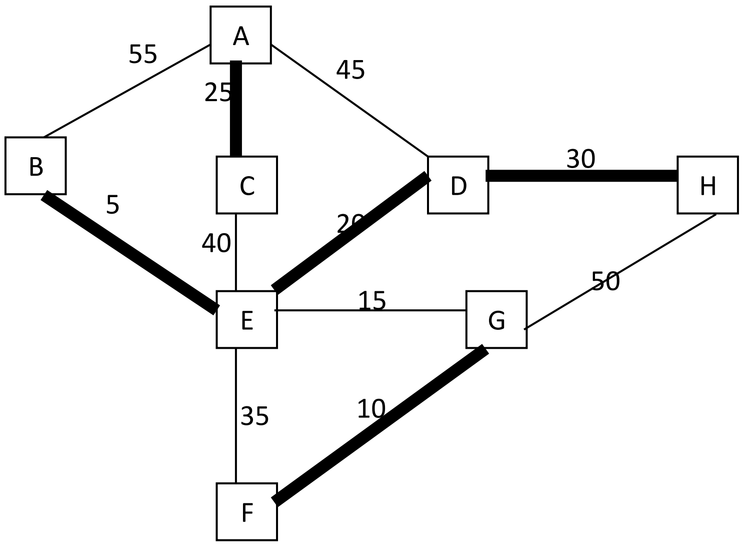
⇒ Eliminate doublicates :

Min edge from {{A}, ∅ } is AC(25)
Min edge from {{B}, ∅ } is BE(5)
Min edge from {{D}, ∅ } is DE(20)
Min edge from {{F}, ∅ } is FG(10)
Min edge from {{H}, ∅ } is HD(30)

We choose F1 subset of F as follows:

F1 = { ({A}, ∅) , ({B}, ∅) , ({D}, ∅) , ({F}, ∅) , ({H}, ∅) }

➔ Colouring AC , BE , DE , FG , HD blue ⇒



F1 defined now as follows

F1 = { T1 , T2 , T3 } , where
T1 = ({ A , C},AC)
T2 = ({B , E , D , H}, BE , ED , DH)
T3 = ({F , G}, FG)

Again

X = {A,C} all vertices in T1
⇒ CE(40) AB(55), AD(45) protruded from X
⇒ Min edge from T1 is CE(40)

X = {B , E , D , H}………………… …….Min edge from T2 is EG(15)
X = {F,G} ………………………………...Min edge from T3 is GE(15)

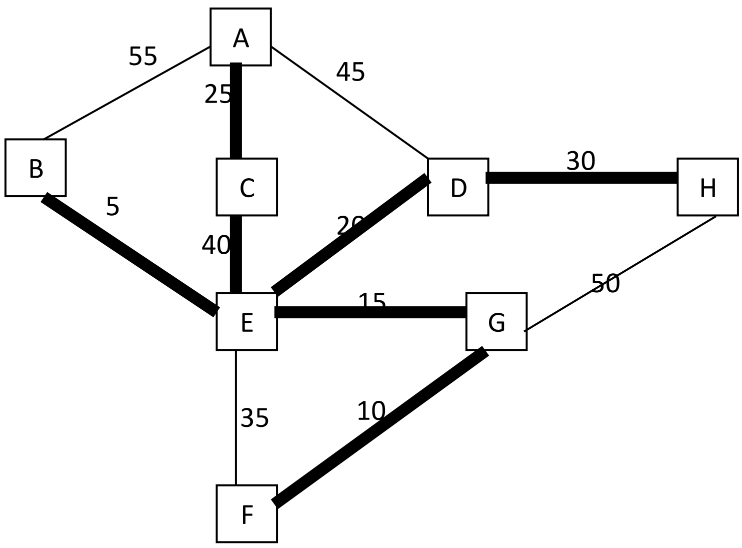
⇒ Eliminate doublicates :

Min edge from T1 is CE(40)
Min edge from T2 is EG(15)

We choose F1 subset of F as follows:

F1 = { T1 , T2}

➔ Colouring CE , EG blue ⇒



⇒ F = { (A , B , C , D , E , F , G , H) , AC , BE , CE , ED , EG , FG , DH }

Number of blue edges is equal to n – 1 = 7

STOP

2- Kruskal's algorithm :

Given G = (V,E) , V ={v_1, … , v_n} , E = { e_1, … ,e_m }

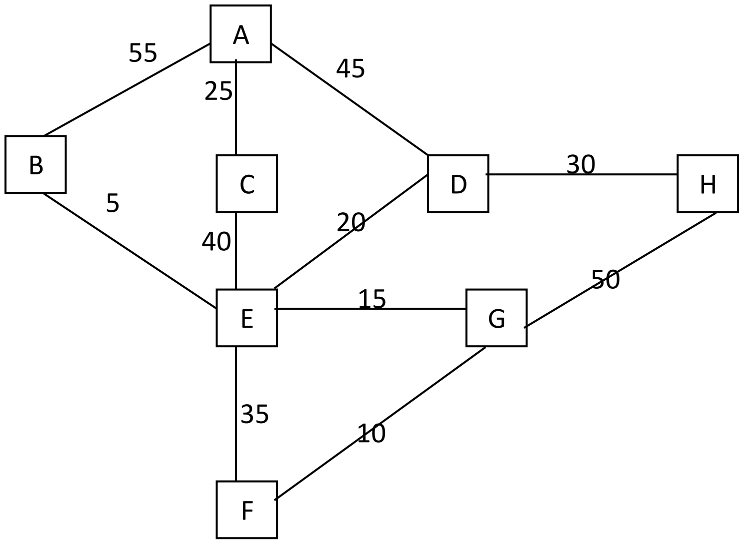
Initialization :

- F collection of blue tree initialized with (***n*** single vertex trees)
- Ordering the edges in increasing order of weights : w(e_1) <= … <= w(e_m)

Body of algorithm :

i = 1 ;
REPEAT
  IF (both ends of e_i are in the same blue tree)
    THEN colouring e_i ***RED***
  ELSE
    colouring e_i ***BLUE***
  i = i + 1 ;
UNTIL (there are n-1 ***BLUE*** edges)

Example :



INITIALIZE :

- F = {({A},∅), ({B},∅), ({C},∅), ({D},∅), ({E},∅), ({F},∅), ({G},∅), ({H},∅) }

- Ordering the edges in increasing order :
  BE(5),FG(10),EG(15),ED(20), AC(25), DH(30), EF(35), CE(40), AD(45), GH(50) , AB(55)

Body of algorithm :

<u>**i = 1;**</u>

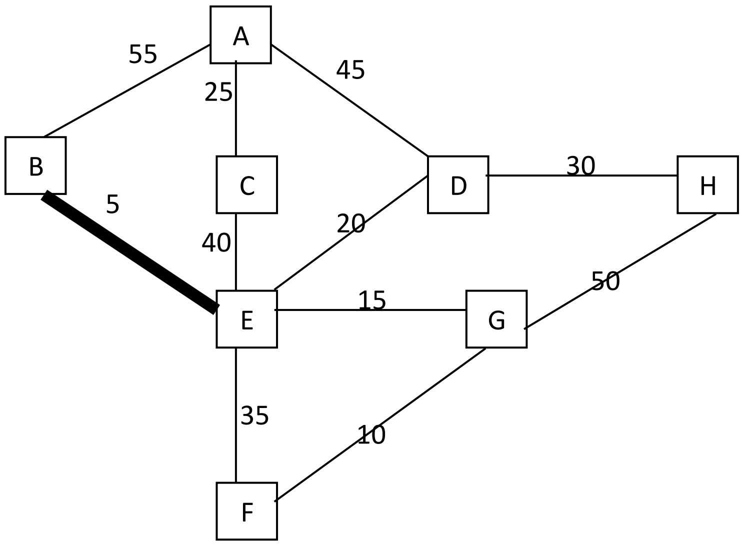
e_1 = BE(5) ;

One end in the blue tree ({B} , ∅) the other in the blue ({E}, ∅) **{different blue trees}** ⇒

Colouring BE(5) ***BLUE***

⇒

F = { ({A},∅) , ({B ,E}, BE) , ({C},∅) , ({D},∅) , ({F},∅) , ({G},∅) ,({H}, ∅) }



<u>i = 2:</u>

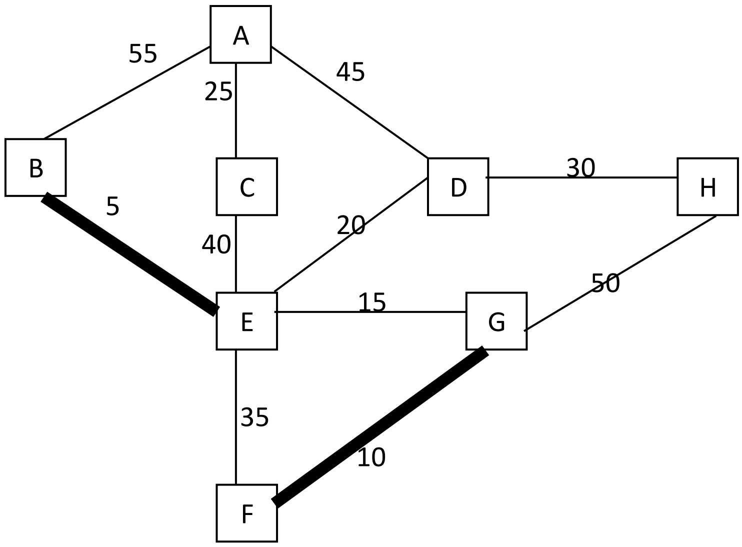
e_2 = FG(10) ;

One end in the blue tree ({F} , ∅) the other in the blue ({G}, ∅)**{different blue trees} ⇒**

Colouring FG(10) ***BLUE***

⇒

F = { ({A}, ∅) , ({B , E }, BE) , ({C}, ∅), ({D},∅) , ({F, G}, FG) , ({H}, ∅) }



<u>i = 3:</u>

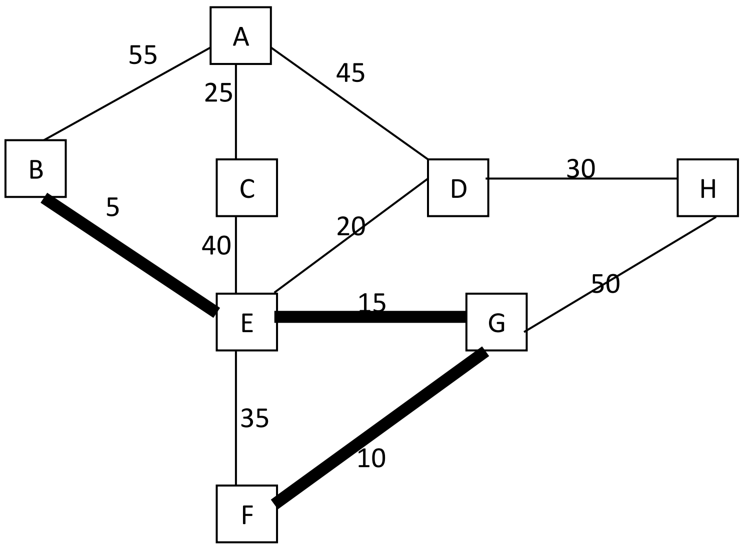
e_3 = EG(15) ;

One end in the blue tree ({B ,E},BE) the other in the blue ({F , G}, FG) **{different blue trees} ⇒**

Colouring EG(15) ***BLUE***

⇒

F = { ({A},∅),({B ,E ,F,G}, BE,EG,FG) ,({C},∅),({D},∅) ,({H},∅)}



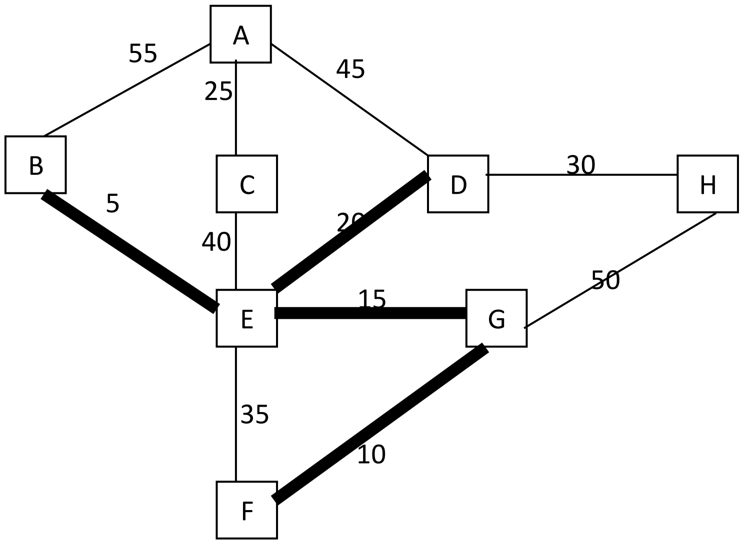
<u>i = 4:</u>

e_4 = DE(20) ;

One end in the blue tree ({B,E, F,G},BE , EG , FG) the other in the blue ({D}, ∅)

{different blue trees} ⇒ Colouring DE(20) ***BLUE***

⇒ F = { ({A}, ∅) , ({B , D , E , F , G }, BE , EG , FG , DE) , ({C}, ∅), ({H}, ∅) }



<u>i = 5:</u>

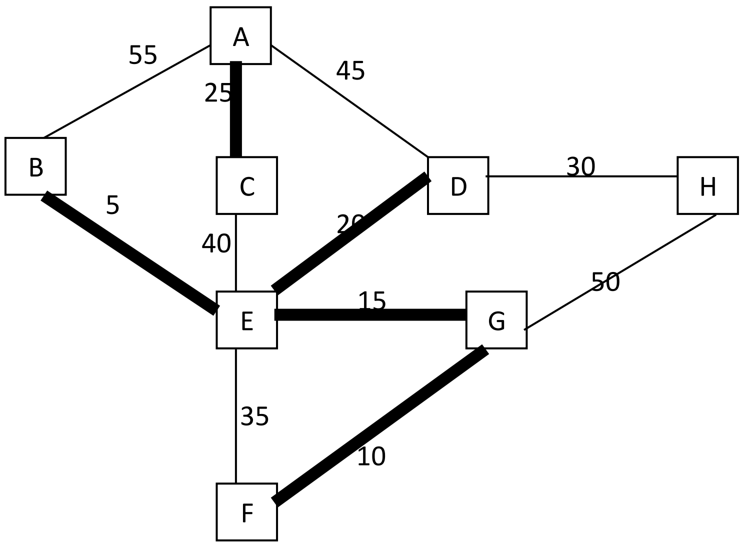
e_5 = AC(25) ;

One end in the blue tree ({A} , ∅) the other in the blue ({C}, ∅)**{different blue trees}** ⇒

Colouring AC(25) ***BLUE***

⇒

F = { ({A , C}, AC) , ({B , D , E , F , G }, BE , EG , FG , DE) , ({H}, ∅) }



<u>i = 6:</u>

e_6 = DH(30) ;

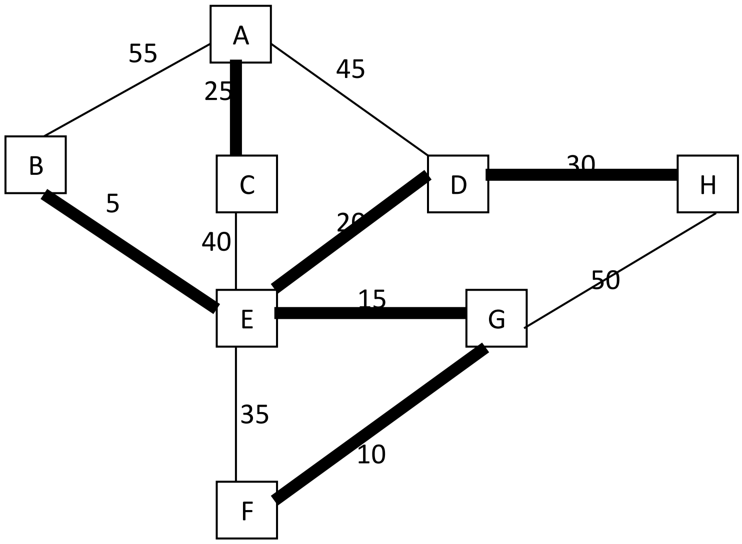
One end in the blue tree ({B ,D,E ,F,G }, BE , EG , FG , DE) the other in the blue ({H}, ∅) **{different blue trees}**

⇒ Colouring DH(30) ***BLUE***

⇒ F = {

({A , C}, AC) ,({B , D , E , F , G ,H }, BE , EG , FG , DE , DH)

}



<u>i = 7:</u>

e_7 = EF(35) ;

Both ends in the blue tree ({B , D , E , F , G , H }, BE , EG , FG , DE , DH)

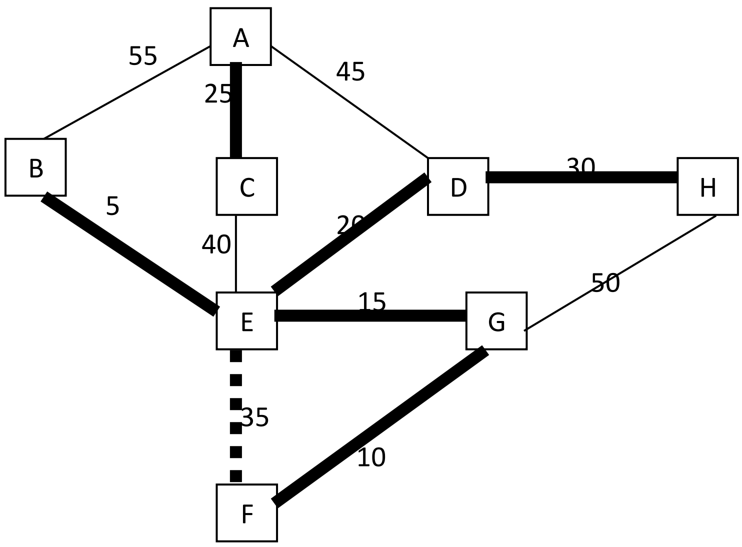
{same blue trees} ⇒

Colouring EF(35) ***<u>RED</u>***

⇒ F = {

({A , C}, AC) ,({B , D , E , F , G ,H }, BE , EG , FG , DE , DH)

}



<u>i = 8:</u>

e_8 = CE(40) ;

One end in the blue tree ({B , D , E , F , G , H }, BE , EG , FG , DE , DH) the other in the blue tree ({A , C}, AC) **{different blue trees}**

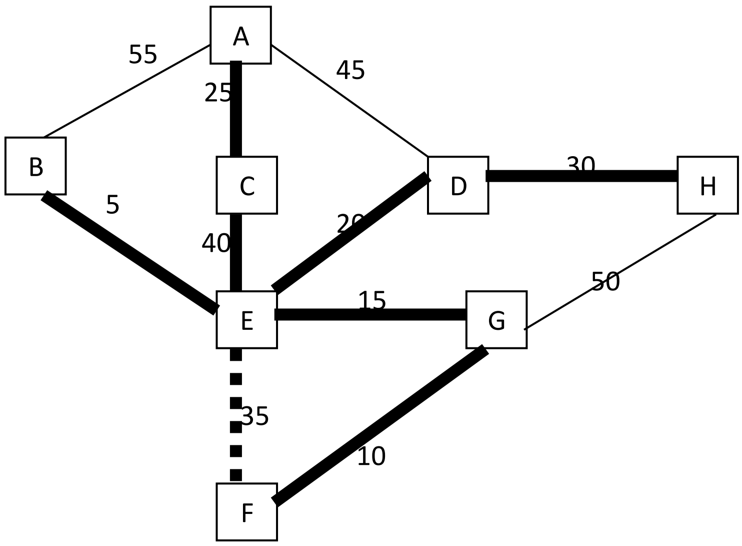
$\Rightarrow$

Colouring CE(40) ***BLUE***

$\Rightarrow$

F = { ({A , B , C , D , E , F , G , H }, AC , BE , CE , EG , FG , DE , DH) }

STOP!! (we have n – 1 = 7 blue edges)



3- PRIM's algorithm :

- Initialization :

T = { One vertex blue tree }

- Body of algorithm :

For i =1 to n-1 do

Begin

*1- Apply the **BLUE** rule to the set of edges protruded from T; // **min edge blue colouring***

→ T := T U { the new blue edge };

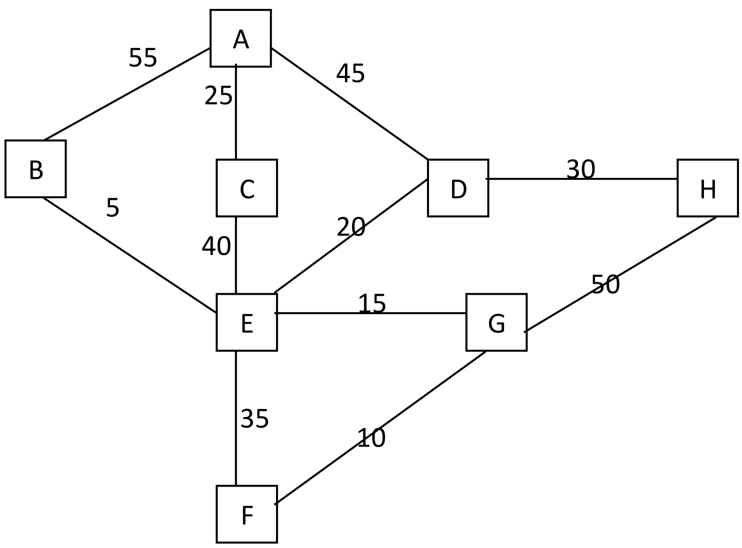
*2- Suppose **e , e`** two edges are protruded from T with **v** as common endpoint not in T*

*and **e , e`** form a cycle **K** , a simple cycle without **red edges***

*→ Apply **RED** rule to **max { e , e` };***

END;

Example :



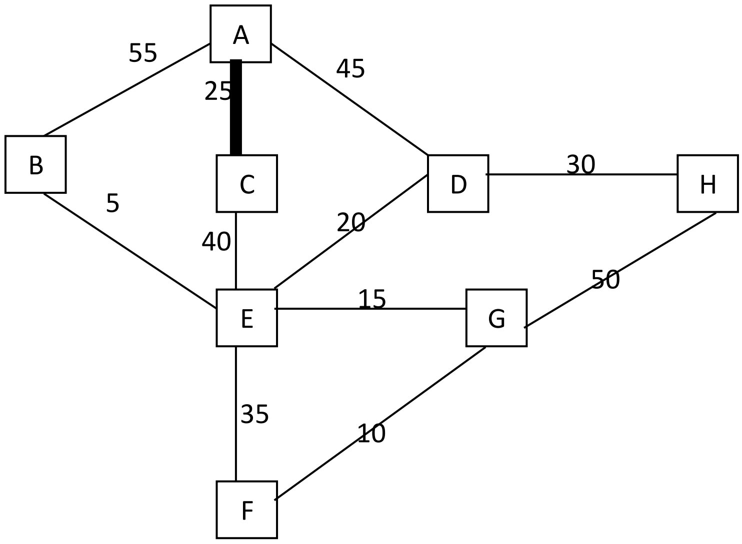
Suppose T = {(A) , ∅}

1- AB(55) , AC(25) , AD(45) **protruded** from T

⇒ AC = min { AB(55) , AC(25) , AD(45) }

⇒ Colouring AC(25) ***BLUE***

⇒ T = {(A,C) , AC}

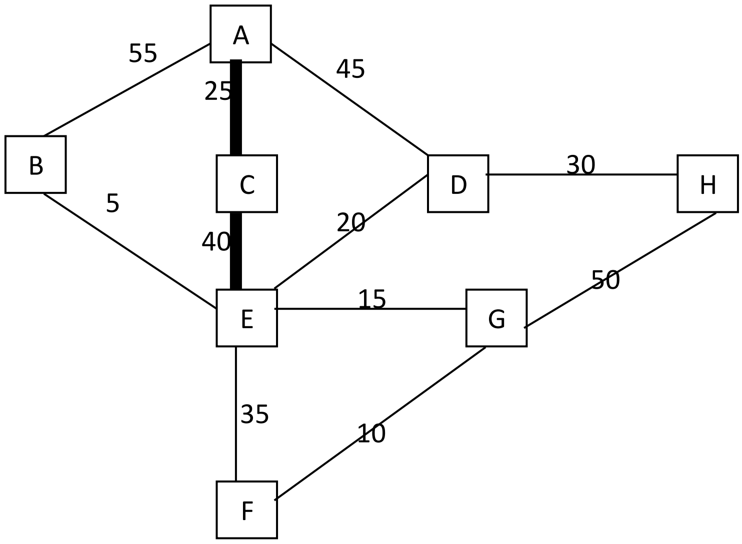


2- AB(55) , CE(40) , AD(45) **<u>protruded</u>** from T

⇒ CE = min { AB(55) , CE(40) , AD(45) }

⇒ Colouring CE(40) ***BLUE***

⇒ T = {(A , C , E) , AC , CE}



3- AB(55) , AD(45) , EB(5) , ED(20) , EG(15) ,EF(35) **<u>protruded</u>** from T

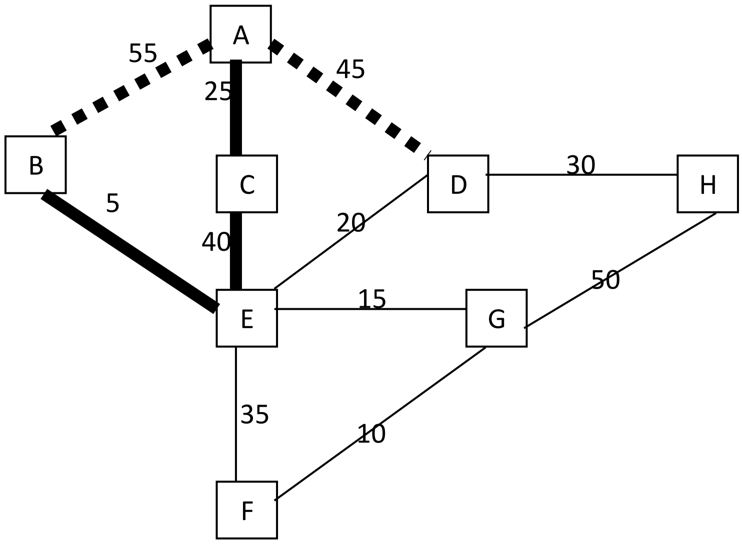
⇒ EB(5) = min { AB(55) , AD(45) , EB(5) , ED(20) , EG(15) ,EF(35) }

And {A , B , E , C , A} , {A , C , E , D , A} are ***cycles*** without any ***RED*** edges

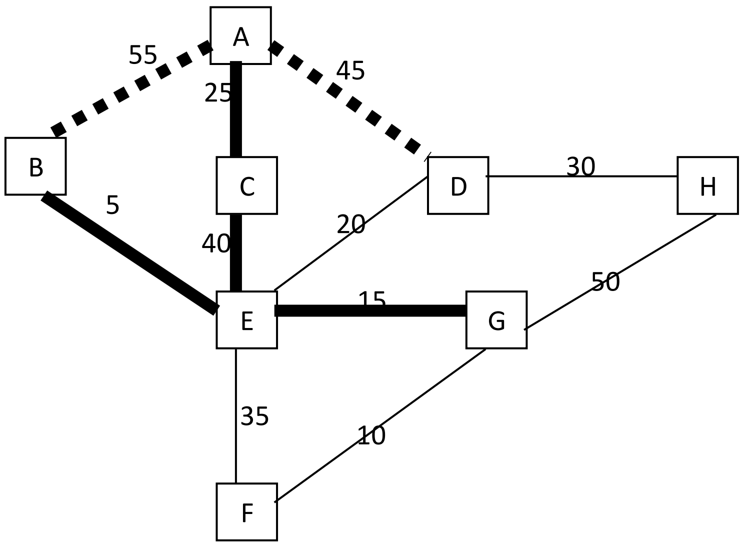
⇒ Colouring EB(5) ***BLUE*** and

⇒ Colouring AB and AD ***RED*** , where AB = max{AB , BE} and AD = max{AD , DE}

⇒ T = {(A , B , C , E) , EB , AC , CE}

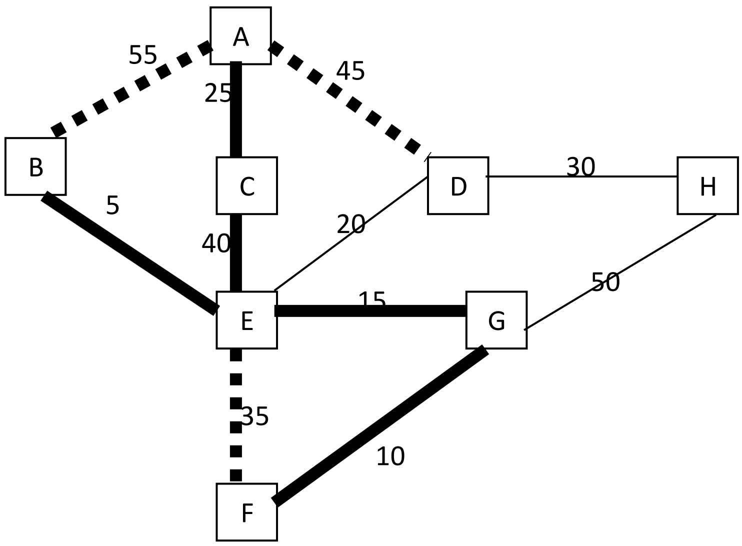


4- AD(45) , ED(20) , EG(15) , EF(35) **protruded** from T
$\Rightarrow$ EG(15) = min { AD(45) , ED(20) , EG(15) , EF(35) }
$\Rightarrow$ Colouring EG(15) ***BLUE***
$\Rightarrow$ T = {(A , B , C , E , G) , EB , AC , CE ,EG}

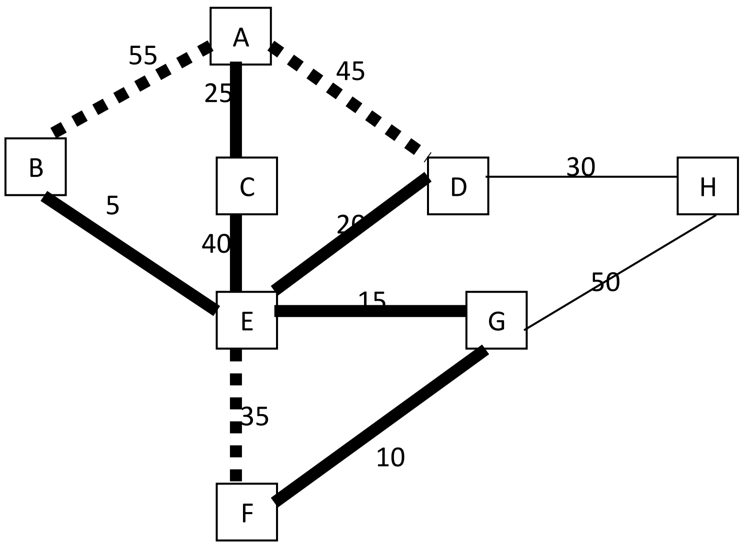


5- AD(45) , ED(20) , FG(10) , GH(50) , EF(35) **protruded** from T
$\Rightarrow$ FG(10) = min { AD(45) , ED(20) , FG(10) , GH(50) , EF(35) }
And { E , F , G , E } is a ***cycle*** without any ***RED*** edges
$\Rightarrow$ Colouring FG(10) ***BLUE*** and

⇒ Colouring EF(35) ***RED*** , where EF(35) = max{EF(35) , FG(10)}
⇒ T = {(A , B , C , E , F , G) , EB , AC , CE , FG , EG}

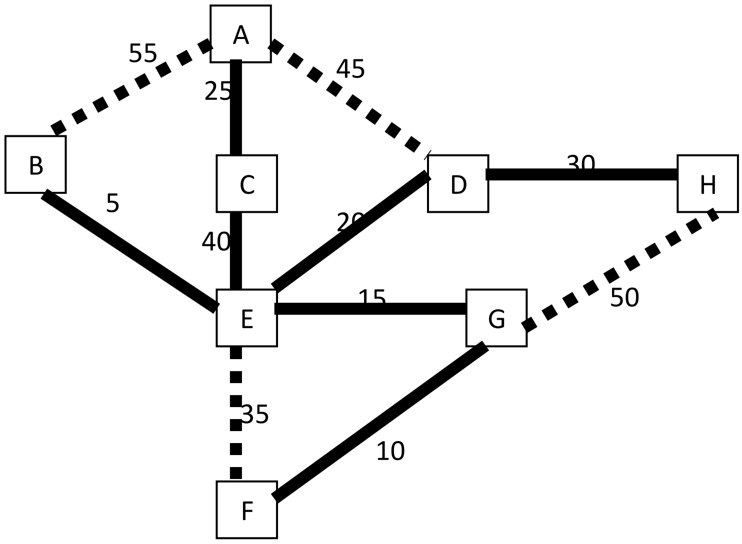


6- AD(45) , ED(20) , GH(50) **<u>protruded</u>** from T
⇒ ED(20) = min { AD(45) , ED(20) , GH(50) }
⇒ Colouring ED(20) ***BLUE***
⇒ T = {(A , B , C , D , E , F , G) , EB , AC , ED , CE , FG , EG}



7- DH(30) , GH(50) **<u>protruded</u>** from T
⇒ DH(30) = min { DH(30) , GH(50) }
And {D , E , G , H , D} is a ***cycle*** without any ***RED*** edges
⇒ Colouring DH(30) ***BLUE*** and
⇒ Colouring GH(50) ***RED*** , where GH(50) = max{DH(30) , GH(50)}

$\Rightarrow$ T = {(A , B , C , D , E , F , G , H) , EB , AC , DH , ED , CE , FG , EG}



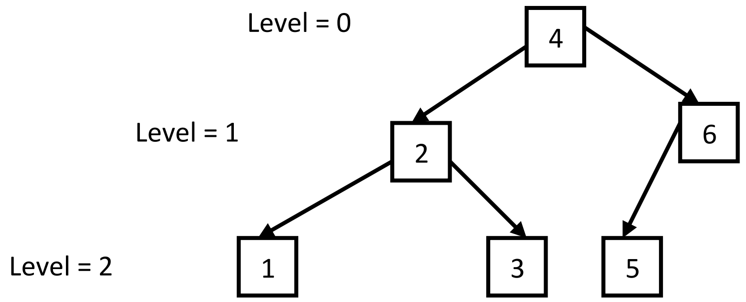
STOP!!!

CH5 : STORING IN BINARY SEARCH TREE

Review :

- Level of binary search tree : n = -1 if empty , n = 0 if only root, else ……
- Height of binary search tree : h = the no of the last level of the binary tree
- No of Node in a binary search tree with height equal to h : $<= 2^{h+1} – 1$
- No of leaves in a binary search tree with height equal to h : $<= 2^{h}$

Example :



- Height : h = 2
- No of Nodes $<= 2^{2+1} – 1 = 7$
- No of leaves $<= 2^{2} = 4$

Data structure :

```
public class TreeNode
{
   protected int info;   //key
   protected TreeNode left;
   protected TreeNode right;

  public TreeNode ()
   {      }

  public TreeNode(int info,TreeNode left,TreeNode right)
   {
      this.info = info;
      this.left = left;
      this.right = right;
   }
 }
```

Search: ***(RECURSION)***

```
void treeSearch (int x ,TreeNode T  )
 {
    if ( T == null)  ➔ stop not found;
    else
       if ( T.info  ==  x)  ➔ stop found;
          else
            if  (T.info > x ) ➔ treeSearch (x ,T.left);
                 else
                   treeSearch (x , T.right);
  }
```

(NON-RECURSION)

```
  begin
    repeat
     if  (T == null)
           ➔ stop;  done = true;
   else
      if  (T.info == x)
        ➔ stop; done = true;
      else
         done = false;
     if ( x < T.info)  ➔    T = T.left
     else
          T = T.right;
   until done
 end;
```

Complexity of binary search tree :

The max number of key comparisons is one more than the Height of the tree.

Example :

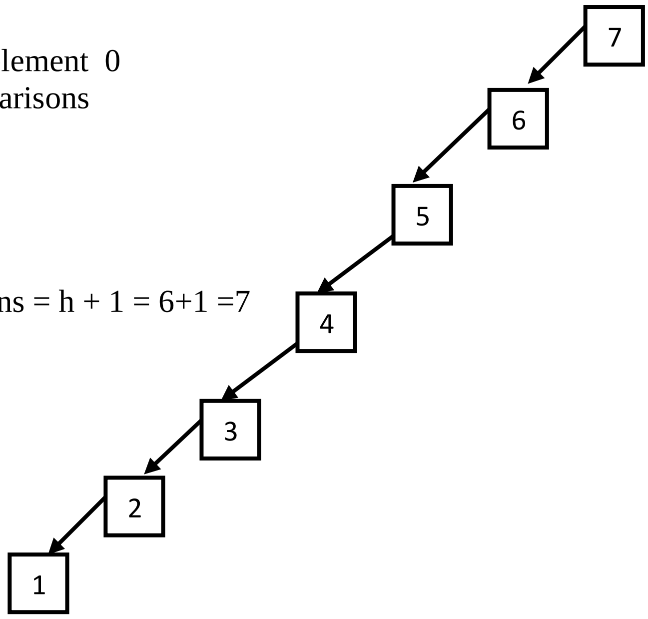
Cost of looking for an element in a binary search tree consists the elements 1,2,3,4,5,6,7

CASE 1

A - Looking for the element 0
costs 7 key comparisons

⟶ **h = N-1 = 6**

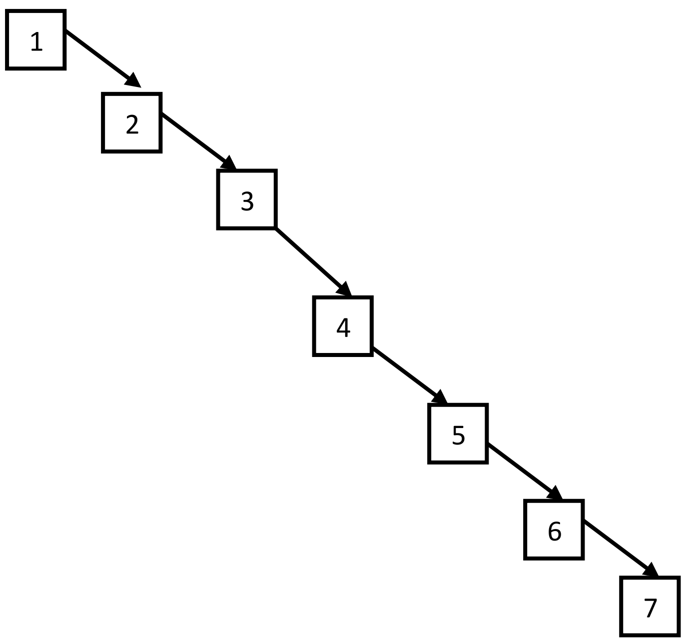
⟶ no of comparsions = h + 1 = 6+1 =7



B- Looking for the element 8
costs 7 key comparisons

⟶ **h = N-1 = 6**

⟶ no of comparsions = h + 1 = 6+1 =7

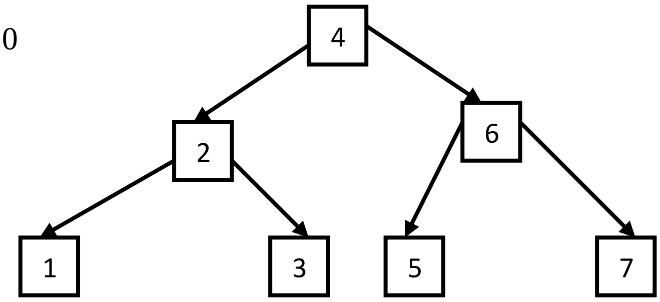


<u>CASE 2</u>

Looking for the element 8 or 0
costs 3 key comparisons

→ h = trunc (**$\log_2 N$**) = **2**

→ no of comparisons =
h + 1 = 2 + 1 = 3

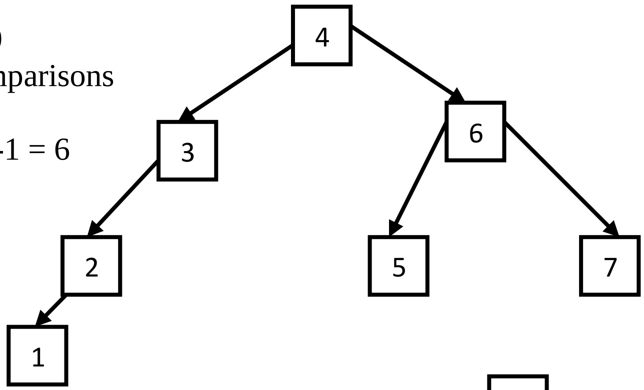


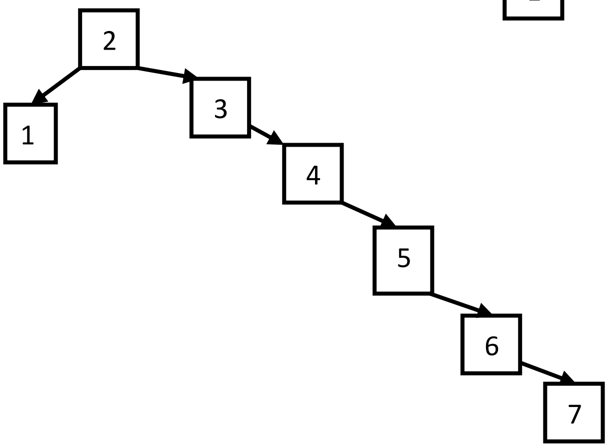
<u>CASE 3</u>

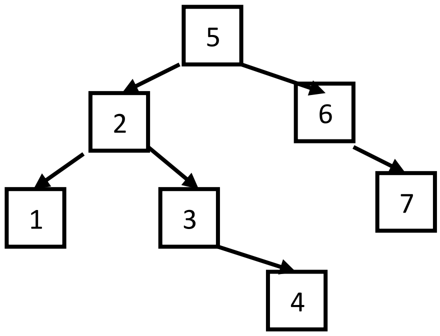
Looking for the element 8 or 0
costs between 3 and 7 key comparisons

→ 2 = trunc($\log_2 7$) < h < N-1 = 6

→ 3 < no of comarisons < 7







……………………………………………………………..
…………………………………………………………….
………………………………………………………….
………………………………………………………….

** The Height of a binary search tree : trunc ($\log_2 N$) <= h <= N –1 ,
where N number of keys

<u> Differences between worst case complexities depends on the Height of B.S.T. :</u>**

1- If the Height of B.S.T. is equal to **<u>N – 1 :</u>**

$\Rightarrow$ **Worst case complexity = h + 1 = N**
$\Rightarrow$ **Worst case complexity is O(N)** **(** ***worst worst case complexity*** **)**

2- If the Height of B.S.T. is equal to **<u>trunc ($\log_2 N$) :</u>**

$\Rightarrow$ **Worst case complexity = h + 1 = trunc($\log_2 N$) + 1**
$\Rightarrow$ **Worst case complexity is O($\log_2 N$)** **(** ***best worst case complexity*** **)**

3- If the Height of B.S.T. is between **<u>N – 1</u>** <u>and</u> **<u>trunc($\log_2 N$) :</u>**

$\Rightarrow$ **Worst case complexity = 2*Ln N $\approx$ 1.386* $\log_2 N$**
$\Rightarrow$ **Worst case complexity is O($\log_2 N$)** **(** ***average worst case complexity*** **)**

<u>Constructing Balanced tree (perfectly) :</u>
To reduce the comparisons is better to use balanced tree.
Two method to construct balanced tree :
1- Weight balancing tree
2- Height balancing tree

Perfectly balanced tree is a binary search tree where number of nodes in the left and the right
Subtrees differ by at most 1.

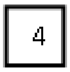
First algorithm (Weight balancing tree) :
No of keys is equal to N
1- Choose one element in the root
2- Choose (N DIV 2) elements to construct left balanced tree (recursively)
3- Remaind keys (N – (N DIV 2) – 1) to construct right balanced tree (recursivley)

Example :

construct a balanced tree (perfectly) of the following keys :

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

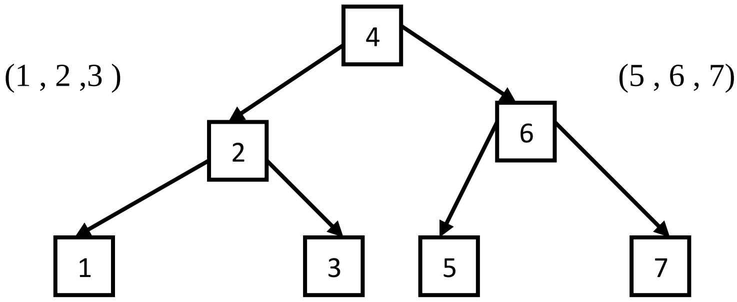
Choose 4 in the root :



Construct a left tree of (7 div 2) = 3 : 1 , 2 , 3 (recursively)

And the remaind keys = N – (N div 2) – 1 = 7 – 3 – 1 = 3 to construct a right tree :

5, 6, 7 (recursively)



Storing in AVL_trees :

Name : **A**del'son **V**elskii **L**andis .

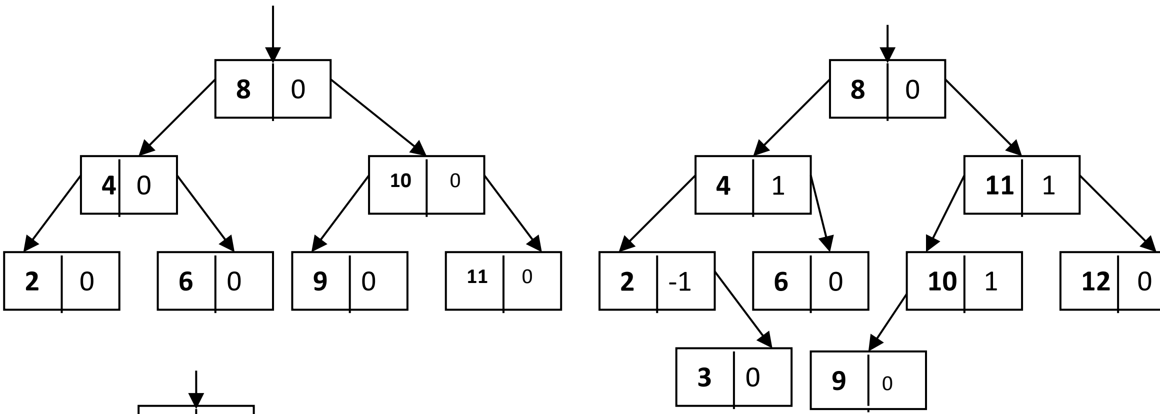
AVL tree is a binary search tree in which for every node the Heights of the left and right subtrees differ by at most 1 .

Data structure of AVL_tree :

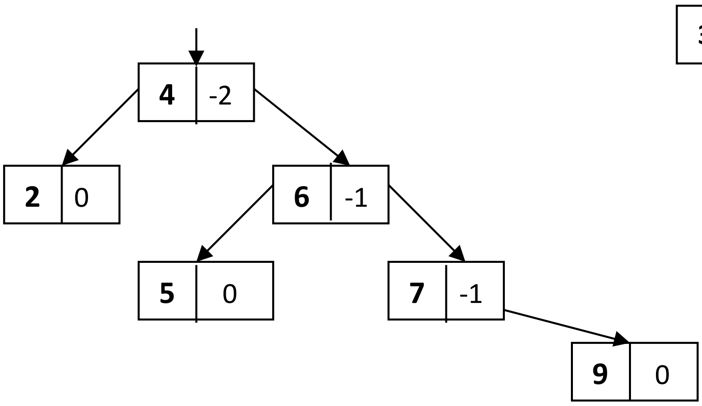
```
class  TreeNode ;
  {
      Object info ;
      TreeNode left;
      TreeNode right ;

      int   balance ;    //   balance ∈   [-1, 0 , 1 ]
  }
```

Where balance = Height of left subtree - Height of right subtree;

Examples :



Not AVL

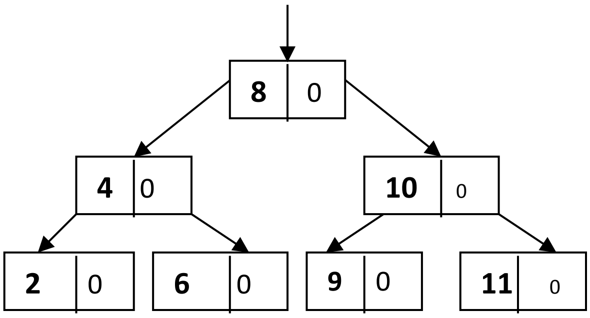


INSERTION IN AVL TREES :

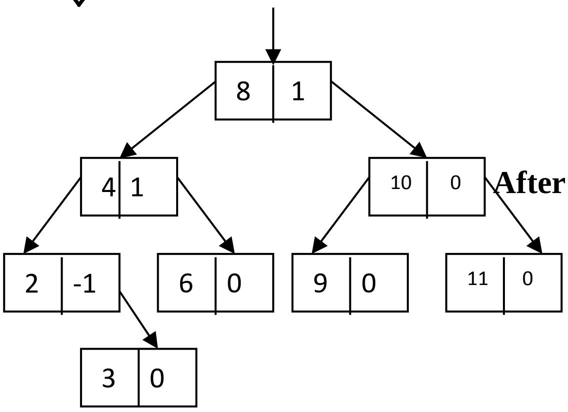
The AVL tree must have its properties after each insertion a new element.

Examples :

1- Insert the element **3** in the following AVL tree :

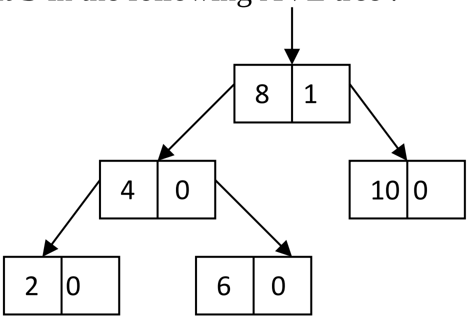


⇓

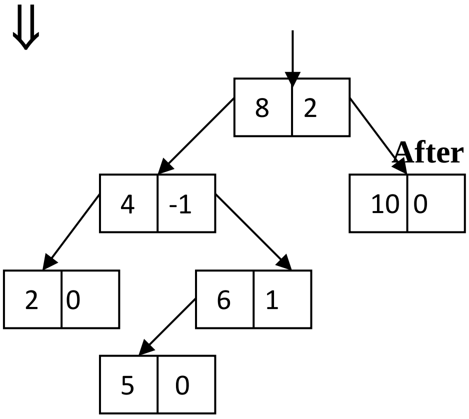


After insertion 3 still AVL tree

2- Insert the element **5** in the following AVL tree :



⇓



After insertion 3 <u>not</u> AVL tree

**** How will be reconstructed the AVL tree after insertion ?**

There are two categories of problems :

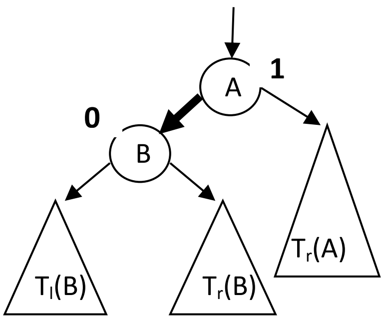
1- **<u>L-Rotation :</u>**
   with two versions :
   a- **LL-Rotation**
   b- **LR-Rotation**

2- **<u>R-Rotation</u> :**
   with two versions :
   a- **RL-Rotation**
   b- **RR-Rotation**

<u>L-Rotation :</u>

Suppose we have AVL tree in which we would like to insert a new element with the following two conditions :

- **A** is ***pivot*** node with ***balance*** = 1 , where A the last node with balance ≠ 0 in the search path
- Insert in **<u>Left</u>** subtree of **A** ,where the root of the left subtree is **B** with ***balance*** = 0



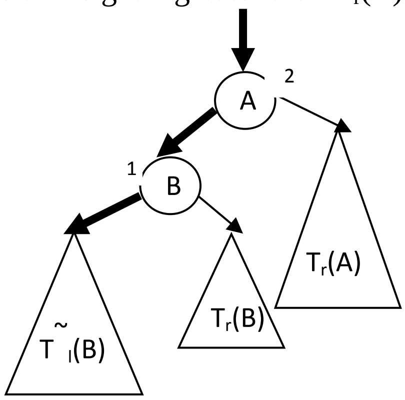
Where the Heights of $T_l(B)$, $T_r(B)$ and $T_r(A)$ are the same

Now there are two cases to consider :

1.1 First case LL-Rotation :

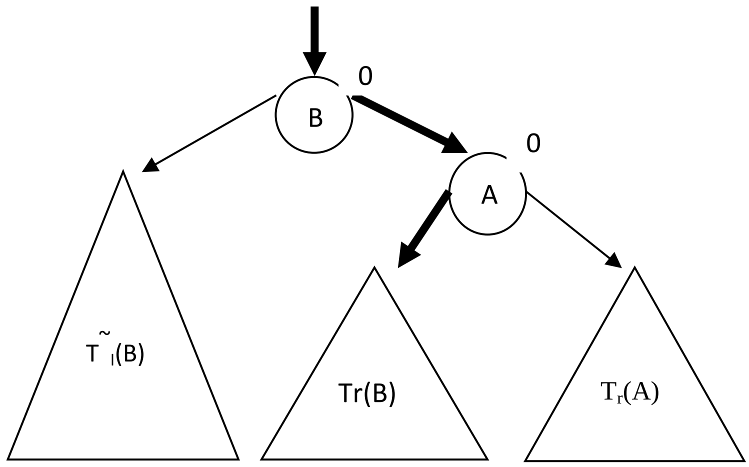
Insert in the left subtree $T_l(B)$ of B

$\Rightarrow \tilde{T}_l(B)$ with Height 1 greater than $T_r(B)$



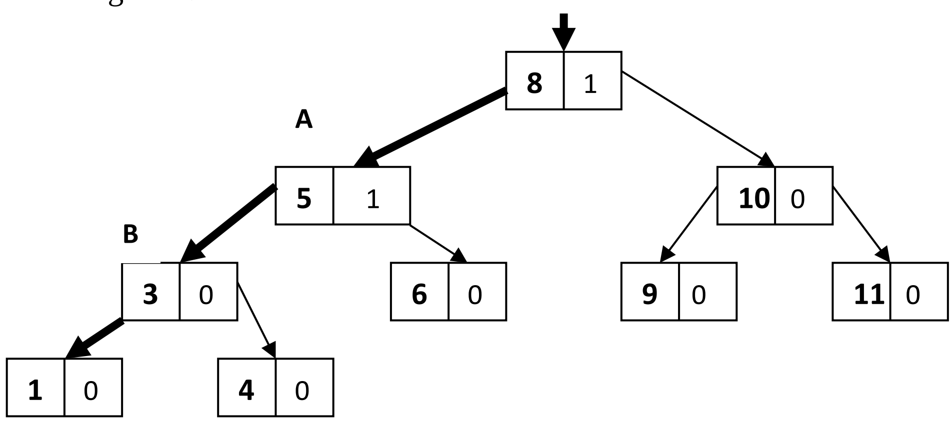
Restructuring the above tree as follows :

1- The pointer , which points to A becomes a pointer to B
2- Right pointer of B becomes a pointer to A
3- The pointer , which points to B as (left pointer of A) becomes a pointer to $T_r(B)$

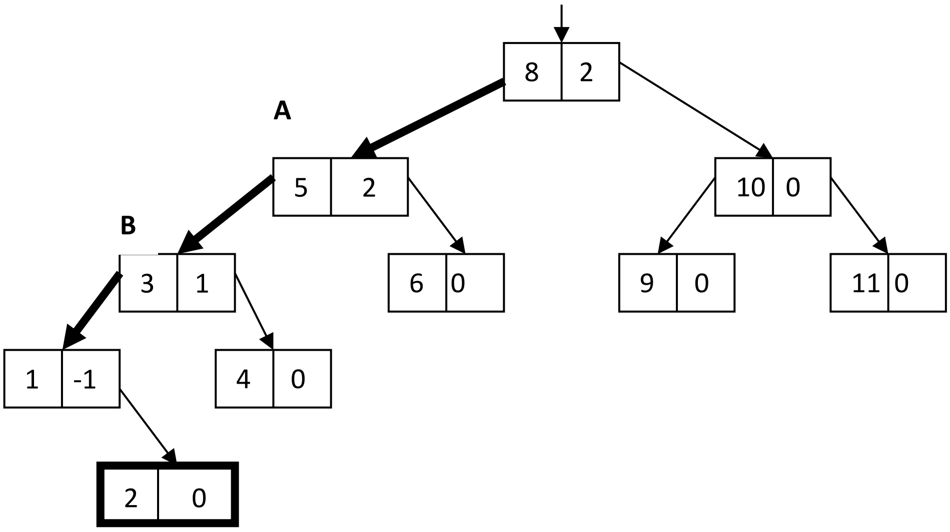


Example :

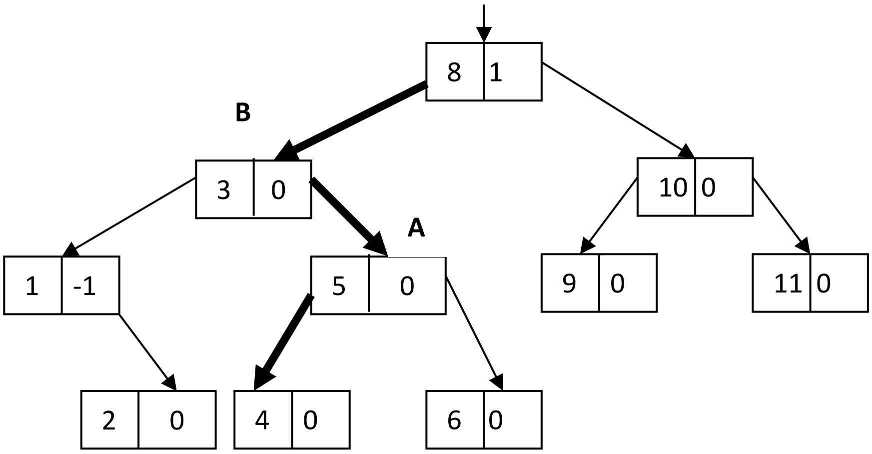
Insert **2** in the following tree :



⇓

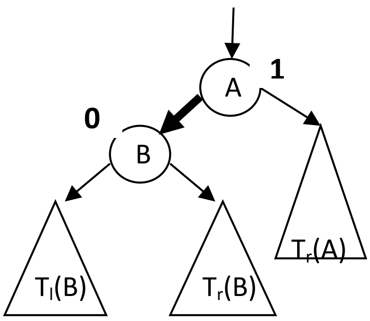


Restructuring ⇒

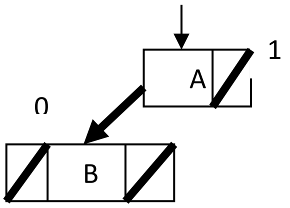


1.2 Second case LR :

Insert in the right subtree $T_r(B)$ of B

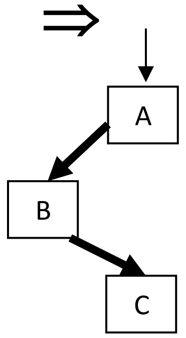


a- $T_r(B) = NULL \Rightarrow T_l(B) = T_r(A) = NULL$



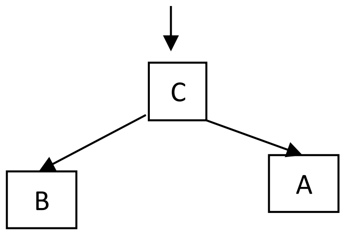
Insert **C** as new element in the right subtree of **B** .

$\Rightarrow$



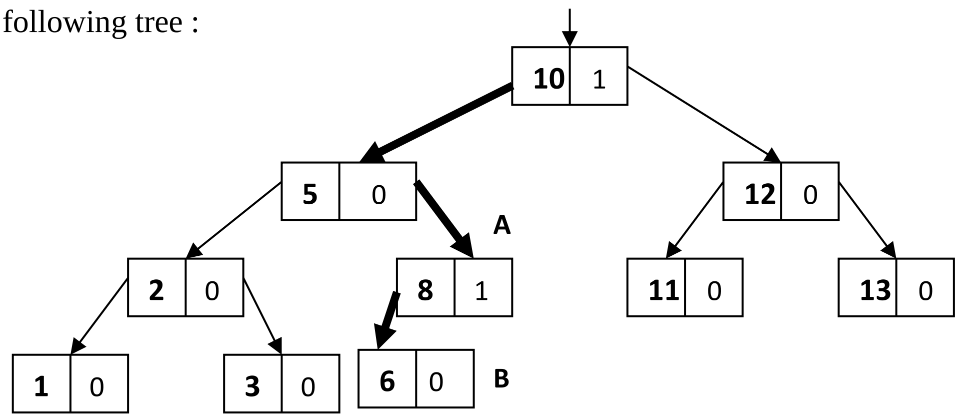
Restructuring the above tree as follows :

1- The pointer , which points to A becomes a pointer to C
2- Right pointer of C becomes a pointer to A
3-Left pointer of C becomes a pointer to B

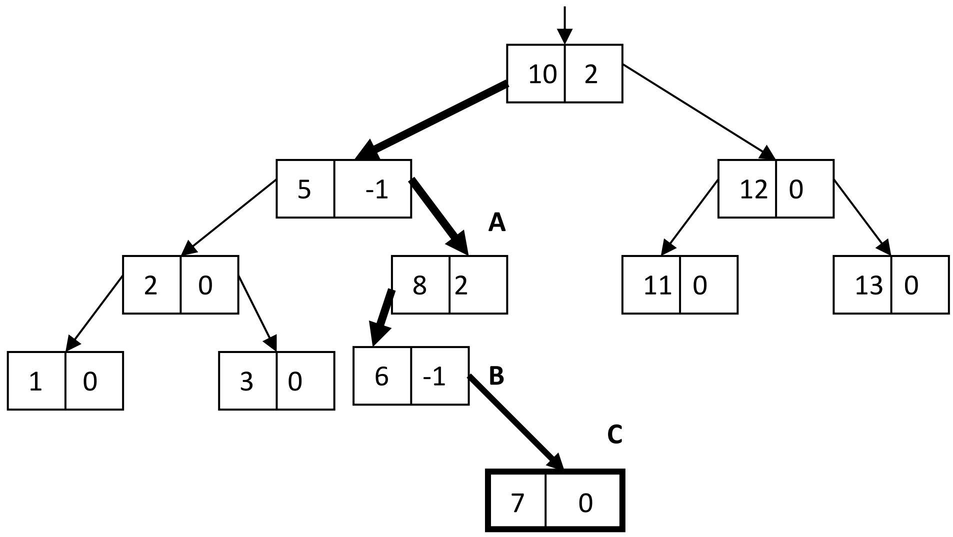


Example :

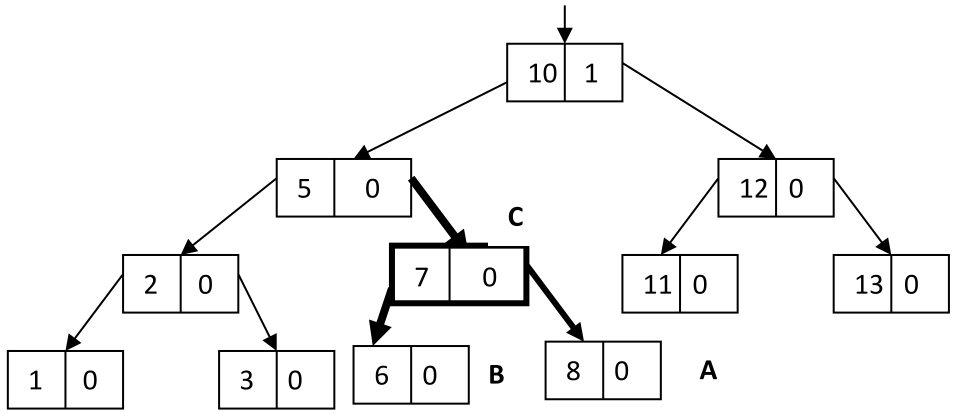
Insert 7 in the following tree :



⇓

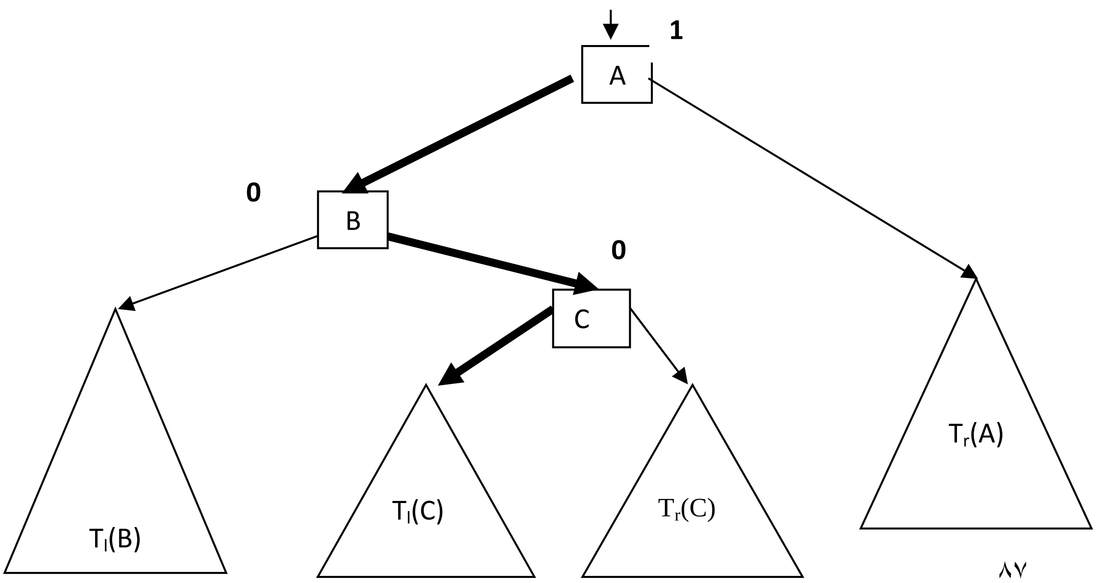


Restructuring ⇒

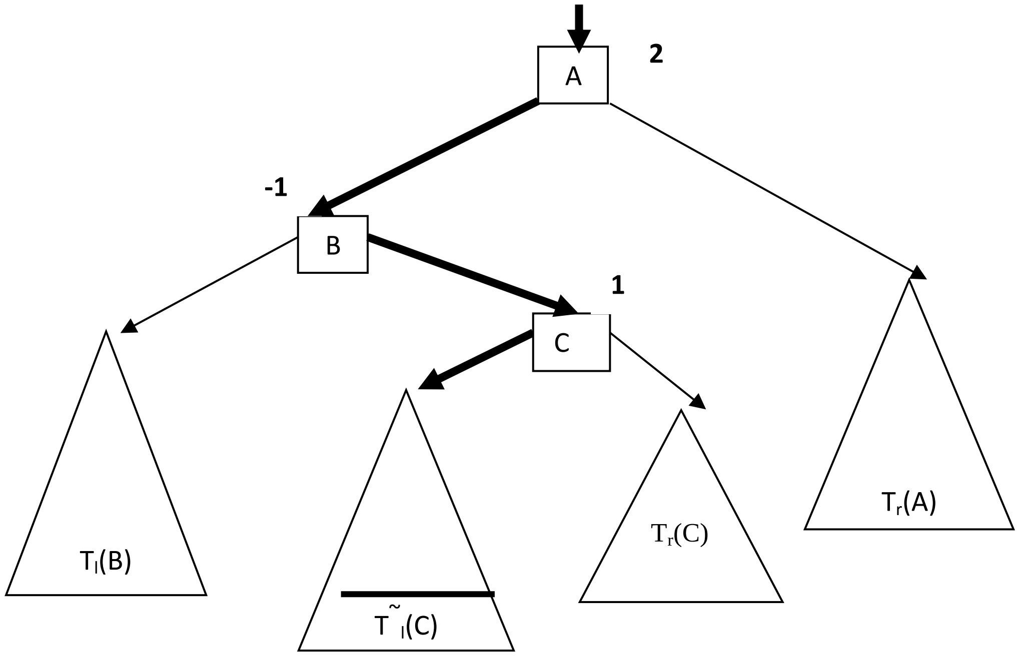


b- Right subtree of **B** ***not*** **NULL** with root **C** (left and right subtree of **C** possibly **NULL**)

- Insert in the ***left*** subtree of C (in T_l(C))

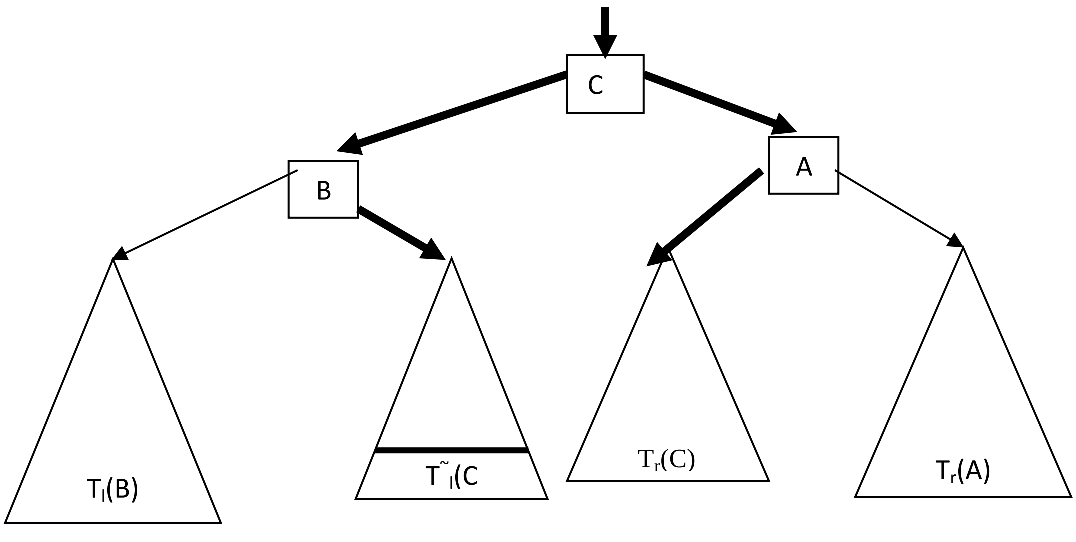


After insertion ⇒



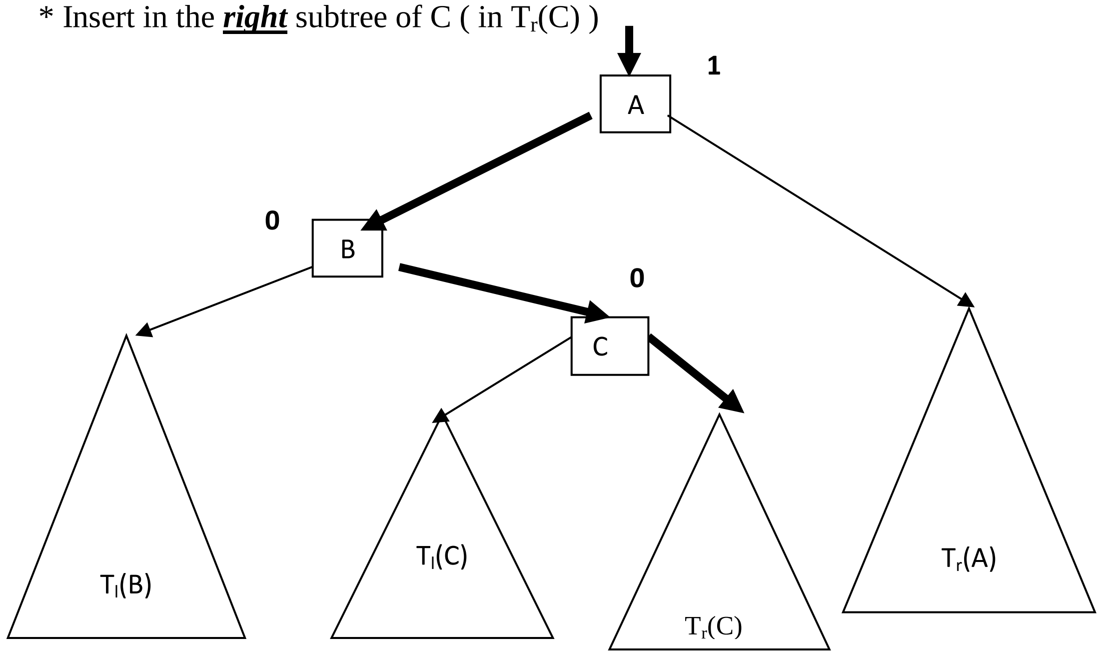
RESTRUCTURING ⇒

1- The pointer , which points to A becomes a pointer to C
2- Right pointer of C becomes a pointer to A
3- Left pointer of C becomes a pointer to B
4- Left pointer of A becomes a pointer to Tr(C)
5- Right pointer of B becomes a pointer to **$\underline{\tilde{T}_l(C)}$**

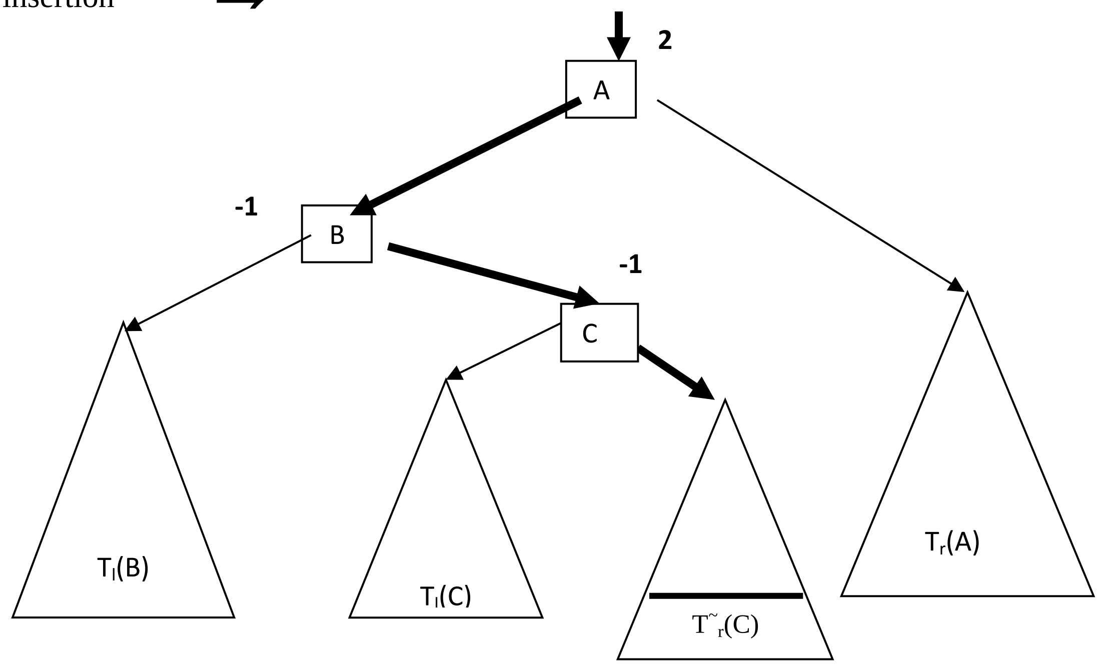


C- Right subtree of **B** ***not*** **NULL** with root **C** (left and right subtree of **C** possibly **NULL**)

* Insert in the ***right*** subtree of C (in $T_r(C)$)

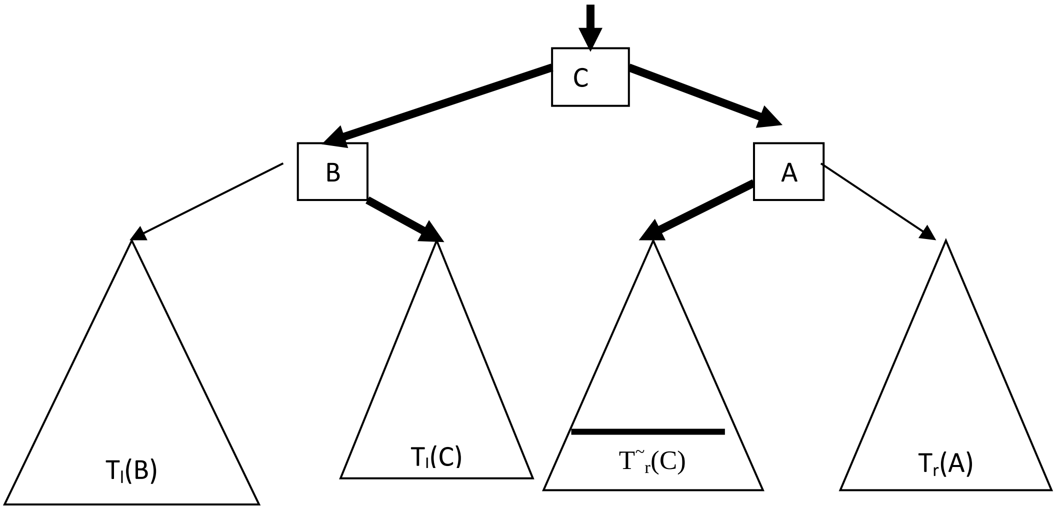


After insertion $\Rightarrow$



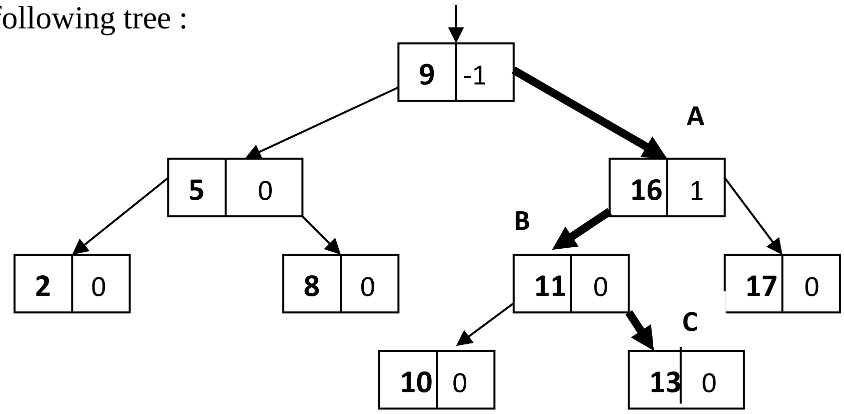
RESTRUCTURING $\Rightarrow$

1- The pointer , which points to A becomes a pointer to C
2- Right pointer of C becomes a pointer to A
3- Left pointer of C becomes a pointer to B
4- Left pointer of A becomes a pointer to **$\tilde{T}_r(C)$**
5- Right pointer of B becomes a pointer to $T_l(C)$

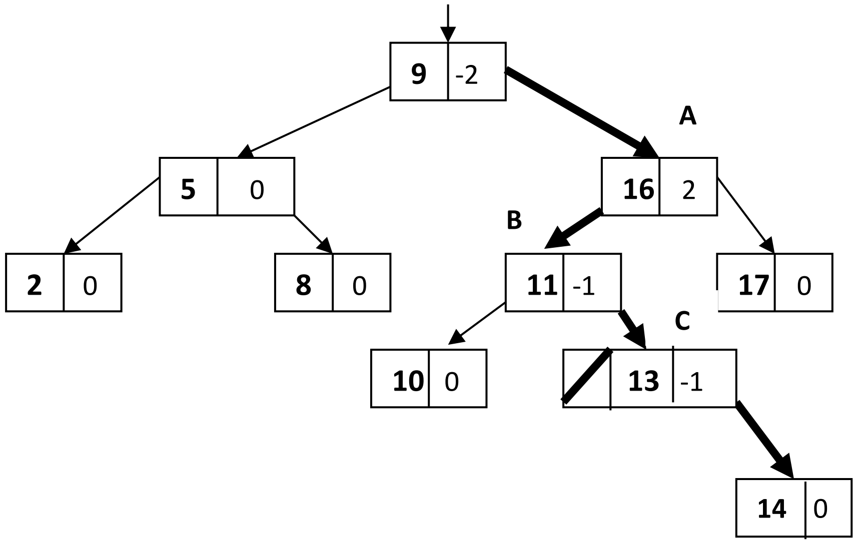


Example :

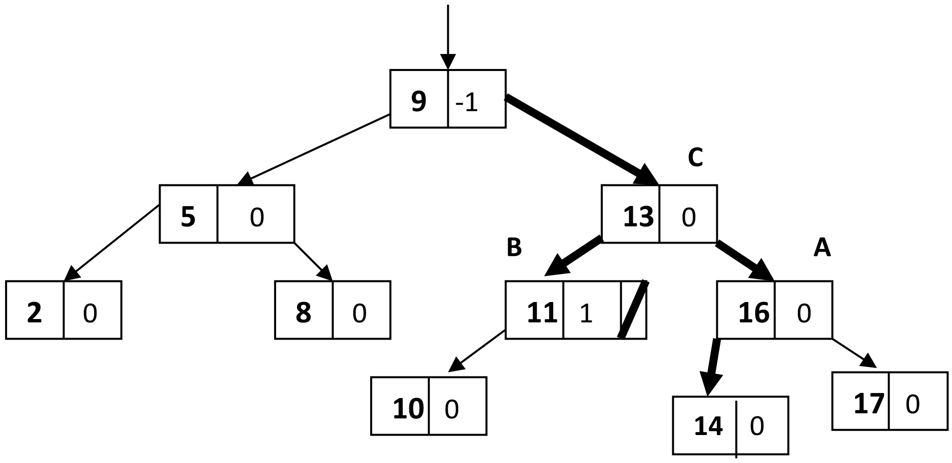
Insert **12** **or 14** in the following tree :



$\Downarrow$



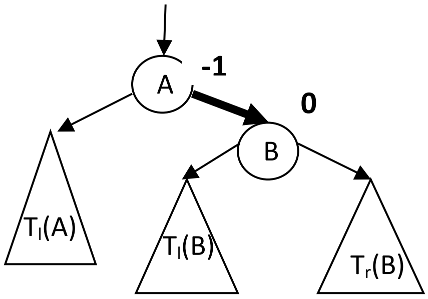
Restructuring $\Rightarrow$



<u>R-Rotation :</u> (Mirroring for L-Rotation)

Suppose we have AVL tree in which we would like to insert a new element with the following conditions :

- **A** is ***pivot*** node with ***balance*** = -1 , where A the last node with balance $\neq$ 0 in the search path
- Insert in <u>**Right**</u> subtree of **A** ,where the root of the right subtree is **B** with ***balance*** = 0



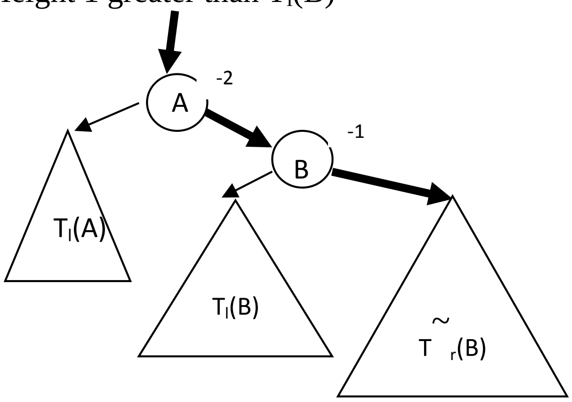
Where the Heights of $T_l(B)$, $T_r(B)$ and $T_l(A)$ are the same

Now there are two cases to consider :

<u>2.1 First case RR-Rotation :</u>

Insert in the right subtree $T_r(B)$ of B

$\Rightarrow \tilde{T}_r(B)$ with Height 1 greater than $T_l(B)$

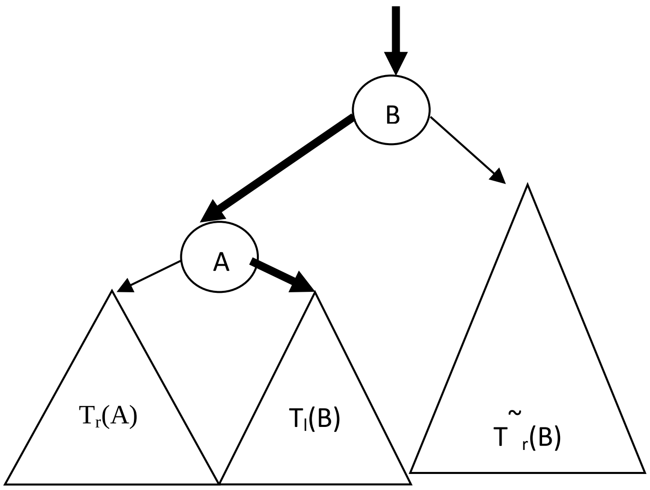


Restructuring the above tree as follows :

1- The pointer , which points to A becomes a pointer to B

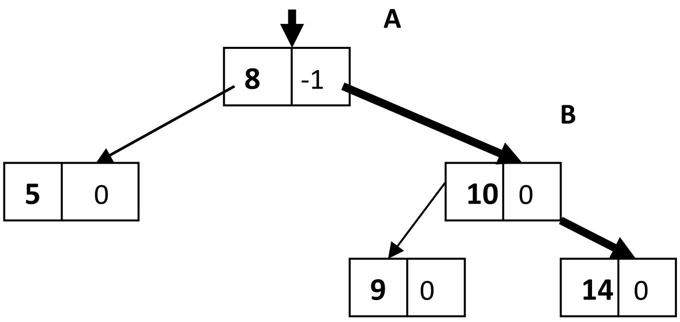
2- Left pointer of B becomes a pointer to A

3- The pointer , which points to B as (right pointer of A) becomes a pointer to $T_l(B)$



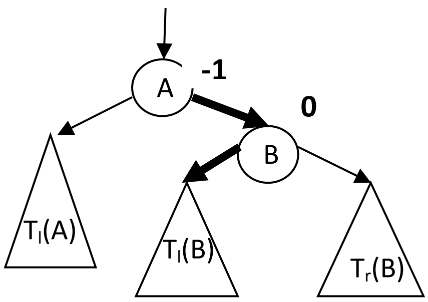
Example :

Insert **12 or 16** in the following tree :

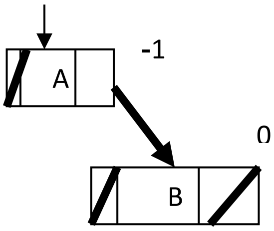


2.2 Second case RL :

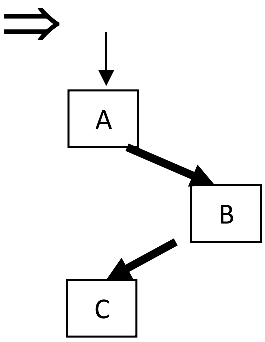
Insert in the left subtree $T_l(B)$ of B



a- $T_l(B) = NULL \Rightarrow T_r(B) = T_l(A) = NULL$

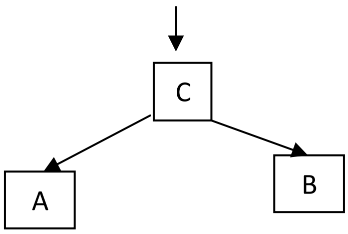


Insert **C** as new element in the right subtree of **B** .



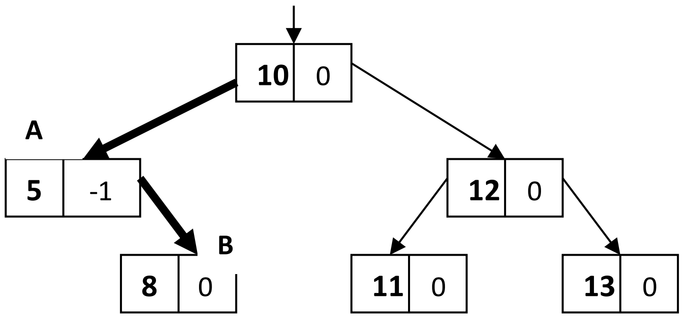
Restructuring the above tree as follows :

1- The pointer , which points to A becomes a pointer to C
2- Right pointer of C becomes a pointer to B
3-Left pointer of C becomes a pointer to A



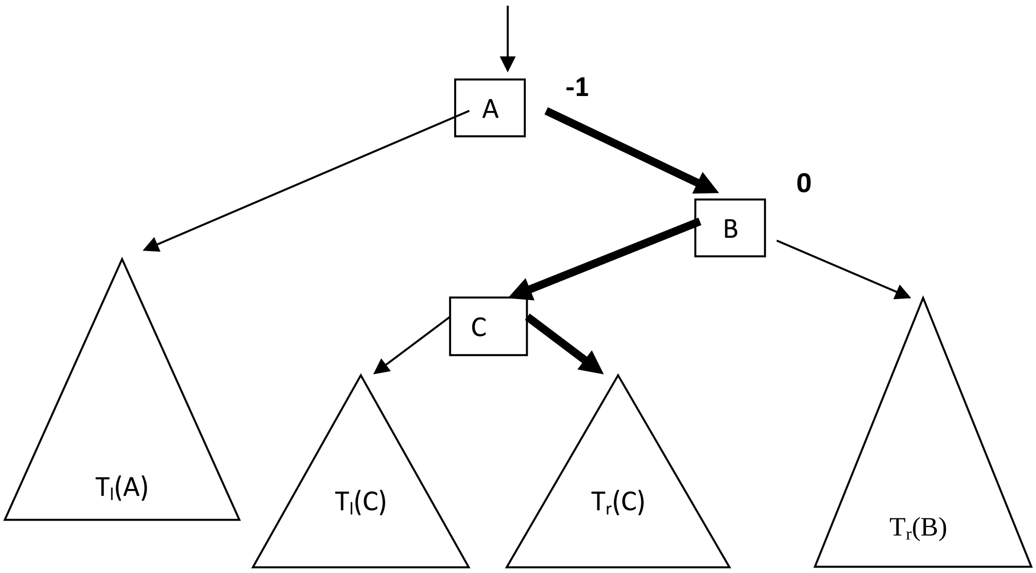
Example :

Insert 7 in the following tree :

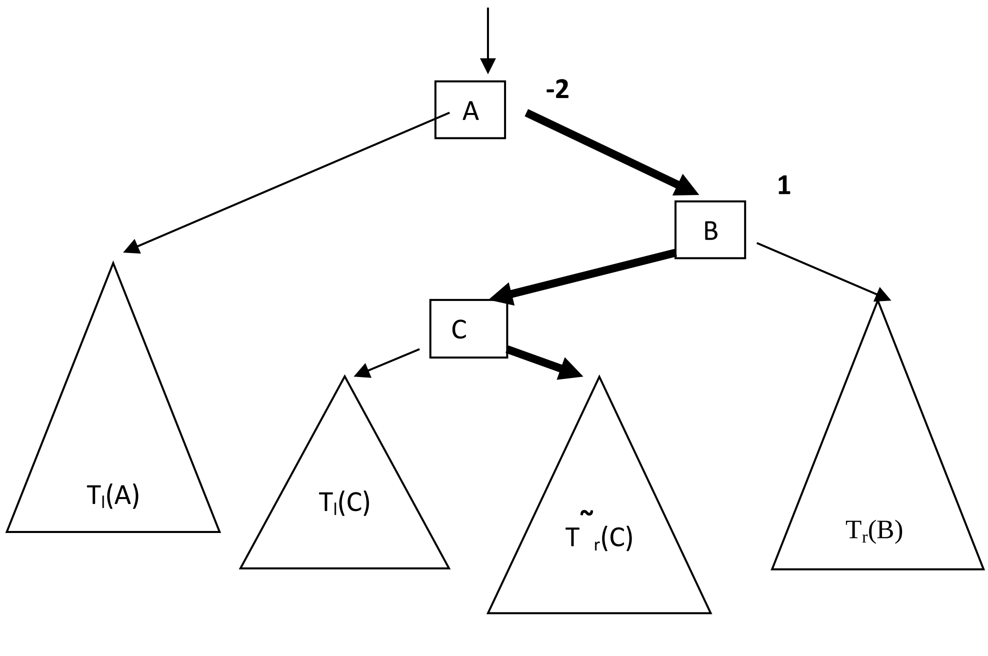


b- Left subtree of **B** ***not*** **NULL** with root **C** (left and right subtree of **C** possibly **NULL**)

- Insert in the ***right*** subtree of C (in $T_r(C)$)



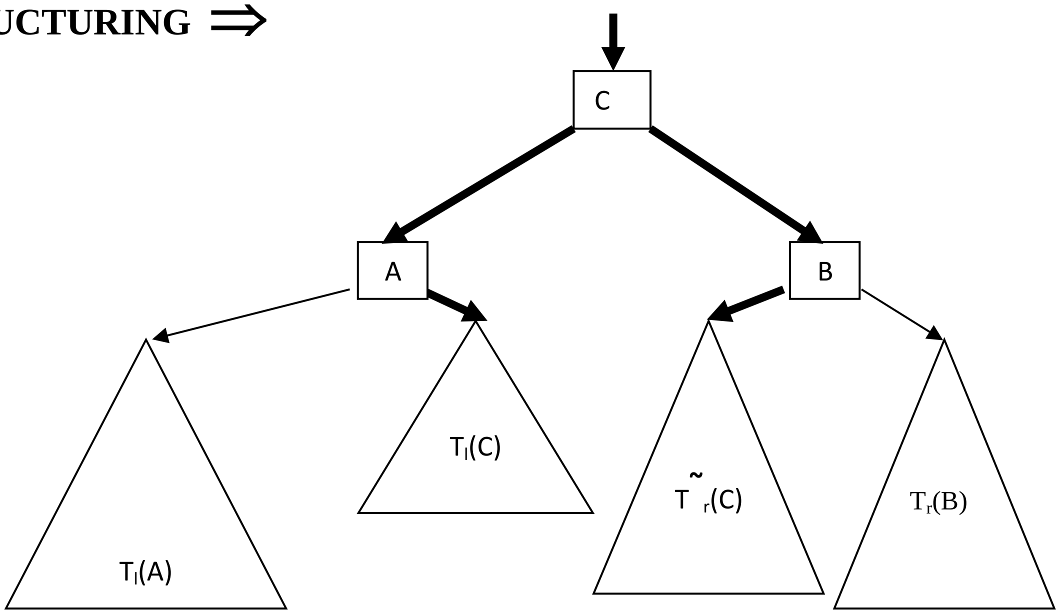
After insertion ⇒



RESTRUCTURING ⇒

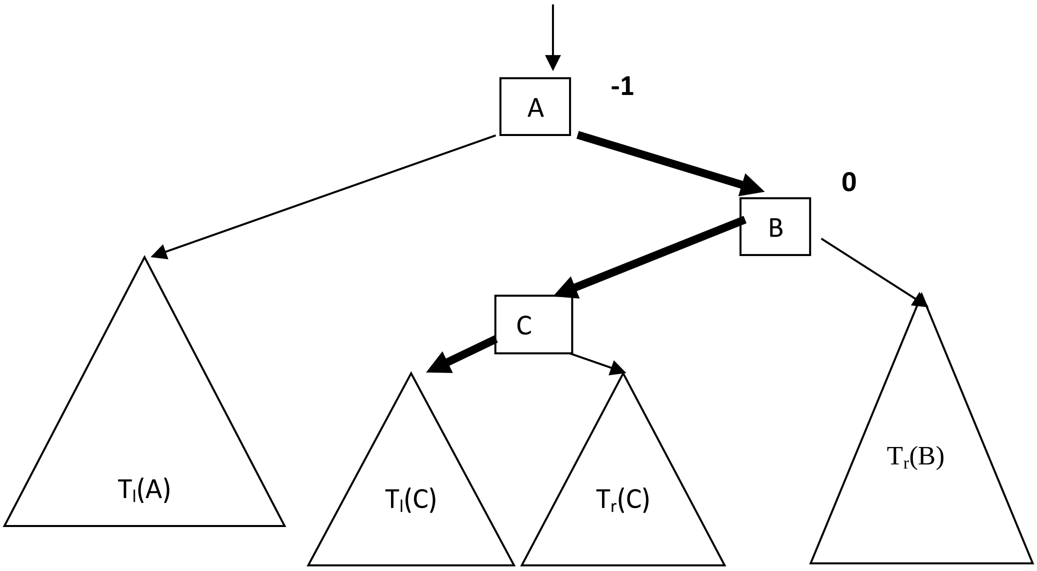
1- The pointer , which points to A becomes a pointer to C
2- Left pointer of C becomes a pointer to A
3- Right pointer of C becomes a pointer to B
4- Right pointer of A becomes a pointer to $T_l(C)$
5- Left pointer of B becomes a pointer to **$\underline{\tilde{T}_r(C)}$**

RESTRUCTURING ⇒

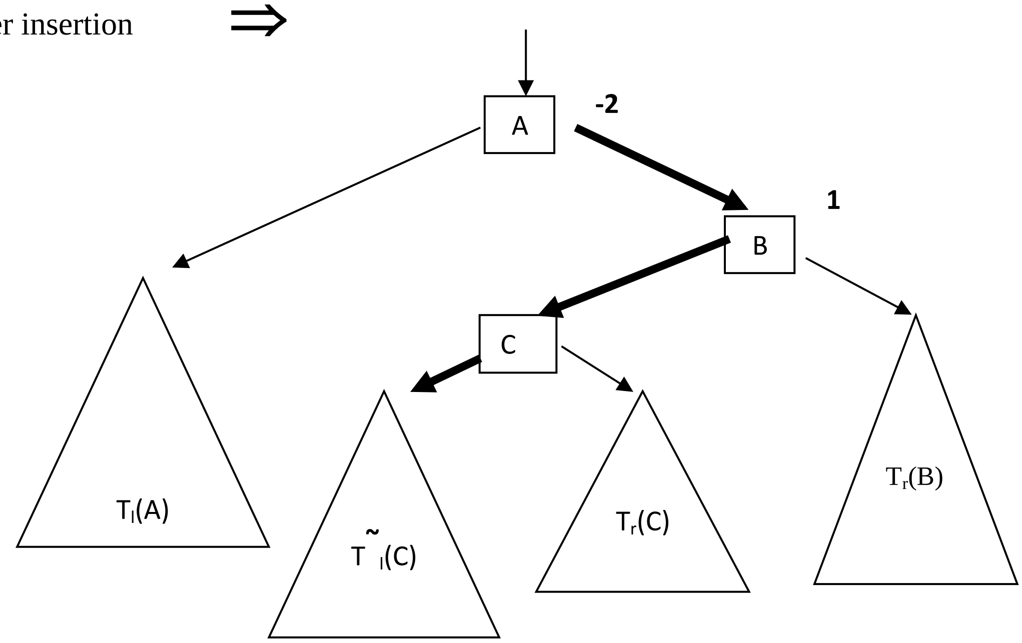


C- Left subtree of **B** ***not*** **NULL** with root **C** (left and right subtree of **C** possibly **NULL**)

- Insert in the ***left*** subtree of C (in $T_l(C)$)



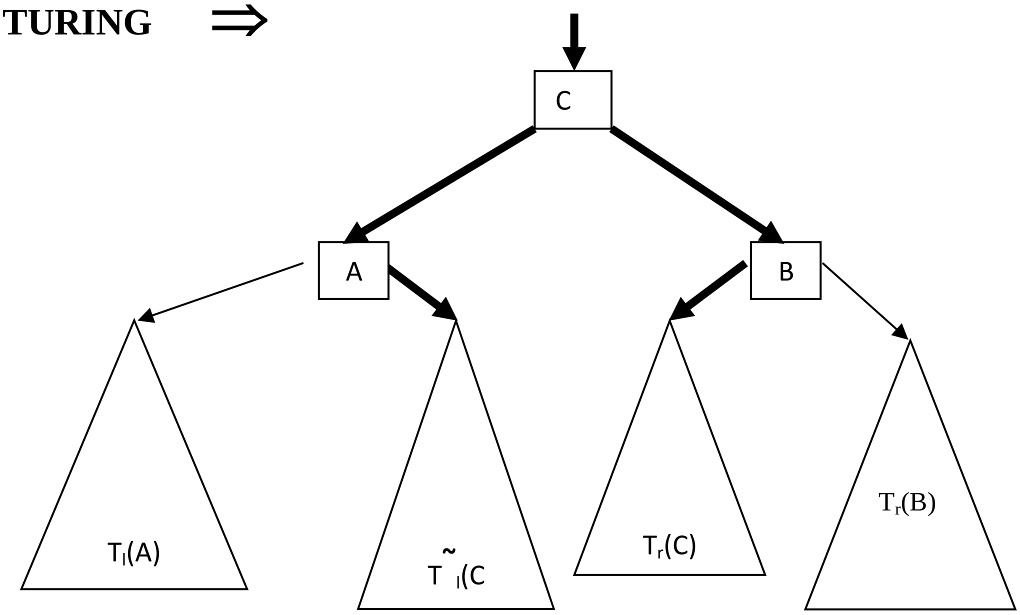
After insertion ⇒



RESTRUCTURING ⇒

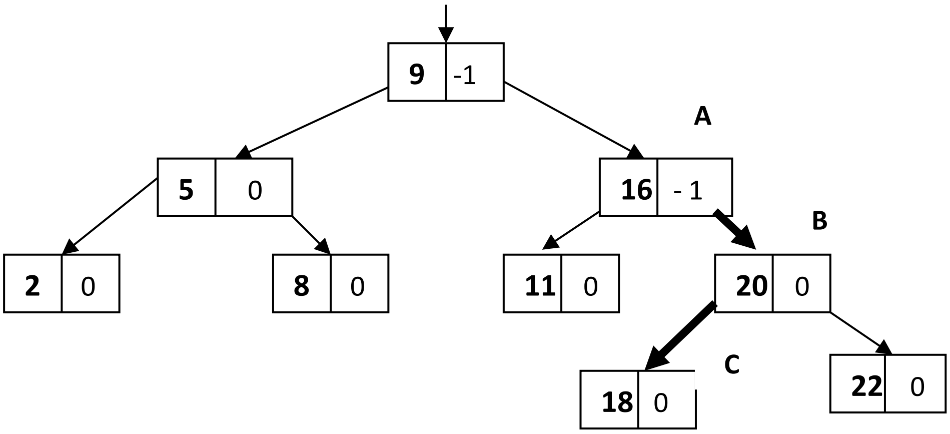
1- The pointer , which points to A becomes a pointer to C
2- Left pointer of C becomes a pointer to A
3- Right pointer of C becomes a pointer to B
4- Right pointer of A becomes a pointer to **$\underline{\tilde{T}_l(C)}$**
5- Left pointer of B becomes a pointer to $T_r(C)$

RESTRUCTURING ⇒



Example :

Insert **17** **or 19** in the following tree :



Complexity of searching in AVL-tree :

Suppose we have a binary search tree contains N elements with height h with

$$\mathbf{trunc(Log_2N) \;<=\; h \;<=\; N-1}$$

THEN the worst case complexity of B.S.T. could be one of the following :

Case 1 : $W(N) = h + 1 = N \quad \Rightarrow \quad W(N)$ is $O(N)$

Case 2 : $W(N) = h + 1 = trunc(log_2N) + 1 \quad \Rightarrow \quad W(N)$ is $O(log_2N)$

Case 3 : $W(N) = h + 1 = 1.386log_2N \quad \Rightarrow \quad W(N)$ is $O(log_2N)$

** Now we calculate the worst case complexity of searching in AVL-tree :

1- First when we have a B.S.T. with N elements and height h $\Rightarrow$

$$h + 1 <= N <= 2^{h+1} - 1$$

$N <= 2^{h+1} - 1$

$\Leftrightarrow 2^{h+1} >= N + 1$

$\Leftrightarrow h + 1 >= log_2(N+1)$

$\Leftrightarrow \mathbf{h >= log_2(N+1) - 1}$ **1**

2- Now suppose we have an AVL-tree with height h

Define $\mathbf{N_h}$ as the least number of elements in AVL-tree with height h $\Rightarrow$

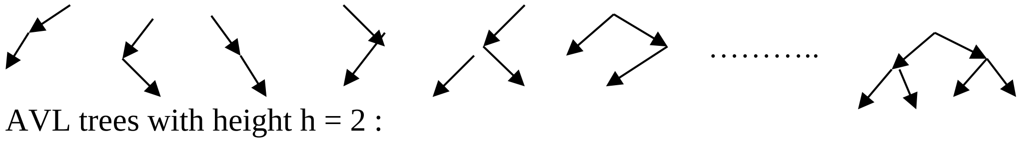
$h = 0 \Rightarrow N_0 = 1$

$h = 1 \Rightarrow N_1 = 2$

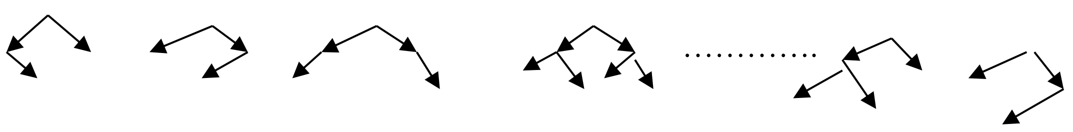
$h >= 2 \Rightarrow \mathbf{N_h = 1 + N_{h-1} + N_{h-2}}$, where N_{h-1} and N_{h-2} the least number of elements in the left and the right subtrees of the AVL-tree

Example :

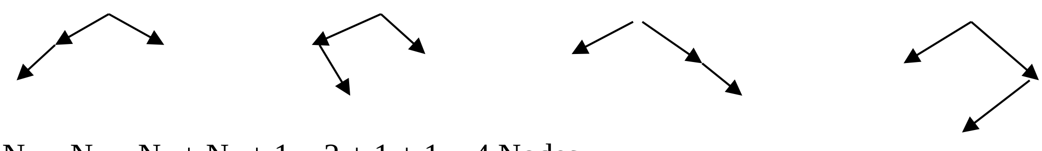
- Determine all B.S.T. with height h = 2 :



- Determine all AVL trees with height h = 2 :



- Determine all AVL trees with least number of nodes and with height h = 2 :



$N_h = N_2 = N_1 + N_0 + 1 = 2 + 1 + 1 = 4$ Nodes

Using **FIB** number and **induction** $\Rightarrow$
$\forall\ h >= 0 : N_h = 1 + N_{h-1} + N_{h-2} = FIB(h+3) - 1$

Because $N >= N_h \Rightarrow N >= N_h = FIB(h+3) - 1$

Definition from Math. :
FIB(k) > $1/\sqrt{5}*X^k - 1$, Where X = $1/2\ (1 + \sqrt{5}\ )$

$\Rightarrow N > 1/\sqrt{5}*X^{h+3} - 2$

…………………..

………………

$\Rightarrow$ **h < $1.4404\log_2(N+2) - 1.328$ …………………….. 2**

FROM **1** and **2** $\Rightarrow \log_2(N+1) - 1 <= h < 1.4404\log_2(N+2) - 1328$

Approximately the height of AVL-tree with N elements is $\log_2 N + 0.25$

$\Rightarrow$ The worst case complexity of AVL-tree with height h for searching is $O(\log_2 N)$

CH6: STORING IN MULTIWAY TREES

MULTIWAY SEARCH TREE:

M.W.S.T of order **n** , **n >= 2** defined as follows :

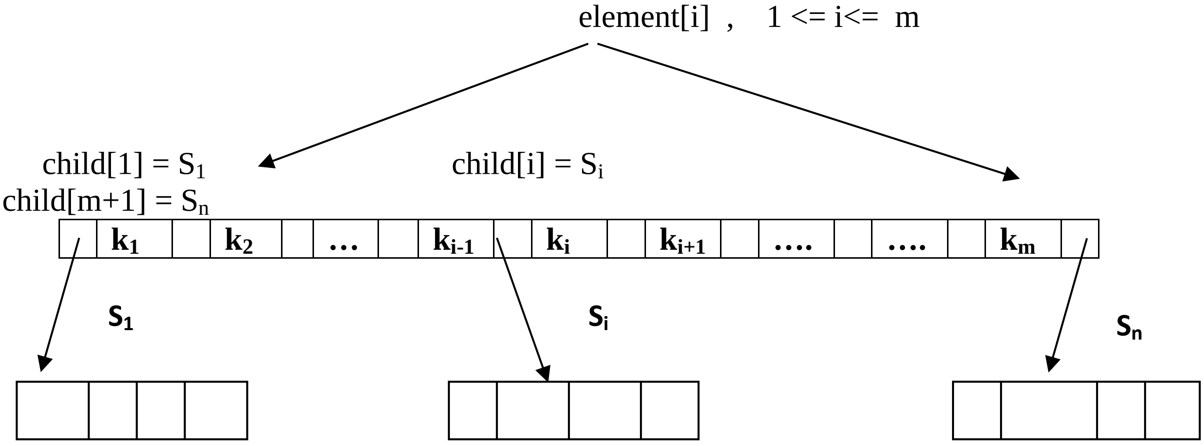
1- Every Node contains between **1** and **m** records sorted in **increasing** order , where **1<= m <= n –1**
2- Max number of subtrees for each **NON**-leaf Node is **m+1** or **n**
3- A Node with **m** records (key of the records : k_1 , k_2 , …, k_{i-1} , k_i , k_{i+1} , …. , k_m) and with S_1 , S_2 , S_3 ,……………., S_{m+1} as subtrees of this Node

$\Rightarrow$

- For every record **X** in S_1 : **X.key <=** k_1
- for every record **X** in S_i : k_{i-1} < **X.key <=** k_i
- For every record **X** in S_{m+1} : **X.key >** k_m

Data structure for M.W.S.T. of order n :

```
class TreeNode
  {
     Object       element[m]  ;
     TreeNode  child[m+1] ; //  child[n]  OR child[order]
     int  number ; //number of keys in any node
   ……
   ……
  }
```



X.key <= k_1 k_{i-1} < **X.key <=** k_i **X.key >** k_m

Searching in M.W.S.T. :

Two conditions must be satisfied to find an element :

1- The node, which hold the element
2- The position in the node : 1<= i <= order -1

FUNCTION place(key , T)

: :

: :

if i = 1 : k <= T.Element[1].key ⇒ place(k,T) = 1

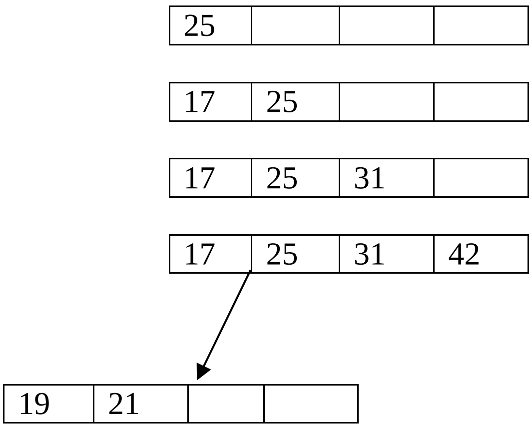
if 1 < i < order : T.Element[i-1].key < k <= T.Element[i].key ⇒ place(k,T) = i

if i = n : T.Element[order] < k ⇒ place(k,T) = order;

Examples :

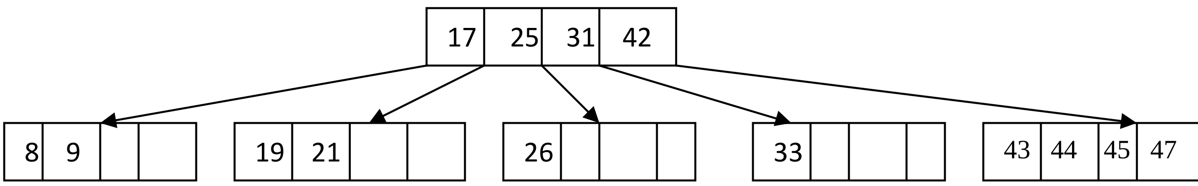
Build up a M.W.S.T. of order 5 using the following inputs :

a- 25 , 17 , 31 , 42 , 21 , 19 , 26 , 33 , 47 , 44 , 45 , 43 , 8 , 9

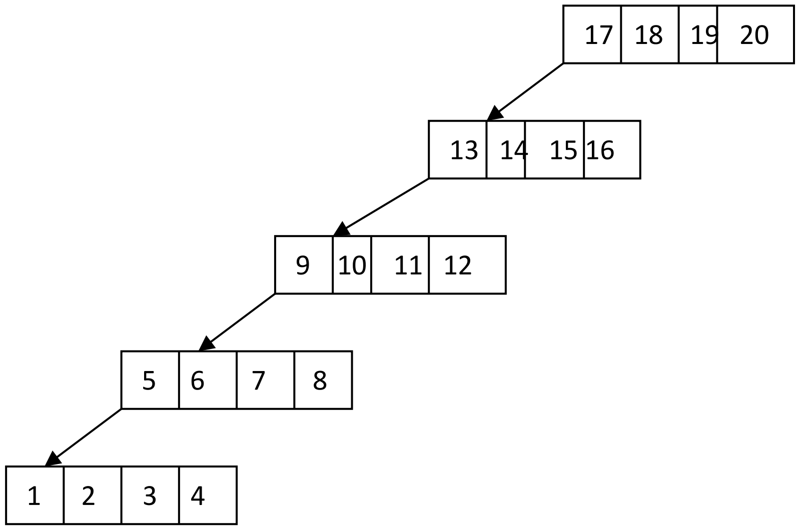


…………………

……………………



b- 20 , 19 , 18 , 17 , 16 , 15 , 14 , 13 , 12 , 11 ,10 , 9 , 8 ,7 , 6 , 5 , 4 , 3 , 2 ,1



B-TREE OF ORDER n >= 2 :

B-tree of order **n** is M.W.S.T. of order **n** such that :

1- The root node may contain between **1** and **n-1** elements
2- Each **NON**-root-node contains between **(n-1) DIV 2** and **n-1** elements
3- All leaves in the same level

Insertion in B-tree :

● B-tree **NULL** ⇒ Create new Node with the new element

● B-tree **NOT EMPTY** , find Node **N** which is **leaf** then :

1- **N NOT** full with number of elements < **n-1** ⇒ insert and then sort the elements in the Node

2- N full **(means : N has n-1 elements)** ⇒ Take the following steps :

a- Insert the new element with the elements in N sorted into temp one dim array **N'**
in increasing order ⇒ X_1**.key** < X_2**.key** < … < X_q < X_{q+1} < X_{q+2} …. < X_n**.key,**
where **q = (n-1) DIV 2**

b- Divide the elements in N' into two new leaves (left and right) as follows :
Left : X_1 , X_2 ,…….,X_q **Right :** X_{q+2}, ………,X_n ,
where **q = (n-1) DIV 2**
And the X_{q+1} will be inserted **(recursively)** to the parent P of N

c- ● **P** is parent of **N** ⇒ insert X_{q+1} in **P**
P NOT full ⇒ apply **1**
P full ⇒ apply **2** (recursively)
● **N** has no parent , Create new Node containing X_{q+1}

(Where the Node with X_1 , X_2 ,…….,X_q as left Node of X_{q+1} and
the Node with X_{q+2}, ……….,X_n as right Node of X_{q+1})

Example :

Insert with n = 5 :
41 , 61 , 36 , 53 , 55 , 52 , 49 , 43 , 67 , 45 , 69 , 71 , 63 , 65 , 57
in the following B-tree , where the order of the tree is n = 5 :

21	31	51	

SOLUTION :

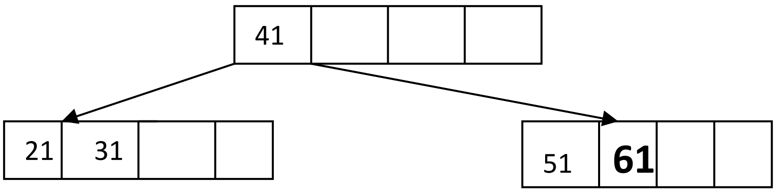
1- Insert 41 ⇒

21	31	**41**	51

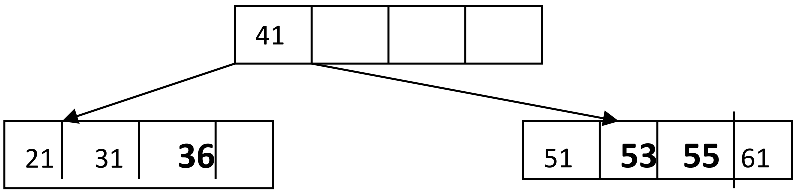
2- Insert 61 ⇒ calculate q = (n-1) DIV 2 = (5-1) DIV 2 = 2

Temp one dim array :

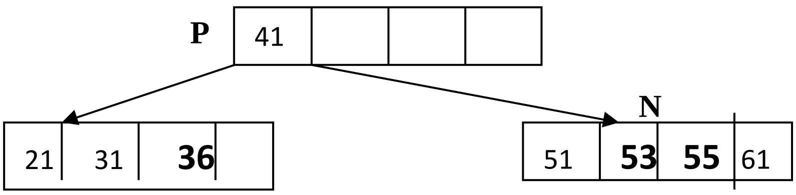
21	31	41	51	**61**
Left		to P	right	



3- Insert 36 , 53 , 55 ⇒



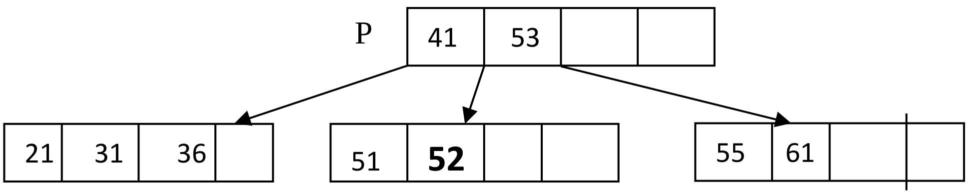
4- Insert 52 into following tree :



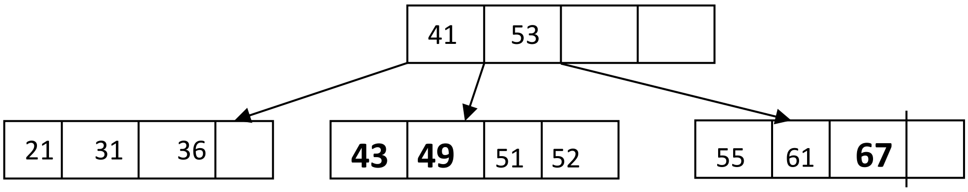
⇒ N is FULL and P parent of N is not full :

Temp one dim array :

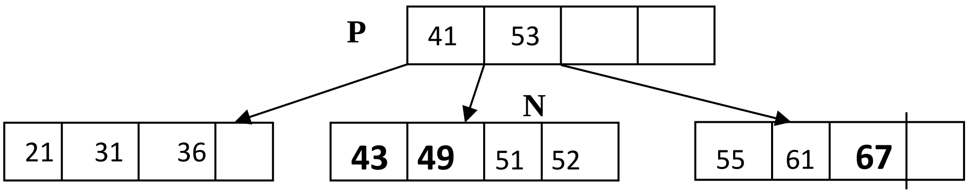
51	**52**	53	55	61
Left		to P	right	



5- Insert 49 , 43 , 67 ⇒



6- Insert 45 into following tree :

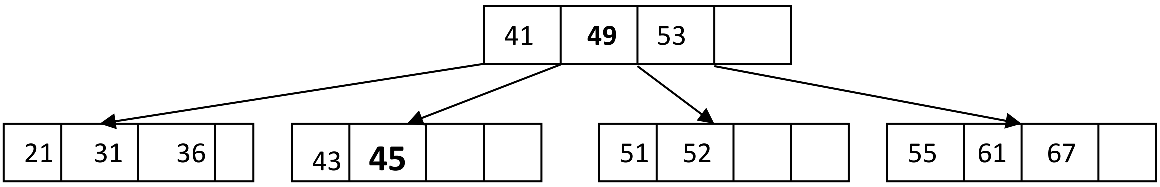


⇒ N is FULL and P parent of N is not full :

Temp one dim array :

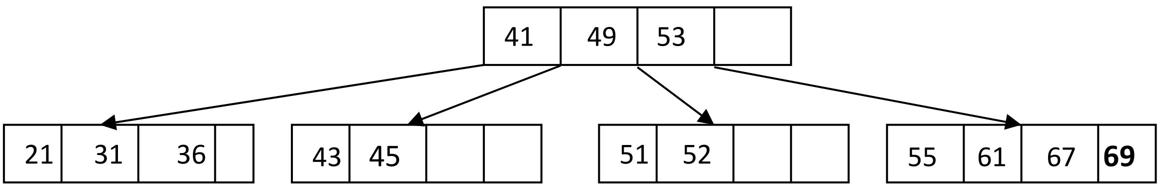
43	**45**	49	51	52
Left		to P	right	

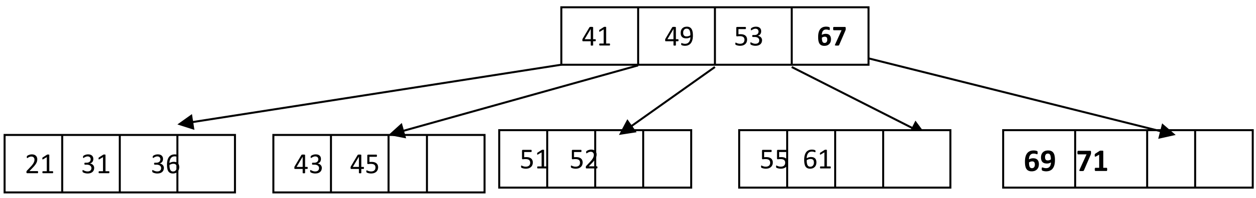
⇒



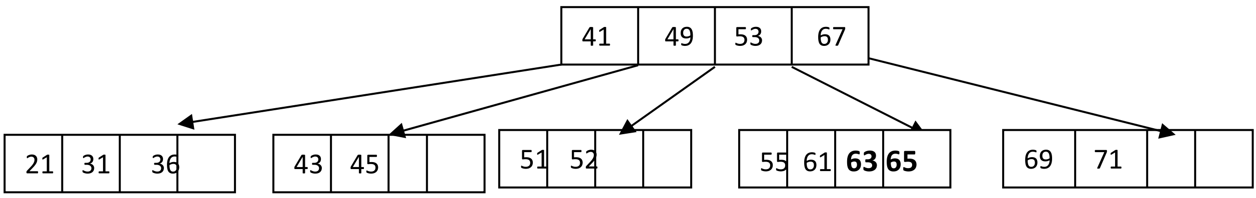
Left Leaf Right Leaf

7- Insert 69 , 71 ⇒

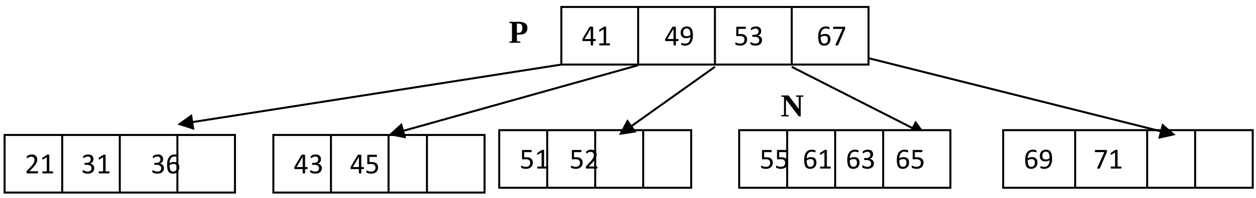




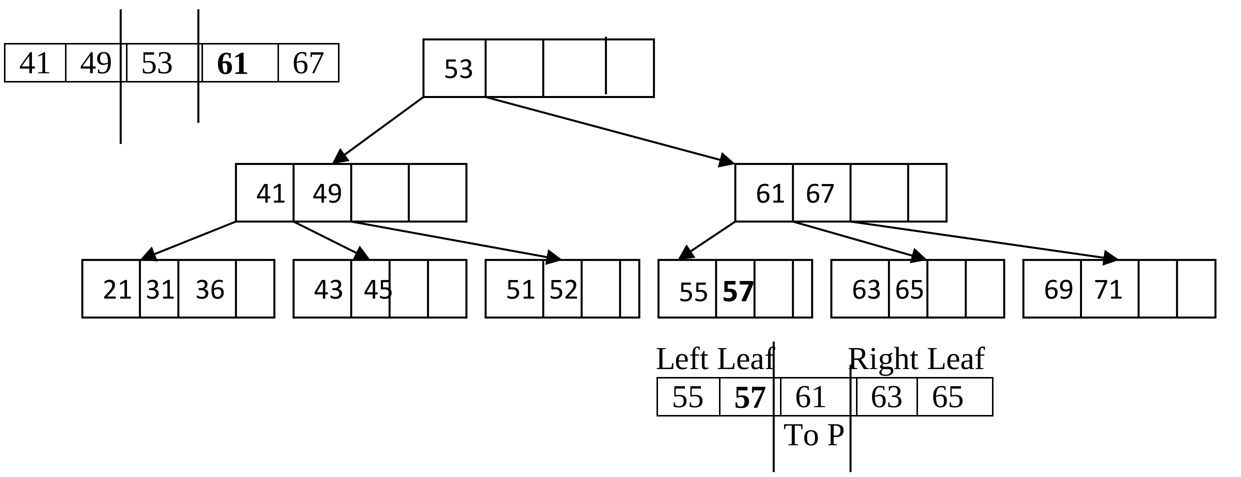
8- Insert 63 , 65 ⇒



9- Insert 57 into following tree :



⇒



Deletion from B-Tree of order n >= 2 :

- Deletion from B-tree is allowed only from a **leaf**.

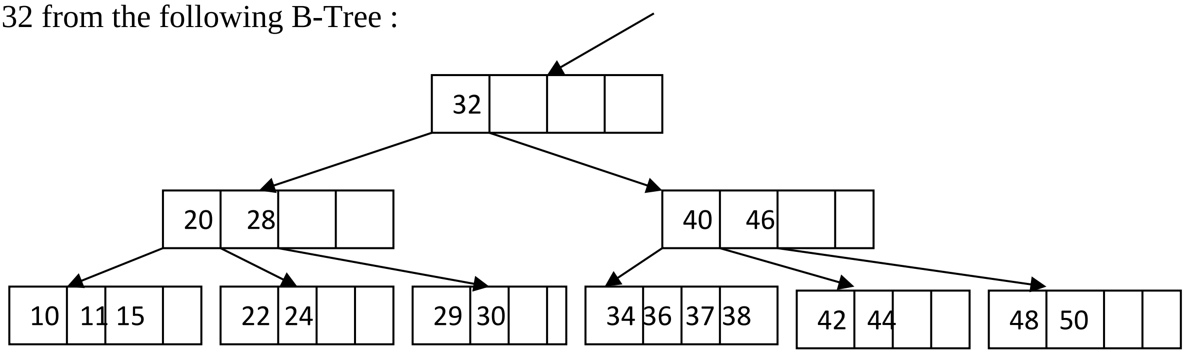
Otherwise : (replace the element **x** [which will be deleted] with the element **y** in the mostleft Node in the right subtree of **x**)

Means :

Suppose **P** defined as pointer to the Node **N** , where we wish to delete from with element **P.element[i]** , but **N** is **NON-leaf-Node** , so we search in **P.child[i+1]** (right child of **P.element[i]**) to find the least element in the leftmost leaf, replace this element with **P.element[i]** and then restructuring the B-Tree after replacing using the followingalgorithm (as Example)

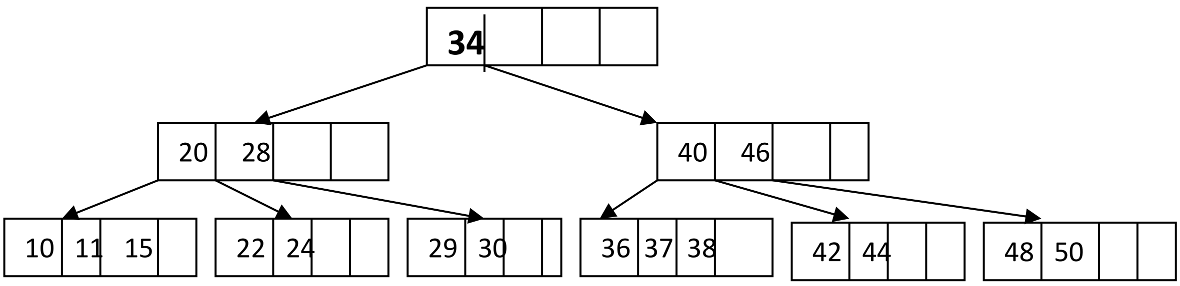
Example (in general for NON-leaf-Node) :

Delete 32 from the following B-Tree :



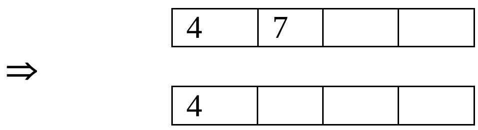
Solution :

Replacing x = 32 with y = 34 ⇒ the B-Tree becomes



Deletion algorithm : (Example B-Tree of Order n = 5)

1- N is Leaf and Root contains **1** element ⇒ after deletion B-Tree is **NULL**
or contains more than 1 elements ⇒ ex. Delete 7



2- Delete from **N** , **N** is leaf ⇒

A- **N** has > **q = (n-1) DIV 2 = (5-1)DIV 2 = 2** elements ⇒ No problem

Example delete **34** from following B-Tree :

33	34	37	38

⇒

33	37	38	

B- **N** with exactly **q = (n-1) DIV 2** = 2 elements ⇒

Suppose **P.child[i]** points to **N**

P.child[i-1] left sibling of **N**

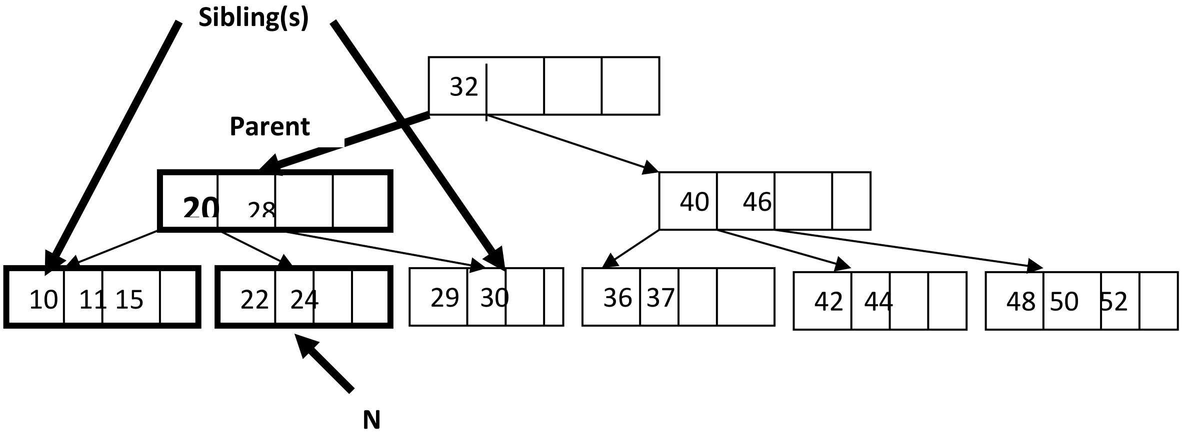
P.child[i+1] right sibling of **N**

B.1- Left or Right or (BOTH) sibling(s) with elements > **q = (n-1) DIV 2** ⇒

Choose the one of the siblings which has more than **q** elements merge it with the rest of **N** and the element which defined as parent of **N** and then restructuring as follows :

Make temp array likes insertion algorithm contaning the chosen Elements with X_1, X_2, ……, X_q, X_{q+1}, X_{q+2},….

Example : (Delete 22 and 42)



RESTRUCTURING ⇒

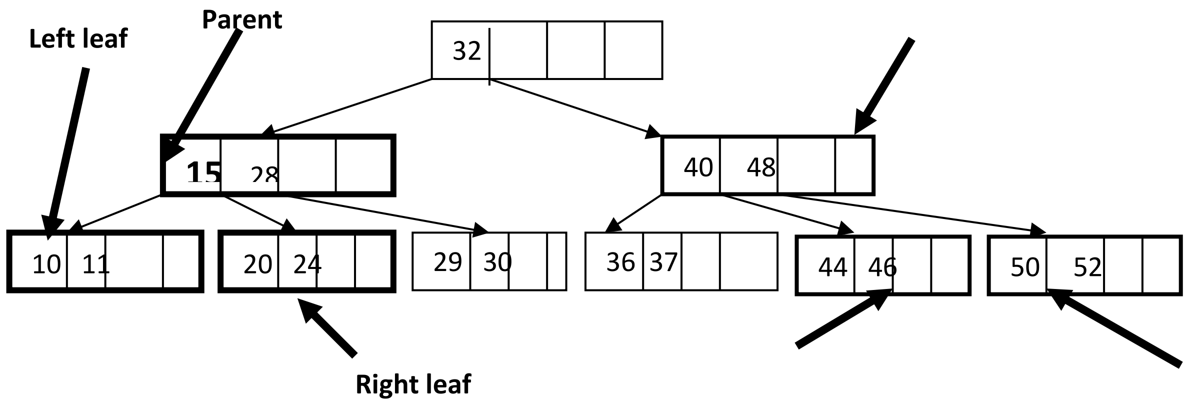
Explaining the case of delete **22** :

- Merge 10 , 11 , 15 , 20 , 24 temp. and sort in increasing order

10	11	15	20	24

- Splitting temp in two leaves left with elements (from X_1 to $\mathbf{X_q}$ and right from $\mathbf{X_{q+2}}$ to the last element then transferring X_{q+1} to parent P of **N**)

10	11	15	20	24
Left leaf		to P	right leaf	



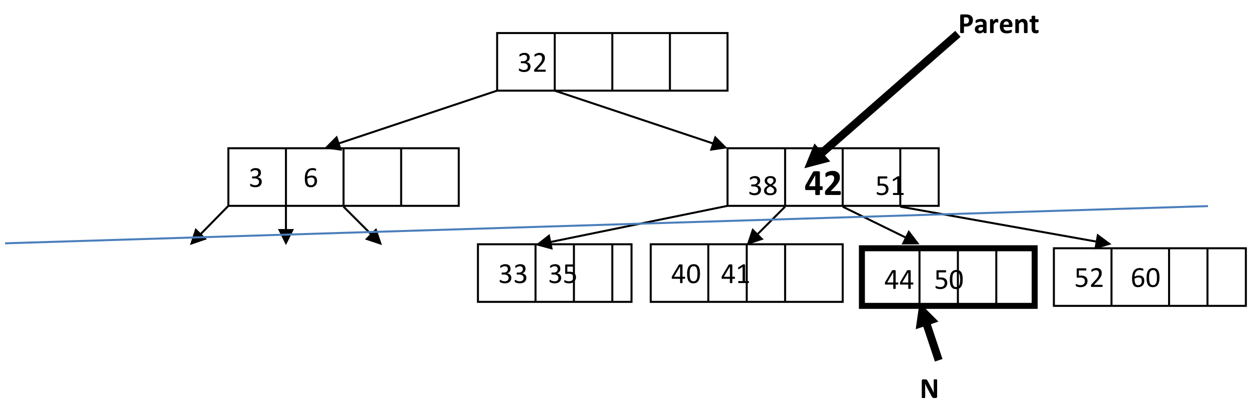
B.2- Sibling(s) with exactly **q** = (n-1) DIV 2 elements ⇒
Form new Node **N'** as **<u>leaf</u>** with the rest **(q-1)** elements from **N** and **q** elements from the chosen sibling and the element from common parent of (**N** and it's sibling) , then restructuring the B-tree likes following example

(There are many cases to suppose):

Example1 : (Delete 44 from the following B-Tree)

B.2.1- Choose left sibling of **N** (**Parent of N NON-Root with elements > q**):
(Recurive)

(HomeWork : Choose right sibling)

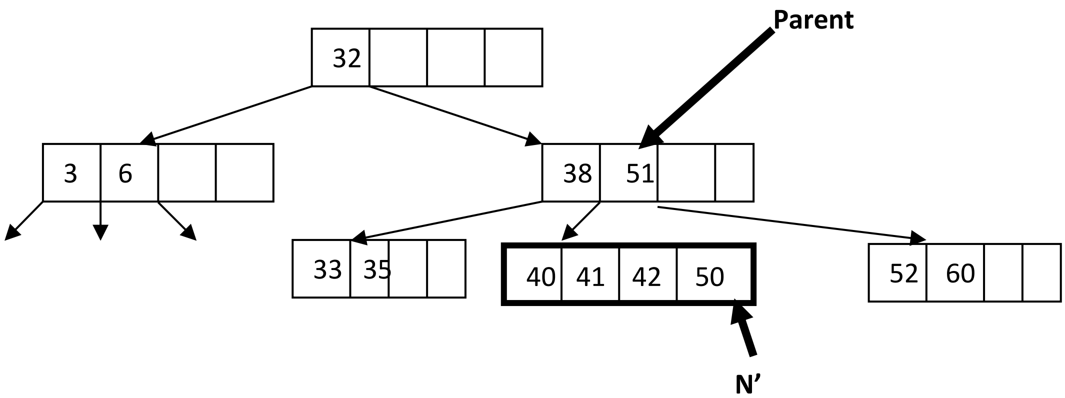


RESTRUCTURING ⇒

Merge 40 , 41 , 42 , 50 in new Node N' in increasing order :

40	41	42	50

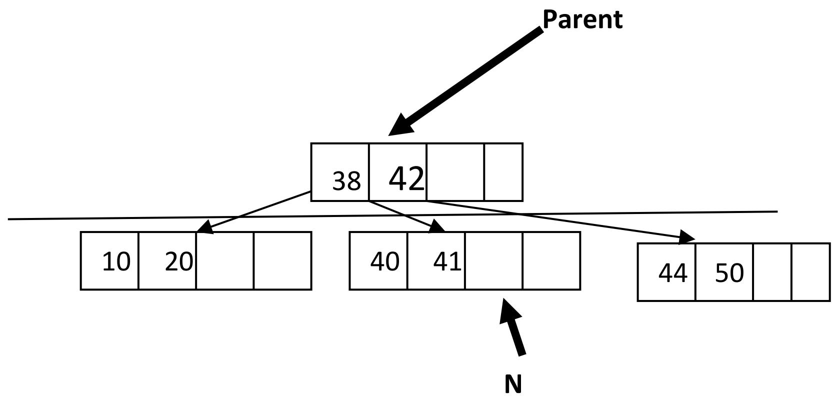
N' as new leaf



Example2 : (Delete 41 from the following B-Tree)

B.2.2- Choose right sibling of **N (Parent of N Root with elements >1 elements)**:

(Recursive) ⇒ restructuring same as in **B.2.1**

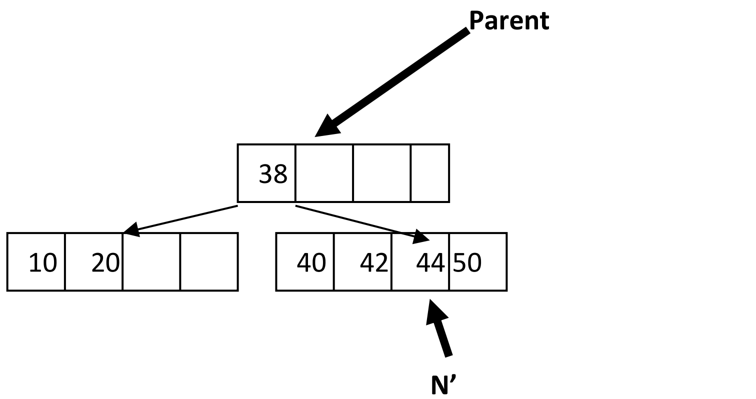


RESTRUCTURING ⇒

Merge 40 , 42 , 44 , 50 in new Node N' in increasing order :

40	42	44	50

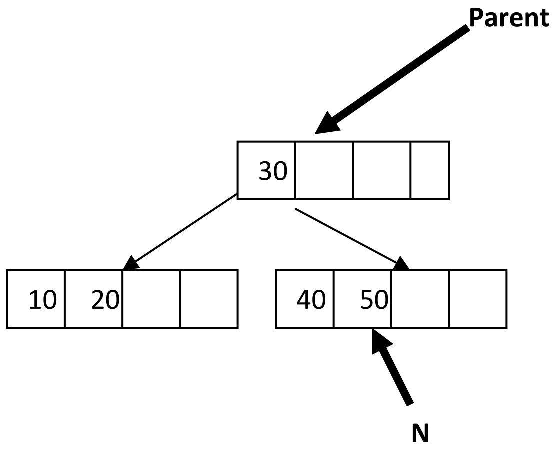
N' as new leaf



Example3 : (Delete 40 from the following B-Tree)

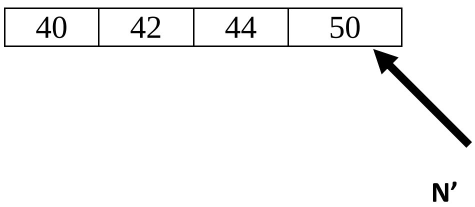
B.2.3- Choose left [there is only left sibling] of **N**

(Parent of N Root with elements *__exactly__* 1 element) :



RESTRUCTURING ⇒

Merge 10 , 20 , 30 , 50 in new Node N’ as leaf in increasing order :



Example4 : (Delete 69 from the following B-Tree)

B.2.4- Choose left [there is only left sibling] of **N**

(Parent of **N** **NON**-Root with elements ***__exactly__*** **q** elements)

:

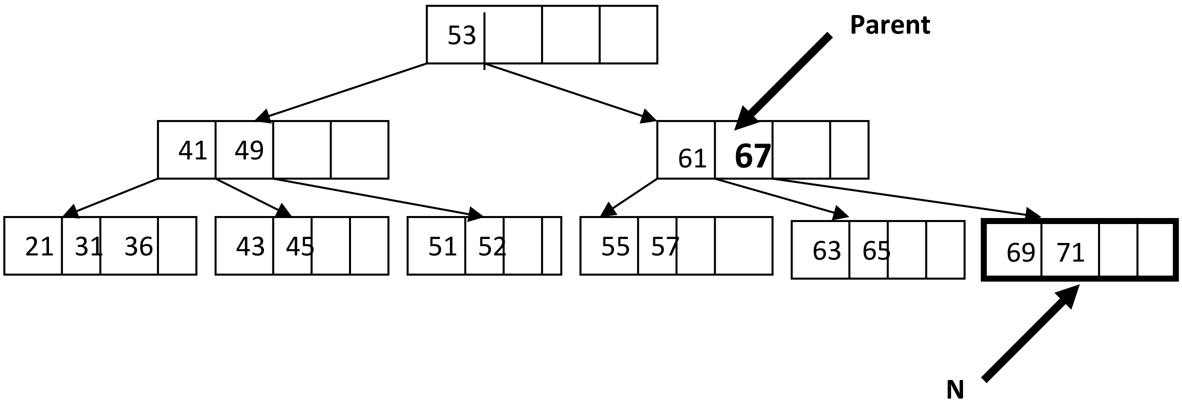
RESTRUCTURING ⇒

IF sibling of Parent with elements > **q** **THEN** using **B.1**

ELSE using **B.2** ***[RECURSIVELY]***

41	49	53	61

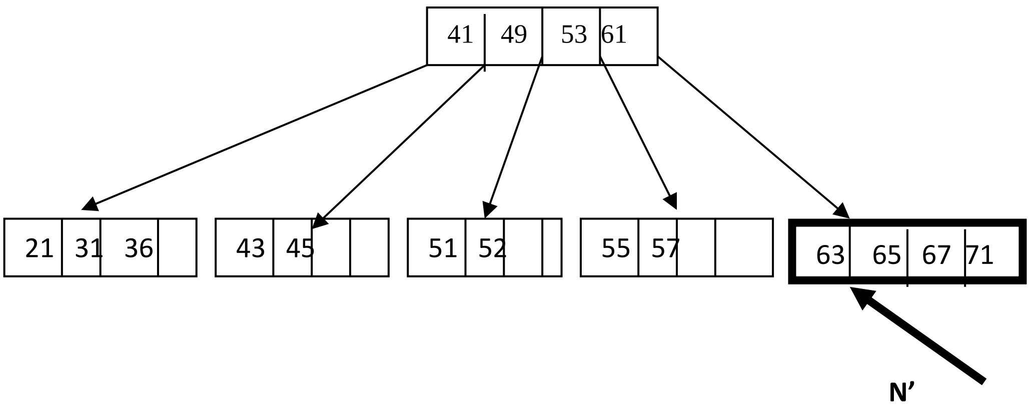
N' new leaf



63	65	67	71

N' as new leaf

RESTRUCTURING ⇒



B*-TREE OF ORDER n :

B*-tree of order **n>=2** is M.W.S.T. of order **n>=2** such that :

1- The root node may contain between **1** and **2((2n-2) DIV 3)** elements
2- Each **NON**-root-node contains between **(2n-2) DIV 3** and (**n-1**) elements
3- All leaves in the same level

Insertion in B*-TREE of order n>=2 :

Idea : Insert into leaf N

Algorithm's idea :

IF (N is not full) ⇒ insert into N in increasing order ;
ELSE (N is full) ⇒
{
IF (N is root) ⇒
{
- Make one dim temp array containing keys of N and the new key in increasing order ;
- Make left leaf with $X_1, \ldots, X_q$ and right leaf with $X_{q+2}, \ldots$, insert X_{q+1} into New Node as parent. (where q = (2n-2)DIV3).
}
ELSE (N is NON-root) ⇒
{
- Looking for sibling(s) of N (left and right)
- IF(sibling(s) not full)⇒choose left or right sibling, then left shifting or right shifting ;
ELSE (sibling(s) full) ⇒
{ - Make temp array length **2n** containing keys of N,the chosen left or right sibling, the common key of parent, and the inserted key in increasing order ;
- Make 3 leaves as follows :
}
}
}

(2n-2) DIV 3 keys to left leaf	Next key to parent (recursively)	(2n-1) DIV 3 keys to mid leaf	Next key to parent (recursively)	2n DIV 3 keys to right leaf
Left Leaf	to Parent P	Mid Leaf	to the same Parent P	Right Leaf

Example :

Suppose we have a B*-Tree of order **n = 7** ⇒

- Root-Node contains between **1** and **8** elements [between 1 and 2(2n-2)DIV3)]
- **NON**-Root-Node contains between **4** and **6** [between (2n-2)DIV3 and (n-1)]

Insert 50 in the following B*-Tree of order **n = 7** :

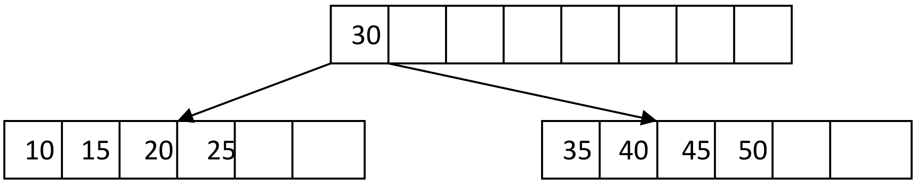
10	15	20	25	30	35	40	45
1	2	3	4	5	6	7	8

⇒ dim array

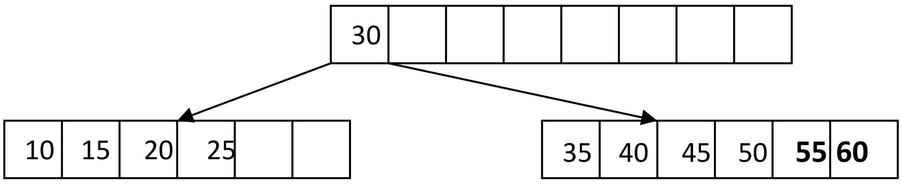
10	15	20	25	30	35	40	45	**50**
1	2	3	4	5	6	7	8	9

RESTRUCTURING ⇒

- Construct ***left*** Node with the first **4** elements
- Construct a Node as ***Root*** with the next element
- Construct ***right*** Node with the last **4** elements

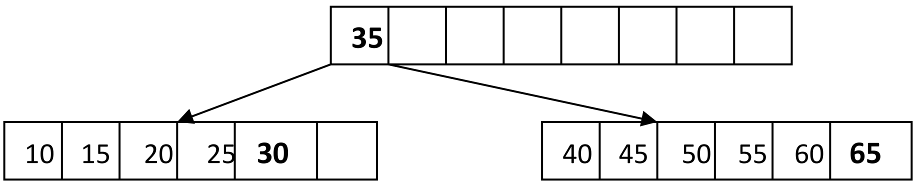


Insert 55 and 60 ⇒

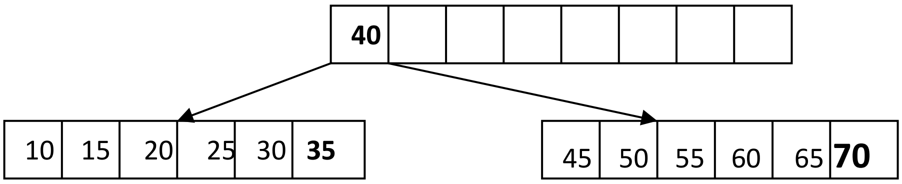


Insert 65 and 70 ⇒ looking for sibling (left), not full, make left shifting as following :

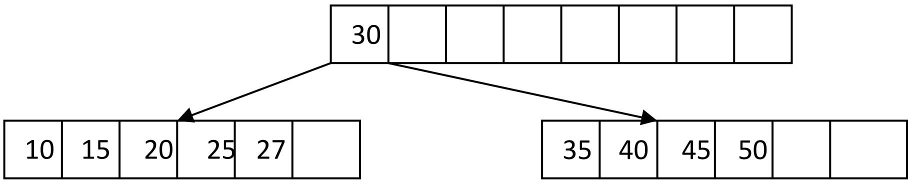
65



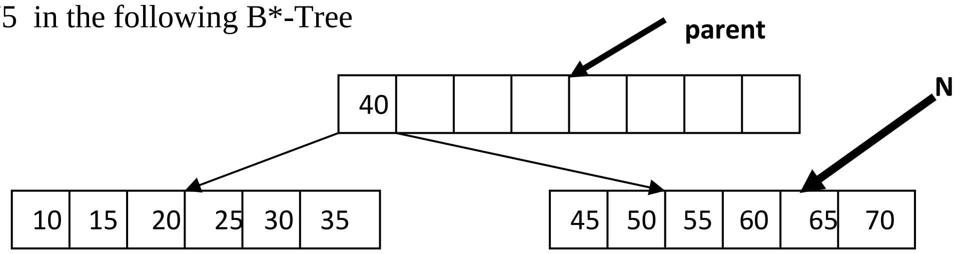
70



HOMEWORK AT HOME : INSERT first 29 and then 9



Insert 75 in the following B*-Tree



⇒

N is NON-Root-Node (N and it's sibling is full)

⇒ restructuring as follows :

- Separate the elements (after insertion) in N and sibling (which chosen) and parent
- of N ⇒ we have 2n elements
- Create 3 new leaves with following :

First leaf with (2n-2) Div 3 elements (as left) , the next element to parent P

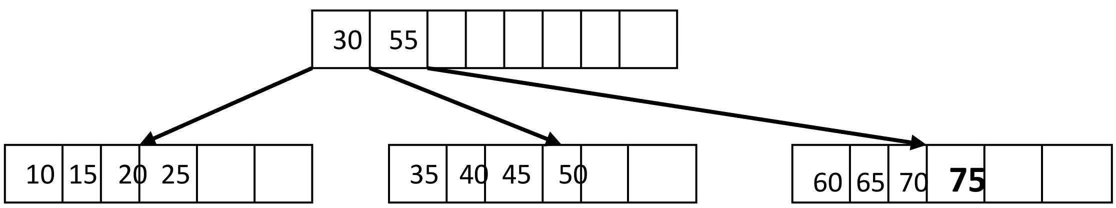
Second leaf with (2n-1) DIV 3 elements (as mid leaf),the next element to parent P

Third leaf with 2n DIV 3 elements (as right)

(2n-2) DIV 3 = 4 | (2n-1) DIV 3 = 4 | 2n DIV 3 = 4

10	15	20	25	30	35	40	45	50	55	60	65	70	75

⇒



Example2 :

Insert the key with the value 33 into the following B^*-Tree of order n = 5 :