April 8

1. **Bubble Sort :-** $T(n) = n-1 + T(n-1)$

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

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

$\therefore O(n^2)$

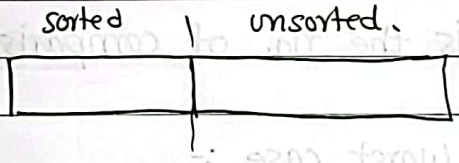
Selection sort :-

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

$$= 1 + 2 + \dots (n-1)$$

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

Insertion Sort :-



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

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

$$\Rightarrow O(n^2)$$

In best case its complexity $O(n)$

2 Linear Search :-

If the number is not present in the sequence, then the algorithm does n comparison

In worst case, the complexity of the algorithm is O(n)

In Average case = $\frac{\sum \text{all cases}}{\text{Total no. of cases}}$

$$= \frac{1+2+3+\cdots+n}{n}$$

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

Binary Search :-

T(n) is the no. of comparisons done

In worst case :-

$$T(n) \leq 2 + T(n/2)$$

$$\leq 2 \log n$$

$$\therefore O(\log n)$$

3 Merge Sort :-

$T(n)$ be the no. of comparisons done

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

$2T(n/2)$ ↑ dividing the array

$2n$ → merge function iterates.

$$= 2n \log n$$

$\therefore O(n \log n)$ → in worst case also

The drawback of merge sort is that it requires one additional array of size n. (ie. extra space)

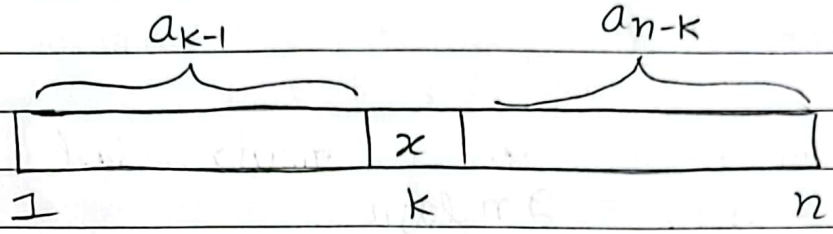
APRIL - 10

1 Randomised Quick sort :-

See book

2 Analysis of RQS :-

Let a_n be the average time token to sort an array of size n.



$$a_n = a_{k-1} + a_{n-k} + (n+1)$$

↳ For selecting pivot

For Av. case, we have.

$$\frac{\Sigma \text{ all cases}}{\text{Total cases}} = \frac{\sum_{k=1}^{n} (a_{k-1} + a_{n-k} + (n+1))}{n}$$

Now $$\sum_{k=1}^{n} a_{k-1} = a_0 + a_1 + \dots + a_{n-1}$$

$$\sum_{k=1}^{n} a_{n-k} = a_{n-1} + a_{n-2} + \dots + a_0$$

$$\sum_{k=1}^{n} (n+1) = n(n+1)$$

So, finally we get

$$n\,a_n = 2.(a_0 + a_1 + \dots + a_{n-1}) + n(n-1) \quad \text{----} (1)$$

Also replacing $n \to n-1$ in orginal eq^n

$$(n-1)\,a_{n-1} = 2.(a_0 + a_1 + \dots + a_{n-2}) + (n-1)n \quad \text{--} (2)$$

From eq (1) & (2) eliminating $2.(a_0 + a_1 \dots a_{n-2})$ we get.

$$n\,a_n = (n+1)\,a_{n-1} + 2n$$

Divide it by $n(n+1)$ both sides.

$$\frac{a_n}{n+1} = \frac{a_{n-1}}{n} + \frac{2}{n+1}$$

Now finding the pattern, we get

$$\frac{a_n}{n+1} = \frac{a_0}{1} + 2.\left(\frac{1}{2} + \frac{1}{3} + \dots + \frac{1}{n+1}\right)$$

$$a_n = 2(n+1)\,H(n+1) + O(n)$$

$H(n) = \sum \frac{1}{k}$ is the Harmonic series.

Now $\sum \frac{1}{k} \leq \int \frac{1}{x} = \log x$

$$a_n \leq 2(n+1)\log(n+1) + O(n)$$

$\therefore$ complexity $= O(n \log n)$ → in avg. case

⇒ This algorithm does not use extra space.

⇒ We should choose median as a pivot to run quick sort in $n \log n$ time in worst case also.

3 Randomised Algo & Rank of Element :-

To run quick sort algorithm in $n \log n$ time in worst case also, we have to choose median element as pivot element so that array can be partitioned into 2 equal halves.

So median = element with rank $n/2$

Rank of an element :-

Rank of X is defined as (No. of elements $< X$) + 1

Rank of X can be found in linear time

⇒ rank = 1

```
for (i = 0; i<n ; i++)
        if (A[i] < X)  rank++ ;
```

Now, lets look at an interesting problem

Q Find an element whose rank is $< n/2 + 1$ i.e. A number bigger than median

$\Rightarrow$ $n/2+1$ is the rank of the median

Algo :-

Pick a random p between 0 to n-1, what is the probability that A[p] is smaller than the median?

See, there will be exactly half of nos. greater than median and exactly half of nos. less than median. So, A[p] smaller than the median, its probability is $1/2$.

Now we repeat the above experiment k no. of times.

i.e. Pick- k numbers from a given sequence of numbers, what is the probability that all of them are smaller than the median?

So, probability that ALL OF THEM are smaller than the median is $= \frac{1}{2} \times \frac{1}{2} \cdots \times \frac{1}{2}$ k times.

$$P = \frac{1}{2^k}$$

So, now saying the some thing differently :-

When we pick K numbers from the given sequence of numbers, then probability that atleast one of them will be greater than the median is equal to $\left(1 - \frac{1}{2^K}\right)$. (Obviously that no. will be the max of K nos.)

⇒ So, now the algorithm is simple.

We pick K numbers from the given sequence of numbers and report the maximum of them.

then with probability $\left(1 - \frac{1}{2^K}\right)$ that reported no. will be greater than the median.

Now, in above sentence replace $K \rightarrow \log n$

⇒ Pick $\log n$ numbers from the given sequence of numbers and report the maximum of them.

then with probability $\left(1 - \frac{1}{n}\right)$ the reported number will be greater than median.

If $n \rightarrow$ large ⇒ Prob. is very high

So, the above algorithm is a randomised algorithm which does only $\log n$ comparisons and return no. greater than median with very high probability.

So complexity $O(\log n)$

⇒ Any deterministic algorithm will require atleast $n/4$ comparisons.

Randomised Algorithm :-

(i) LAS-VEGAS ALGORITHM :- Always give correct answer.

Running time depends on the input and the random choices the algorithm makes.

Ex ① Randomised Quick Sort
② Randomised Find Rank
③ Hashing

(ii) MONTE-CARLO ALGORITHM :- Are efficient but the answers are wrong with very low probability

Ex ① Algorithm to find element bigger than median
② Primarly Testing

Randomised Find Rank

Designing a Find Rank (rank) function which will return i such that Rank(A[i]) is rank.

⇒ RFindRank code in lms

$T(n)$ is the no. of comparisons done by the RFindRank Algorithm

$$T(n) = n + \underset{\uparrow \text{for partition}}{} \text{Max}_{0 \le k \le n-1} \{ T(k), T(n-k) \}$$

In worst case

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

In the case when partition function divides the elements into 2 equal groups (i.e when pivot is the median)

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

$$= n + n/2 + n/4 - -.$$

$$\le 2n$$

$T(n)$ is $O(n)$

Now the question is that how we pick such a good pivot ??

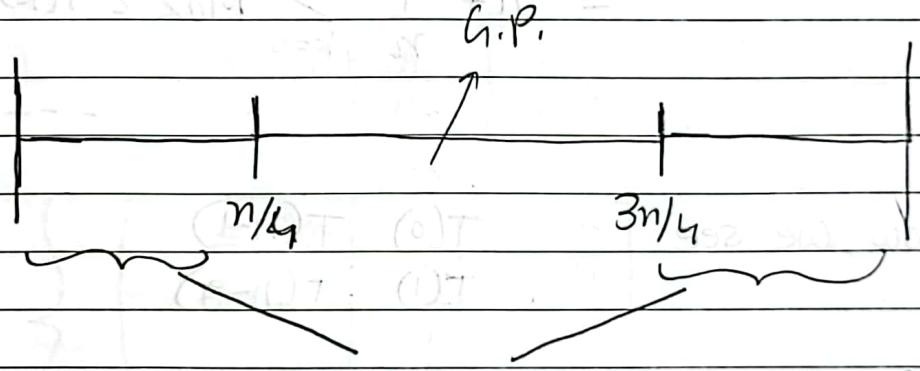
Good Pivot :-

Good Pivot is a pivot if the rank of pivot is in between $n/4$ and $3n/4$

⇒ If we pick a random pivot, the probability that it is good pivot is $1/2$.

On average, in every alternate iteration we shall pick a good pivot.

⇒ If we pick a good pivot, we shall discard at least $n/4$ numbers.



One set among them will definitely be discarded after partition function.

Hence we will be left with at most $3n/4$ numbers for next iteration.

Hence in this case $T(n) \leq 2n + T(3n/4)$

↑ We have to do 2 iteration to find good Pivot

$= 8n$

$O(n)$

- **Calculation for Av. case complexity of RFindRank :-**

We know

$$T(n) \leq n + Max\{T(k), T(n-k-1)\}$$

$$0 \leq k \leq n-1$$

$$\text{Av. case complexity} = \frac{\Sigma \text{ all cases}}{\text{Total cases}}$$

$$\leq \frac{\sum_{k=0}^{n-1} n + Max\{(T(k), T(n-k-1)\}}{n}$$

$$\leq n + \frac{1}{n} \sum_{k=0}^{n-1} Max\{T(k), T(n-k-1)\} \quad \text{---} (1)$$

Now, we see

$T(0)$	$T(n-1)$	$0 \leq k \leq n/2$
$T(1)$	$T(n-2)$	
⋮		
$T(n/2)$	$T(n/2)$	
⋮		$n/2 \leq k \leq n-1$
$T(n-2)$	$T(1)$	
$T(n-1)$	$T(0)$	

They are symmetrical

$\therefore$ Now eq^n ① becomes

$$\leq n + \frac{2}{n} \sum_{k=n/2}^{n-1} T(k)$$

Now we will basically show that $T(n) < cn$ $c \to$ constant for large values of n (i.e. for $n > n_0$)

Proof is by induction

we assume $T(k) < ck \quad k < n$

So $$T(n) \leq n + \frac{2c}{n} \left[\sum_{k=1}^{n-1} k - \sum_{k=1}^{n/2-1} k \right]$$

$$\leq n + \frac{2c}{n} \left[\frac{n(n-1)}{2} - \frac{n/2(n/2-1)}{2} \right]$$

$$\leq n + c \left[n - \frac{n}{4} \right]$$

$$\leq \boxed{n + \frac{3nc}{4} < cn} \rightarrow \text{we find values of } c$$

$$\boxed{c \geq 4}$$

$\therefore T(n) < 4n \quad \therefore$ complexity $O(n)$

15 APRIL

1. Count Sort :- Sort acc. to key value (i.e. largest value in the array is taken as key)

Algo :-

```
Count (a, n, K)                    K → key
{   int count [K+1] , b[n] ;
for(i=0 ; i<=K ; i++)
{ count[i]=0;
}
    for (i=0 ; i<n ; i++)
    {   ++ count [a[i]] ;
    }
    for (i=1 ; i<=K ; i++)
    {
        count [i] = count[i] + count [i-1] ;
    }
    for ( i=n-1 ; i>=0 ; i--)
    {
        b [-- count [a[i]]] = a[i] ;
    }
```

```
for (i=0 ; i<n ; i++)
{
    a[i] = b[i] ;
}
```

Complexity → Only 5 for loops

$\therefore O(n+k)$

Drawbacks :-

(i) Extra space required

(ii) Nos. should be between 0 to K+1. Negative & floating point arn't allowed.

2 Radix / Bucket Sort :-

Radix → start from L.S.B

Bucket → start from M.S.B.

Radix :-

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

Use count sort in all intermediate steps

Let k be the no. of digits of the largest no. present in the array.

Code :-

```
j = 1

while (-- k > 0)                    m → largest digit (key)
{   i → 0 to m      count[0] = 0

    i → 0 to n      count[(A[i]/j) % 10] ++

    i → 1 to m      count[i] += count[i-1] ;

    i → n-1 to 0    b[-- count[(A[i]/j) % 10]] = A[i]

    i → 0 to n      A[i] = b[i]

    j = j x 10 ;
}
```

Complexity :- $O(k) \times O(n)$

See bucket sort from book

20 APRIL

1 Fibonacci number modulo 10 USING RECURSION

Problem statement :-

$F(0) = 0 \qquad F(1) = 1$

$F(n) = (F(n-1) + F(n-2)) \% 10$

Write an efficient algorithm that will give F(n)

ALGORITHM 1 :- Using Recursion

```
int rec (int n)
{   if (n<2)
        return (n);
    else
        return ((rec(n-1) + rec(n-2)) %. 10)
}
```

Lets see how many operations we are doing in this func.

① comparing $n<2$ ③ computing $n-2$
② computing $n-1$ ④ addition ⑤ modulus

So, we are doing here 5 operations.

Let $T(n)$ = number of basic operations req. to find $F(n)$

$$T(n) = T(n-1) + T(n-2) + 5$$

$$\Rightarrow \quad < T(n-1) + T(n-1)$$

$(\because \text{as } T(n-1) = T(n-2) + T(n-3) + 5 > T(n-2) + 5)$

$$\Rightarrow \quad < 2T(n-1)$$

$$\Rightarrow \quad < 2^2 T(n-2)$$

$$\Rightarrow \quad < 2^k T(n-k)$$

This will terminate when $n = k$, Hence

$$\Rightarrow \quad < 2^n$$

$$\therefore \boxed{T(n) = O(2^n)}$$

$T(100) < 2^{100} \approx 10^{30}$ operations.

But this is upper bound only. So, we are still not able to conclude that this algo is efficient or not

So, lets calculate lower bound

$$T(n) = T(n-1) + T(n-2) + 5$$

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

$$> 2T(n-2)$$

$$> 2^2 T(n-4)$$

$$\vdots$$

$$> 2^k T(n-2k)$$

$$> 2^{n/2}$$

$\boxed{T(n) \text{ is } \Omega(2^{n/2})}$

Space complexity :-

$S(n)$ = Amount of space required to compute $F(n)$

$$S(n) = \text{Max}(S(n-1), S(n-2)) + 1$$

↳ storing

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

$$= n \rightarrow \text{it is both upper \& lower bound}$$

$S(n)$ is $\Theta(n)$

Hence Algorithm 1 takes

⇒ Exponential time
⇒ Linear space

ALGORITHM 2 :- Using an array.

Lets take an array f[] which will store all the values of F(n) we are computing.

```
int dp(int n)
{
    f[0] = 0
    f[1] = 1

    for ( i = 2 ; i <= n ; i++)
    {
        f[n] = ((f[n-1] + f[n-2]) % 10) ;
    }
}
```

Hence Algorithm 2 takes

⇒ Linear time
⇒ Linear space (it takes a lot of space)
$n <= 10^6$ (because this is size of max array we can declare)

ALGORITHM 3 :-

Algorithm 2 stored all the values but for computation of F(n), we need to store only previous 2 values i.e. F(n-1) & F(n-2).

```
int linear ( int n)
{   a=0      b=1      c=n

    for (i=2 ; i<=n ; i++)
    {       c= (a+b) % 10 ;
            a=b ;
            b=c ;

    }

 return c

}
```

Algorithm 3 takes n can be 10^9 also here

⇒ Linear time
⇒ Constant space

3 Can we compute $F(10^{1000})$ fast.

PERIODIC SEQUENCE :-

When some terms repeat over and over.

$a_1, a_2, a_3 \dots$

if satisfying $a_{n+p} = a_n$, then its period is p.

Now, F(n) is a periodic sequence with period 60.

$F(60) = 0 = F(0)$ & $F(61) = 1 = F(1)$

V.Imp

The moment any 2 consecutive nos. repeat, the whole sequence starts repeating because 3rd no. depends only on previous 2 nos. and so on.

Formal Proof that such sequence has to be periodic :-

Pigeon Hole Principle :-

If A and B are two finite sets such that $|A| > |B|$ then there are not be any one-one function from A to B.

Now F(n) can take only values from 0 to 9

So, if you take 11 numbers from the sequence, then all the numbers can not be distinct. Any one no. must repeat.

But we have to show here that two consecutive nos. repeat, so, we take pair.

Consider the order pair $(F(n), F(n-1))$, there can be at most 100 distinct order pairs. (10×10 possibilities)

So, if you take 102 numbers from the sequence, then there are i & j such that $(F(i), F(i-1))$ is same as $(F(j), F(j-1))$.

Hence sequence is periodic.

But one more strong thing we know (No proof)

$$\Rightarrow F(n) = (F(n-1) + F(n-2)) \% m$$

is periodic sequence with period less than $6m$

(No proof of this)

4 So, Now we know that

$$F(n) = (F(n-1) + F(n-2)) \% 10$$

is a periodic sequence with period 60. We need to compute F[i] for i=0 to 60 and then afterwards

F[n] will be same as F[n%60]

If n is smaller than 2^{64}, then F[n] can be found in constant time. (because 64 bit computer we have)

Now, the question is can we find F(n) for a 1000 digit number.

How do we store a 1000 digit no.??

⇒ A K digit no. can be stored in an array.

Then decimal rep. of that no. will be

$$N = \sum A[i].10^i$$

For example, 123 can be stored as A[0] = 3, A[1] = 2 and A[2] = 1

Now, once we store the no. in array, we have to perform modules operation.

Computing modulo :-

```
p = 60
r = A[k-1]
i = k-2

while (i>=0)
{
    r = (r*10 + A[i]) % p
    i--;
}
```

Complexity :- O(K)

K → no. of digits

⇒ O(log N)

∵ K ≈ log N

So, F(n) can be computed in O(logn) time.

⇒ Now, We GENERALISE TO MODULO m (instead of 10)

$$F(n) = (F(n-1) + F(n-2)) \% m$$

is periodic sequence with period less than 6m

Now, we find the period and then F[N] = F[N%P]

To Find the Period :-

See Algorithm 3, inside the for loop after we update our a & b, we check that

if a == F(0) && b == F(1), then the period is found.

Finding the period will take O(m) time.

and then computing modulo will take O(logn) time.

Hence, Finding F(n) can be done in total of

O(m + logn) time [For large n if n is stored in array]

5 We create a Power Function now first.

given X and n, How to compute X^n ??

```
Power (x, n)
{   if (n==0)
        return 1

    else if (n%2 == 0)
        return Power (x*x , n/2)

    else
        return x* Power (x*x, n/2)
}
```

Lets see why this is correct

if n is even , $n = 2k \Rightarrow n/2 = k$

$X^n = (X^2)^k = X^{2k}$

if n is odd , $n = 2k+1$

$X.(X^2)^k = X^{2k+1} = X^n$

Lets now see Time Complexity

$T(n)$ be the number of multiplications that the algorithm does to compute X^n

$T(n) = 1 + T(n/2)$ if n is even

$T(n) = 2 + T(n/2)$ if n is odd

$T(n) < 2 + T(n/2) < 4 + T(n/4) < 2\log n$

$T(n) > 1 + T(n/2) > 2 + T(n/4) > k + T(n/2^k)$

$> \log n$

We are substituting $k \to \log n$
here because it terminates once $T(0) = 1$ or $T(1) = 1$

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

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

$\log n < T(n) < 2\log n$

$\boxed{T(n) \text{ is } \Theta(\log n)}$ ($\Theta \to$ theta)

Now we look at iterative algorithm instead of recursion.

```
Power (X, n)
{    y = 1

  while (n>0)
  {    if (n%2 == 1)
            y = y * X

       X = X * X

       n = n/2
  }

  return y
}
```

This is $O(\log n)$ algorithm

Now lets see if n is in binary.

$n = \sum b_i 2^i$ is the binary rep. of n

b_i can be either 1 or 0.

So $\Rightarrow$ $X^n = X^{\sum b_i 2^i} = \prod X^{b_i 2^i}$ $(a^{b+c} = a^b \cdot a^c)$

↑ product

if $b_i = 0$, then i^{th} term is 1
if $b_i = 1$, then i^{th} term is X to the power 2^i.

In previous prgm, checking $n \% 2$ is odd is similar to checking whether b_i is 1.

Now, Lets look at Power function to compute

$A^n \rightarrow$ where A is a 2×2 matrix

```
Power (A, n)
{   Y = 1 → Identity matrix
    while (n > 0)
    {   if (n % 2 == 1)   Y = Mult (Y, X)
        X = MULT (X, X)        will work in
        n = n/2                constant time
    }
    return Y ;          X & Y are matrices.
}
```

This is $O(\log n)$ algorithm

But how to find A^n, when N is a 1000 digit number ??

If N is given in Binary representation, say in array B.

```
Power (A, B)
{  Y = 1      i = 0

   while (i < k)                    k → length of array B
   {  if (B[i] == 1)                (basically no. of digits in
            Y = Mult(Y, X);          binary rep. of n)

      X = Mult (X, X)

      i++
   }
   retun Y
}
```

This is $O(\log n)$ algorithm

Just change while loop to

$y = y * (x \text{ digit})$;

$x = x \wedge 10$

What if N is given in Decimal representation ??

⇒ Don't convert decimal to binary

instead we have to make 10 if conditions like we made only 1 in case of binary.

6 Now we come back to our problem of computing F(n)

$$F(n) = (F(n-1) + F(n-2)) \bmod m$$

We also know that A^n can be computed in O(logn) time, where A is a 2×2 matrix and n is a 1000 digit no.

$$\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix} \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

$$= A \begin{bmatrix} F(n) \\ F(n-1) \end{bmatrix}$$

$$= A^2 \begin{bmatrix} F(n-1) \\ F(n-2) \end{bmatrix}$$

$$= A^n \begin{bmatrix} F(1) \\ F(0) \end{bmatrix}$$

Hence $\begin{bmatrix} F(n+1) \\ F(n) \end{bmatrix} = \begin{bmatrix} a & b \\ c & d \end{bmatrix} \begin{bmatrix} 1 \\ 0 \end{bmatrix}$

where $\begin{bmatrix} a & b \\ c & d \end{bmatrix} = \begin{bmatrix} 1 & 1 \\ 1 & 0 \end{bmatrix}^n$

Finally c will give us $F(n)$

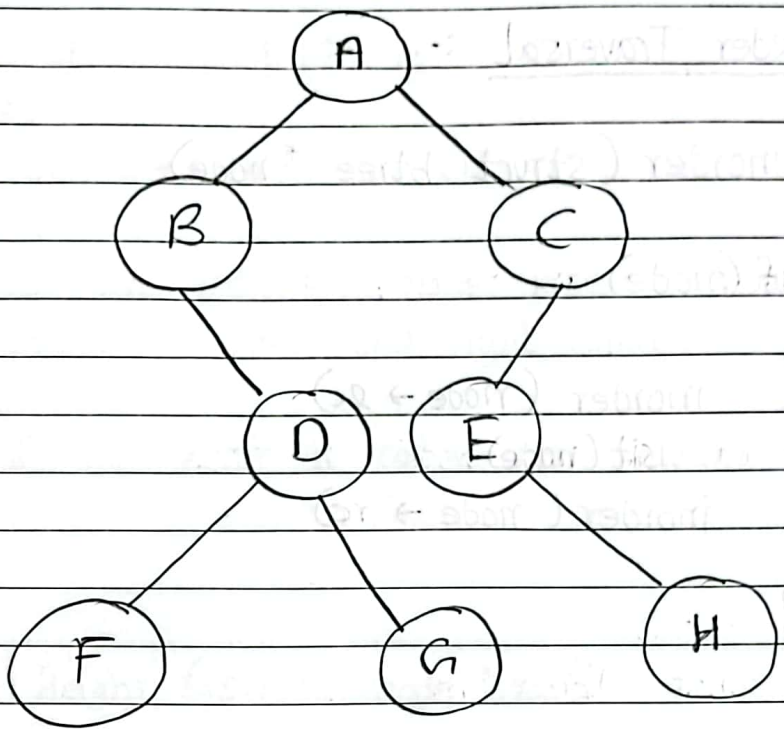
Note ⇒ As we are computing modulo, so while we calculate A^n in every iteration, we will ensure that the nos. inside the matrix is modulo m.

All multiplications had to modulo m.

22 APRIL Code-tantra

BINARY TREES :-

```
struct btree {

    int data;
    struct btree *lc, *rc, *p; };
```

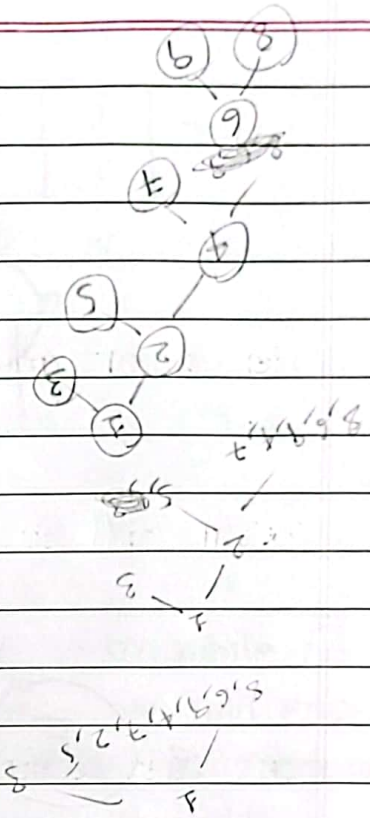


Pre-Order Traversal :-

```
void preorder (struct btree *node)
{   if (*node)
    {   visit(node)
        preorder (node → lc)
        preorder (node → rc)
    }
}
```

In Order Traversal :-

```
void inorder (struct btree *node)
{   if (node)
    {   inorder (node → lc)
        visit (node)
        inorder ( node → rc)
    }
}
```



Post Order Traversal :-

```
void postorder (struct btree *node)
{   if (node)
    {   postorder (node → lc)
        postorder (node → rc)
        vsit(node)
    }
}
```

NOTE :- Given single traversal ~~of length n~~.

$n(1) = 1$, $n(2) = 2$, $n(3) = 5$ & $n(4) = 14$

$n(r)$ → No. of trees possible if no. of nodes is r

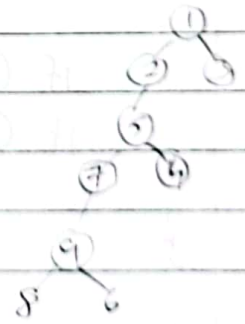
- **HEIGHT OF A NODE :-**

Height of leaf node is 0

Height of an internal node is +1 maximum of height of left child and right child.

Height of root node is called the height of binary tree.

```
int getHeight (struct node * root)
{   if (root == NULL)
        return -1;

    else {

        left_sub = getHeight (root → left)
        right_sub = getHeight (root → right)

        if (left_sub > right_sub)
            return (left_sub + 1)
        else
            return (right_sub + 1)

    }
}
```



- **LEVEL OF A NODE :-**

Level of the root node is 0

Level of any non root node is 1+ level of the parent node

Height of the root node is same as the maximum level of a node in the binary tree.

- **LEVEL ORDER TRAVERSAL :-**

Visit all the nodes of smaller level before visiting the nodes of higher level. Nodes in the same level are visited from left to right.

```
Enqueue (Q, root)
while (Q) {

    node = Dequeue(Q);
    visit(node);

  if (node → lc)   Enqueue (Q, node → lc)
  if (node → rc)   Enqueue (Q, node → rc)

}
```

We can implement Pre order traversal using stack.

⇒ Just replace Enque by push & Deque by pop in previous L.O.T. program.

We can also implement Post order traversal using 2 stacks.

⇒ S1 & S2 be 2 stacks

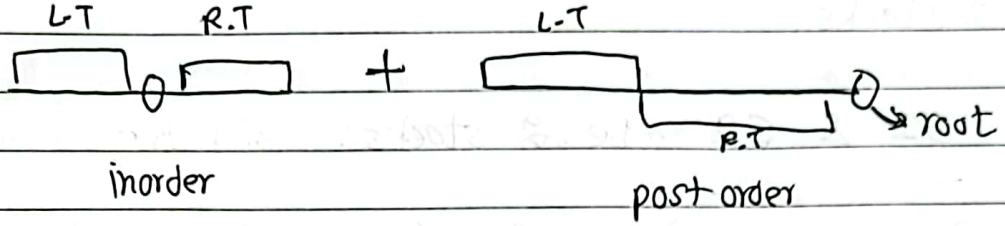
Post order

```
Push (S1, root) ; Push (S2, true)
while (S1) {
  node = Pop(S1) ; flog = pop(S2)
  if (flog) {
    Push (S1, node) ; Push (S2, false);
    if (node → rc) { Push (S1, node → rc) ; Push (S2, true) }
    if (node → lc) { Push (S1, node → lc) ; Push (S2, true) }
  }
  else visit (node)
}
```

Exchange them for Inorder

Now given to us

- in order + post order → Unique tree
- in order + pre order → Unique tree
- pre order + post order → Not unique tree



In first 2 cases, nodes on left side & right side are fixed.

TERMINOLOGIES :-

⇒ A full binary tree / Proper binary tree in which every node other than the leaves has two children.

⇒ A perfect binary tree is a full binary tree in which all leaves are at the same level.

⇒ In a complete binary tree, every level (except possibly the last) is completely filled and all nodes are as far left as possible.

⇒ A degenerate tree is where each parent node has only one associated child node.

THEOREM :-

I The number of leaf nodes is less than or equal to the number of internal nodes. + 1.

i.e. $L <= I + 1$ (For any binary tree)

Proof :- By induction on no. of nodes in binary tree (n)

if $n = 1$

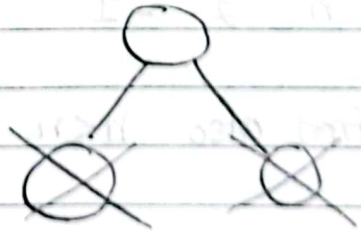
then no. of internal nodes = 0
no. of leaf nodes = 1

As $1 <= 0 + 1$, so this is true for $n = 1$

Now to proove it for n nodes, we assume that this theorem is applicable on any tree with (n-1) nodes. We call it strong induction.

CASE-I :- If given binary tree with n nodes, we go to one of the leaf node and see if it has a sibling. if Yes, then we delete that 2 nodes.

We left with (n-2) nodes & above th^{m} is applicable for (n-2) no. of nodes



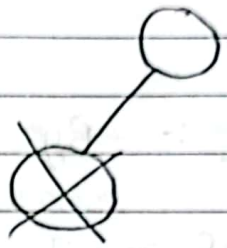
$L' = L - 1$ and $I' = I - 1$

↳ 2 leaf node deleted & 1 internal node become leaf

$L' <= I' + I$

- $L <= I$

Case-2 :- If No, then we delete that 1 leaf node only.



- $L' = L$ and $I' = I - 1$
- $L' <= I' + 1$
- $L <= I + 1$

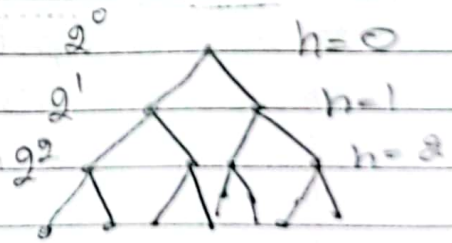
II In any binary tree

$$n+1 <= 2^{h+1}$$

$h \rightarrow$ height

Equality holds in case of perfect binary tree

$$n = \sum 2^i$$



$$n = 2^{h+1} - 1$$

And also $h < n$

But this eq^n / th^m only gives upper bound.

Now, the question is if h is given, we have to design a binary tree with min no. of nodes.

Answer ⇒ We can make a degenerate tree

i.e.

So, the min. no. of n is equal to h+1 (if h starts from 0)

III From I and II

$$n+1 <= 2^{h+1}$$

$$2L <= I + L + 1 <= 2^{h+1}$$

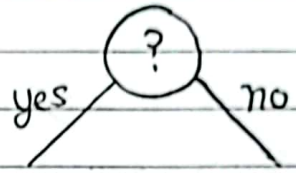
$$\boxed{L <= 2^h}$$

27 APRIL.

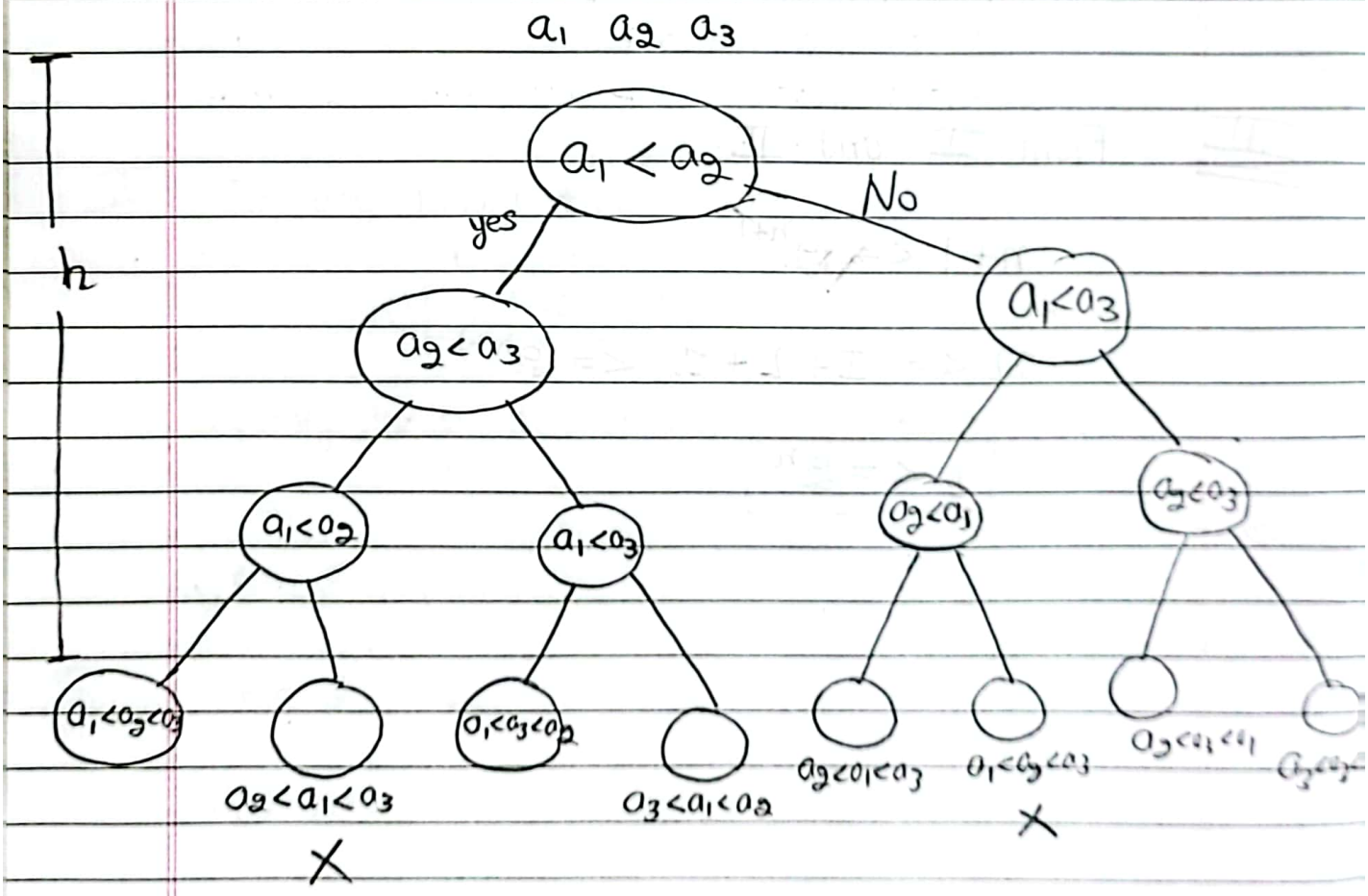
1 Lower Bound for Sorting.

Any comparison based sorting algorithm would require atleast nlogn no. of comparisons. (i.e. $\Omega(n \log n)$)

Decision Trees :-



Lets implement decision tree in bubble sort algorithm



X → We cannot reach these leaf nodes

So, we start from the root node and we get sorted sequence only when we reach leaf node

So, the no. of comparison done by sorting algorithm is exactly equal to height of tree.

Now we know that

$$L <= 2^h$$

$$\Rightarrow \boxed{h \geq \log L}$$

Another thing to observe here is that if we have a sequence of n nos., we can have at max n! no. of diff. permutations.

So in every permutation, we will follow different path along the tree

Hence No. of leaf nodes in decision tree should be atleast n!

$$\boxed{L \geq n!}$$

(L → No. of leaf nodes in decision tree)

So, summary

The no. of comparisons done by algorithm in worst case

$= $ height of decision tree

$\geq \log L$

$\geq \log n!$ --- ①

Now we deal with $\log n!$

We know $n! = \prod_{i=1}^{n} i \qquad i < n$

$< n^n$

$\Rightarrow \log n! \leq n \log n$ --- ② (Upper bound)

Also, we know $n! = \prod_{i=1}^{n} i$

$> \prod_{i=n/2}^{n} i$

$> \prod_{i=n/2}^{n} n/2 \qquad (\because i \geq n/2)$

$> (n/2)^{n/2}$

$\Rightarrow \log n! > n/2 \log n/2$ --③ (lower bound)

So, eq^n 1 becomes

$\geq \log n!$

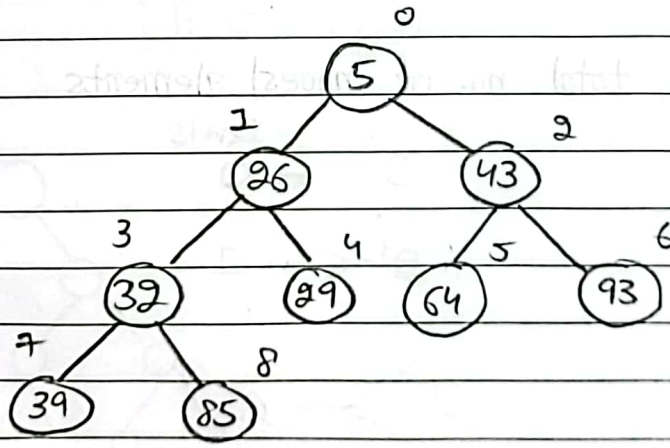
$\geq \frac{n}{2} \log \frac{n}{2}$

$\Rightarrow \Omega(n \log n)$ Hence prooved.

29 APRIL

1 Min (BINEARY HEAP) :-

Binary Heap is a complete binary tree in which each node is associated with a key, at each node of the tree, value of the key is smaller than or equal to the value of key of both of its children. (called Heap property)



while popping, we bosically do the level order traversal

Now this tree can be represented in the form of array.

H[] =	5	26	43	32	29	64	93	39	85
	0	1	2	3	4	5	6	7	8

We will maintain an array instead of tree

V.V.Imp If we want to go from one node to its parent node, we have to do $\left(\frac{i-1}{2}\right)$ ($i \rightarrow$ index)

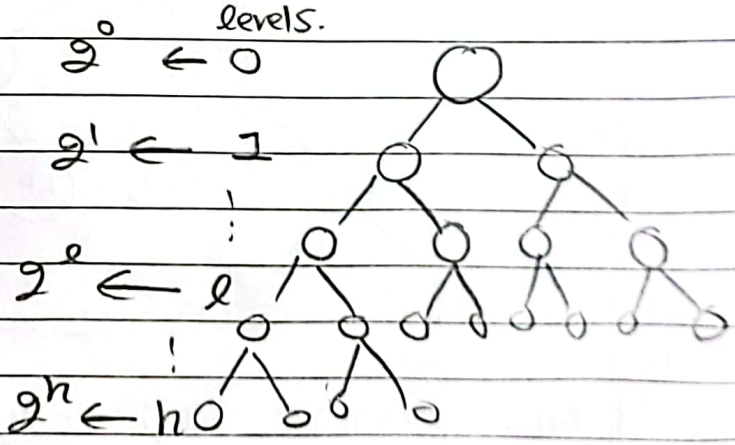
Node to its left child $\rightarrow$ $2i+1$
Node to its right child $\rightarrow$ $2i+2$

THEOREM :-

Height of a Binary Heap with n numbers is $\theta(\log n)$

where $n \rightarrow$ total no. of nodes/ elements

Proof :-



Now $n > \sum_{i=0}^{h-1} 2^i$ (exclude the nodes at last level)

$n > 2^h - 1$ (lower bound of n)

$\Rightarrow$ $\boxed{h < \log(n+1)}$

We can also get upper bound of n

$n \leq \sum_{i=0}^{h} 2^i$ (completely fill the last level)

$n \leq 2^{h+1} - 1$

$\Rightarrow$ $\boxed{h \geq \log(n+1) - 1}$

So, we get $\log(n+1) - 1 \leq h \leq \log(n+1)$

$\boxed{h \text{ is } \Theta(\log n)}$

2 Now we consider the previous heap

Now we perform **insert** and **delete** operation on our heap.

INSERT :- We insert the any new element, we place that element at last in order to satisfy property of complete Binary tree

Now we build a Bottom Up heapify function in order to satisfy Heap property after insertion.

Let the element to be inserted be X.

$\theta(\log n)$

```
void Add (int H[], int X, int *n) {
    H[*n] = X ;    → inserting at last
    *n = *n + 1 ;
    BottomUpheapify (H, *n-1) ;
}
```

Note n → passed to void Add function is the total no. of elements

```
void BottomUpheapify (int H[], int i) {
    p = (i-1)/2          p storing index of parent node

    while (p > -1) {
        if (H[i] < H[p]) {
            Swap (H[i], H[p])
            i = p
            p = (i-1)/2   }
        else  p = -1 ;  }
}
```

DELETE MIN :- Min will be present at root (i.e. index 0).

To satisfy the property of complete binary tree, we remove the root node and set the last node as root node.

Now we build a TopDown heapify function in order to satisfy heap property.

```
void DeleteMin (int H[] , int *n) {
    *n = *n - 1
    Swap (H[n] to root)    // Discard the root
    TopDownheapify (H, 0, *n) ; }
```

$\Theta(\log n)$

```
void TopDownheapify (H[] , i, n) {
    while (2i+2 < n)    checking if node has valid right child.
    {
        if (H[2*i+1] < H[2i+2])
            c = 2i+1
        else  c = 2i+2

        if (H[i] > H[c]) {
            Swap (H[i], H[c]) ;
            i = c ;
        }
        else  i = n ; }
```

$\log(n)$ time

Now the corner case is that if node has only left child. (But then we will be doing one last comparison only because we have reached at end)

```
c = 2i+1
if (c<n && H[i] > H[c]) {
    Swap (H[i], H[c])
} }
```

Decrease key :- This is basically updating any node. (Changing to smaller value)

```
void Decrease key (H[], i, X) {
    H[i] = X;
    Bottom Up Heapify (H, i);
}
```

$\Theta(\log n)$

changing to greater value

```
void Increase key (H[], i, X) {
    H[i] = X;
    Top Down heapify (H, i);
}
```

$\Theta(\log n)$

3 HEAP SORT

First we see how to Build a Heap given a sequence of n numbers.

Method-1 :- Suppose the n numbers are stored in array A and we have to make heap (i.e array H)

```
for i→0 to n
        Add(H[], A[i], i)
```

→ $O(n \log n)$

↳ it contains BottomUp heapify

Method-2 :- Now we fill the array H reversely (i.e. build the heap from bottom)

So atleast first $n/2$ elements will just settle at the leaf nodes (∵ there are atleast $n/2$ leaf nodes)

So, we start from $n/2$ element

```
i = n/2
while (i>=0)
{ TopDownheapify (H, i--); }
```

Now we have to proove its complexity

Imp Note :- In a heap, if a node is at height i then calling TopDownheapify, then no. of comparisons done is at most i

So, now when we are at height say i, no. of comparisons done for each node is i & no. of nodes at height i is less than equal to $\frac{n}{2^i}$

$$\therefore T(n) \leq \sum_{i=0}^{h} i \times \frac{n}{2^i}$$

$$\leq n \sum_{i=0}^{n} i/2^i$$

↳ A.G.P.

$$\leq 2n$$

$$\therefore \boxed{T(n) \text{ is } O(n)}$$

Heap Sort :- {

BuildHeap (H[], n) {

$\Theta(n \log n)$

m = n
for (i=0 ; i<n : i++) {
DeleteMin (H, &m)
}

}

The following can be done in $\Theta(\log n)$

- BottomUp heapify
- TopDown heapify
- Insert
- DeleteMin
- DecreaseKey
- IncreaseKey
- Delete - DecreaseKey + Delete min

- Min → To return the min element, can be answered in $O(1)$ time (just give the root)
- Build Heap in linear time
- Heap sort in $\Theta(n \log n)$ time

4 MAY

1. Suppose we want to implement the following operations.

- Add ()
- Delete Median ()

⇒ Median is an element with rank $\left(\frac{n}{2}+1\right)$.

Method 1 :-

- Add() → Just add at the end O(1)
- Delete Mdian → Using RFind Rank O(n)

To delete a single no. it takes O(n) time, so to empty or to do n Delete median operations, the whole complexity will become $O(n \times n) = O(n^2)$

Method 2 :- Both operations Add() & Delete median will be done in O(logn).

Two HEAPS :- We will maintain 2 heaps

- One will have all the median and the nos. smaller than the median
- Other heap will have all the elements bigger than the median.

For the first case → Let HS be the Max Heap and has lets say ns no. of elements.

For the seand case → Let HB be the MinHeap and has nb no. of elements.

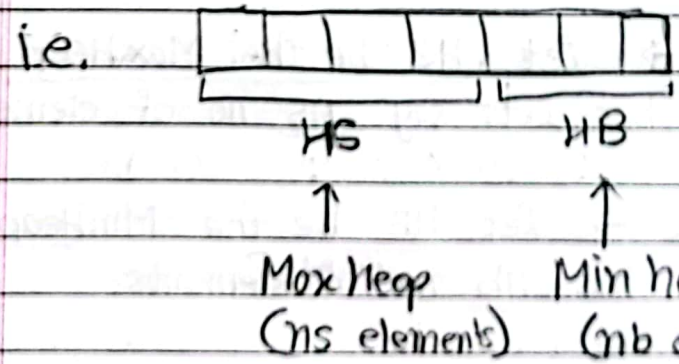
Note that either $ns = nb$ or $ns = nb + 1$

↑ even ↑ odd

⇒ Median will always be stored in HS and as it is a MaxHeap, so when we have to give the median, we just pop the root node.

```
Delete Median () {

    median = HS[0]                          O(logn)

    Delete Max (HS, &ns) :

    if (nb>ns) {

        Delete Min (HB, &nb) :
        Add (HS, HB[nb], &ns) ; }

}
```

→ See this prgm through an array



→ Whole array is arranged in ascending order only

⇒ If (nb>ns) → Unstability in heaps.

```
Add (x) {                                          O(logn)
    if (x > HS[0]) {
        Add (HB, x, &nb);
        if (nb > ns) {
            DeleteMin (HB, &nb)
            Add (HS, HB[nb], &ns) } }
    else {
        Add (HS, x, &ns);
        if (ns > nb+1) {
            DeleteMax (HS, &ns)
            Add (HB, HS[ns], &nb) } }
```

They are different

⇒ DeleteMin is called only for min heap

⇒ DeleteMax is called only for max heap

⇒ The whole program performing n Add & n Delete Median operations, complexity becomes **O(n logn)** in Method 2

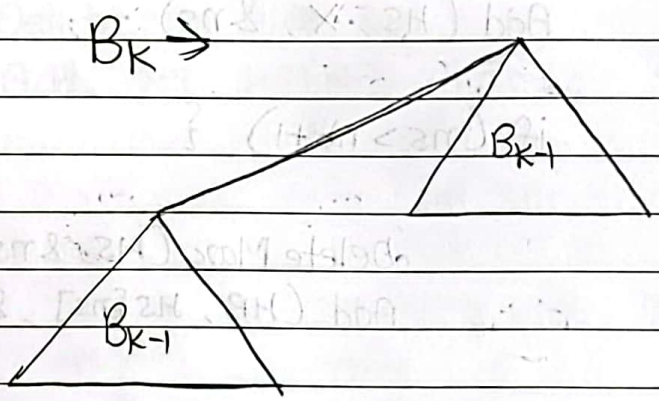
2 BINOMIAL HEAPS :-

BINOMIAL TREES :-

Binomial trees of order k is defined inductively and denoted by B_k.

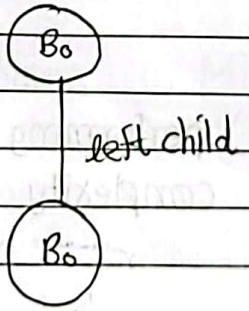
B_0 is just one node

B_k is constructed by taking two copies of B_{k-1} and making one as the **left child** of the other.

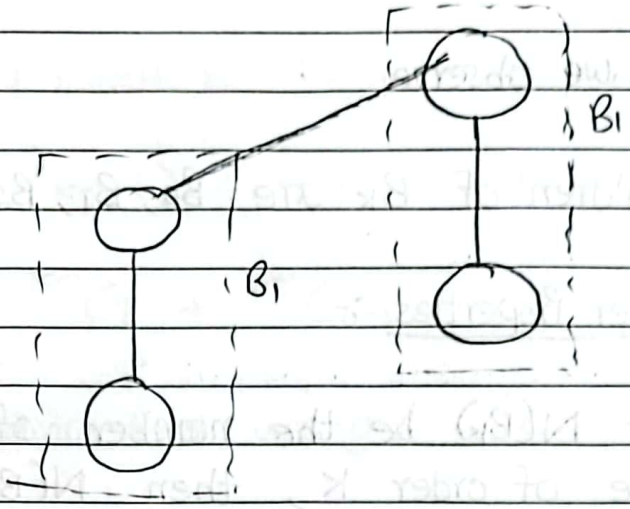


B_0 → ○ (single node)

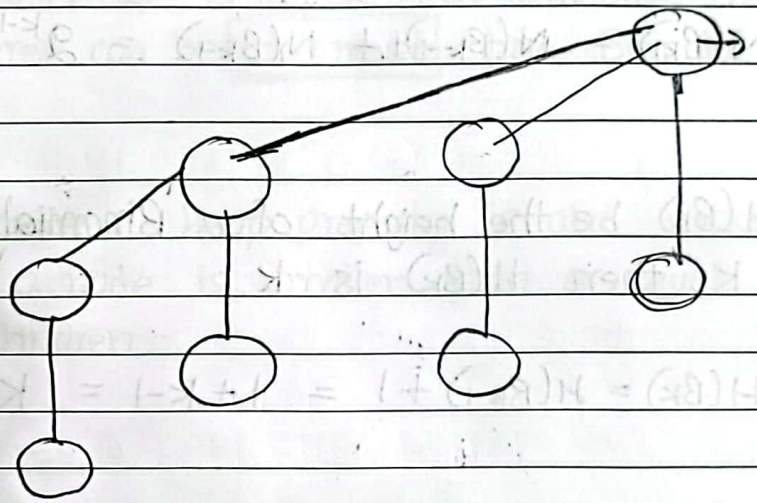
B_1 →



$B_2 \rightarrow$

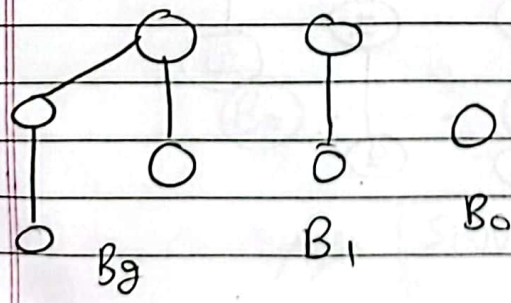


$B_3 \rightarrow$



Interesting property :- If I delete the root node in B_3, we also delete all the links through root node.

We are left with B_0, B_1 & B_2 (i.e. all the children of B_3)



We get exactly 1 copy of each child

So, we observe

Children of B_K are $B_0, B_1, B_2, \ldots, B_{K-1}$

Other Properties :-

⇒ If $N(B_K)$ be the number of nodes in a binomial tree of order K, then $N(B_K)$ is 2^K.

Proof is by induction.

$$N(B_K) = N(B_{K-1}) + N(B_{K-1}) = 2^{K-1} + 2^{K-1} = 2^K$$

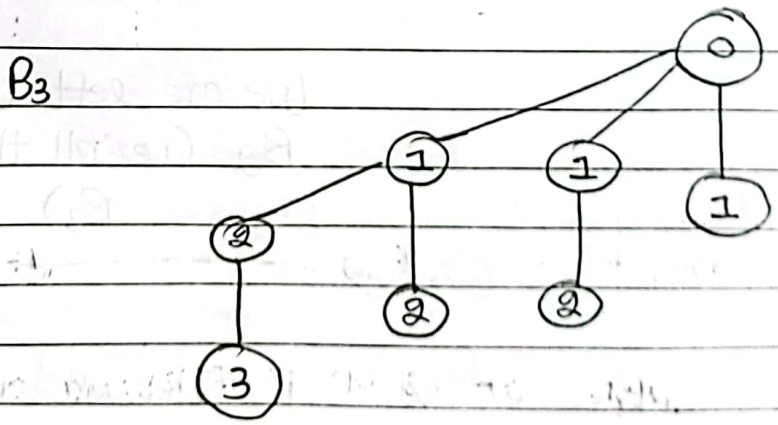
⇒ If $H(B_K)$ be the height of a Binomial Tree of order K, then $H(B_K)$ is K.

$$H(B_K) = H(B_{K-1}) + 1 = 1 + K - 1 = K \quad \text{(By induction)}$$

⇒ Let $L(K, i)$ be the number of nodes which are at some level i in B_K, then

$$L(k, i) = {}^{K}C_i$$

B_3



Proof is again by induction.

$$L(k,i) = L(k-1,i) + L(k-1,i-1)$$

$$= {}^{k-1}C_i + {}^{k-1}C_{i-1}$$

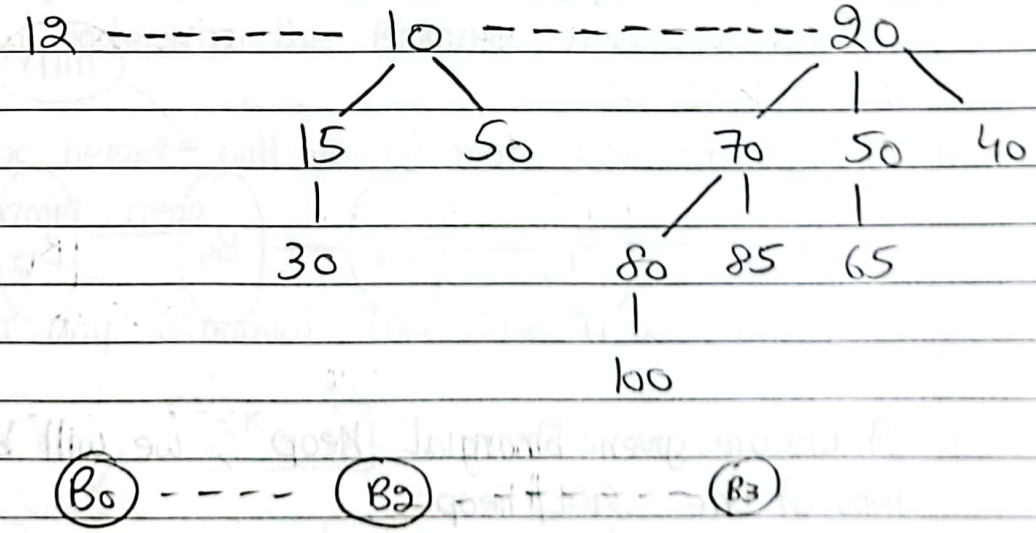
When we expand this, we get ${}^{k}C_i$

BINOMIAL HEAPS :-

A Binomial Heap is a set (collection) of Binomial Trees such that there can be **at most** one Binomial Tree of any degree

At each node, Heap property is also satisfied i.e. the value of a node is smaller than or equal to the value of its children.

Example :-



B_0 ---- B_2 ---- B_3

Should be in increasing order of B_K

Now Lets say we are given n no. of nodes. Can be make a binomial heap with n nodes ??.

The answer is yes.

We see binary representation of n and B_k will be present if and only if k^{th} bit of binary rep. of n is 1.

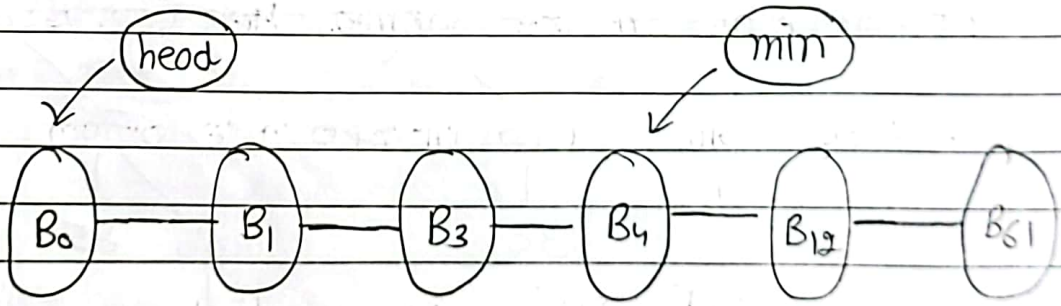
Let say n = 107

$$n = 1101011$$
$$6\,5\,4\,3\,2\,1\,0$$

So B_0 --- B_1 --- B_3 --- B_5 --- B_6

Structure of a Binomial Heap :-

All the roots of the trees are linked through a circular singly linked list - root list



If we are given binomial Heap, we will be given head of the list/ heap.

AA going from Left to Right → strictly inc. k of B_k

Min pointer will point to the node which has min value

Each node has

- Value
- Pointer to next node
- Parent pointer
- Child list
- No. of children of the node

⇒ The number of nodes in the root list is exactly equal to the number of 1's in the binary representation of n, which is $\Theta(\log n)$ ← atmost $\log n$

So, traversing the L.L. will take atmost $\log n$ time

⇒ The height of a Binomial Heap is the maximum height of any node in the Binomial Heap (i.e. L.L.)

∵ Max height will be of root node of any B_K in Binomial heap.

For any Binomial Tree (i.e. if we look at any B_K)

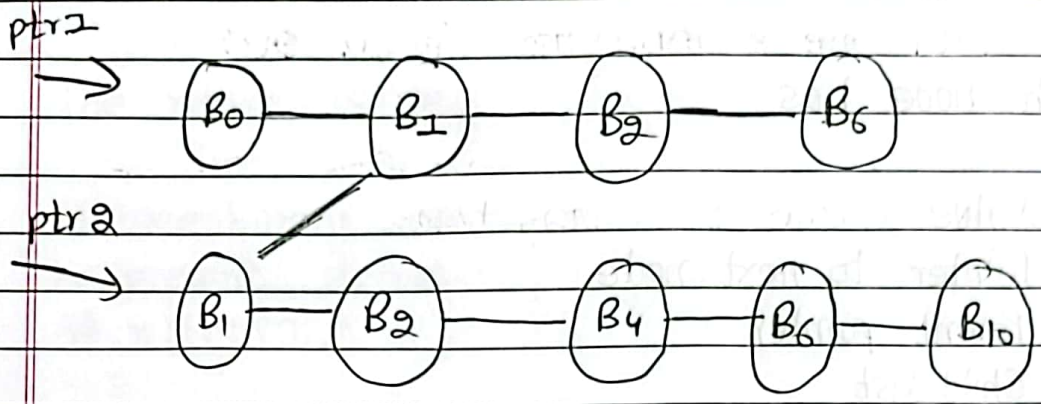
$2^K \leq n$ ($n \rightarrow$ Total no. of nodes in that B_K / Heap)

(For any Binomial Tree In Binomial Heap)

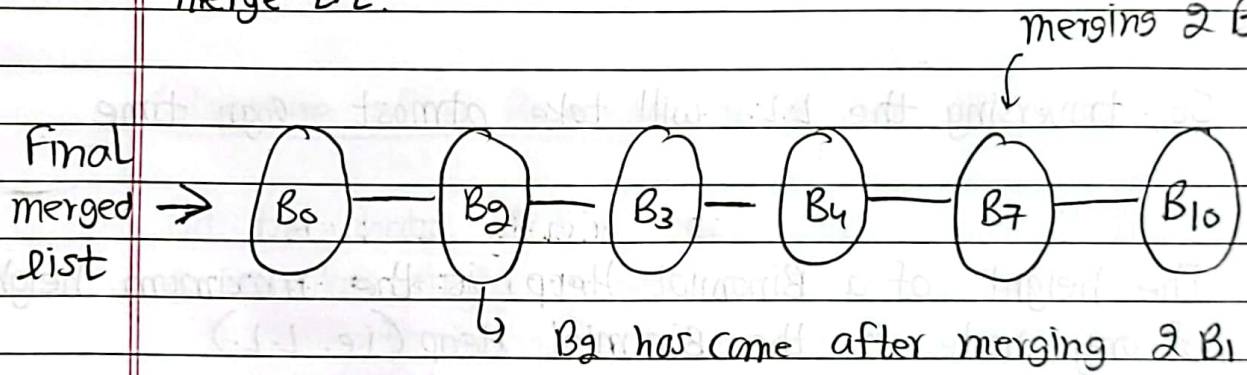
$K \leq \log n$

So the height of any tree can be atmost $\log n$

3 Union of Binomial Hopes :-



The process is similar to merge function. The only difference is that when ptr1 & ptr2 points to node with some volve of k (B_k), we merge both to make them B_{k+1} and then add that B_{k+1} to new merge L.L.



The process is also similar to adding two binary numbers.

```
  1 0 1 0 1 1 1   ← ptr1
+ 1 0 0 1 0 1 1   ← ptr2
-----------------
1 0 1 0 0 0 1 0   ← Merged L.L.
```

⇒ Union is $\theta(\log n)$ → The no. of comparisons = No. of Nodes in L.L. = $\log n$

So, we merged two Binomial heaps in $\Theta(\log n)$ time which we were not able to do in case of Binary Heaps. (in case of Binary heap, it was $O(n)$)

Rest all operations of Binomial heaps will take same time as in case of Binary heaps.

Add Operation :-

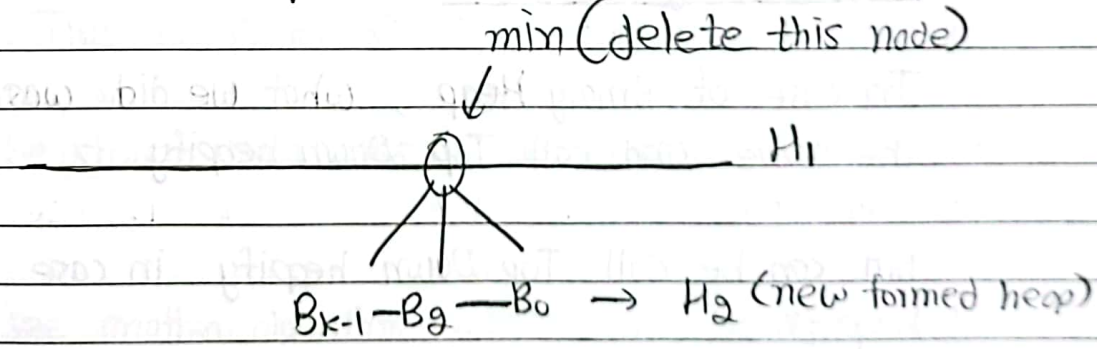
Create a new node (single node itself is a heap)

And merge this with the original heap.

Can be done in $\Theta(\log n)$

Delete Min Operation :-

We have a min pointer. Delete that node.



We are left with its children. So concatenate all the children, it forms a heap

Merge this heap with original (H_1) heap.

$\Theta(\log n)$

Decrease key Operation :-

Go to that node and decrease the value and call Bottom up heapify

$\Theta(\log n)$

Delete Operation :-

Now to delete any arbitary node, decrease the value to 1 less than the minimum and call delete min

Decrease key + Delete Min

$\Theta(\log n)$

Increase key Operation :-

In case of Binary Heap, what we did was we increase the value and call Top Down heapify.

But can be call Top Down heapify in case of Binomial Heap ??

Answer is yes but it will not be efficient

Because in case of Binary heap, we had atmost 2 children for each node, but in case of Binomial Heap, we can have atmost $\log n$ children for any node.

Thus no. of comparisons will increase.

So, it will take $\log n$ time to find the smaller valve child and then interchanging & setting will take another $\log n$ time.

$\therefore \Theta(\log^2 n)$

Efficient Way :-

Inc. key operation can be done in $\Theta(\log n)$ time by Deleting the node and then adding the node with the new valve

$\Theta(\log n)$

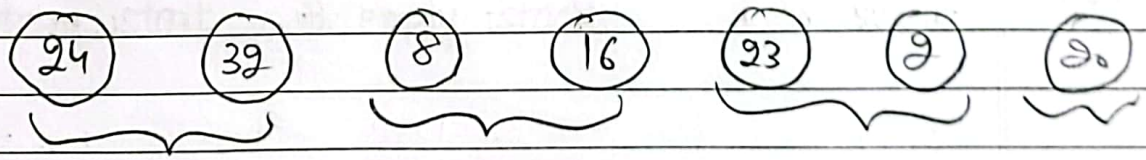
Build Heap Operation :-

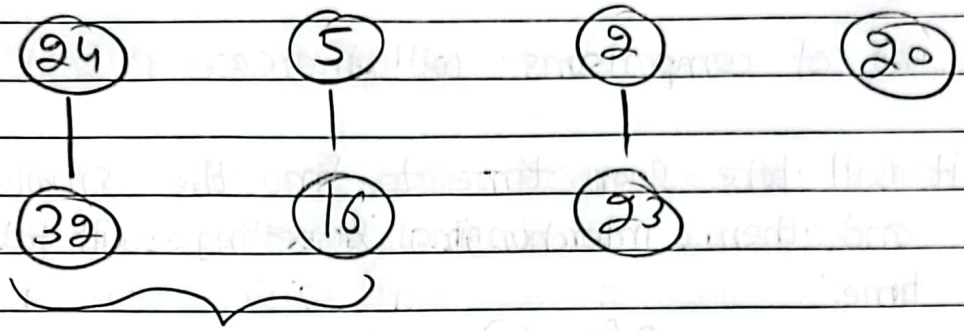
Add n numbers one by one

This is $O(n \log n)$

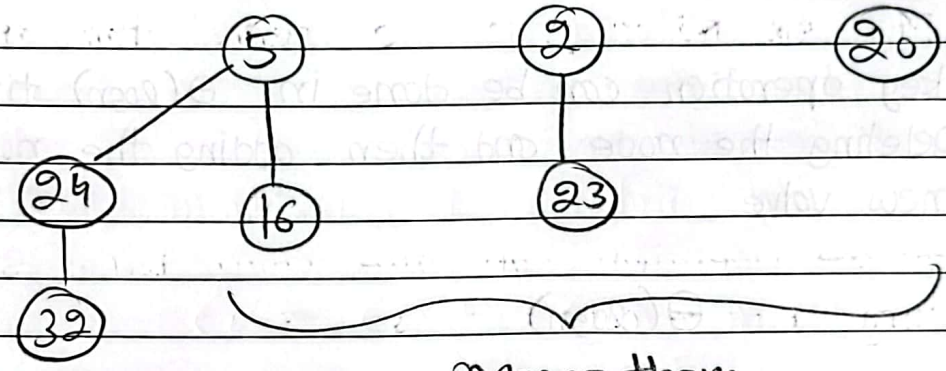
Can be shown that this is linear time (in Amortised Analysis)

Lets see another algorithm which build Heap in $\log n$ time.





Merge because some order.



Merge them

T(n) be total no. of merges done

$$T(n) = \frac{n}{2} + n/4 + n/8$$

is $\Theta(n)$

6 May AMORTIZED ANALYSIS

1 I have an algorithm which does one of the following in each iteration on a stack which was initially empty.

- Push
- Multipop (K) – pops K elements from the stack

⇒ Multipop is costly operation

One of the operation can take $O(n)$ time so the complexity of the algorithm is $O(n^2)$

But we will proove that its equal to $2n$.

Operation	Actual Cost	Amortized cost
Push	1	2
Multipop (K)	K	0

✓ No. of pushes ≥ No. of multipop (∵ $2n$ operations max)

Note → start with empty stack

POTENTIAL FUNCTION :-

$$\Phi : D \rightarrow R^{+}$$

(Data structure) (Positive Real no.)

Let amortized cost of i^{th} operation be denote by $\overline{C_i}$.

Let actual cost of i^{th} operation be denoted by C_i

Now, amortized cost is defined is.

$$\overline{C_i} = C_i + (\text{Change in Potential})$$

$$\overline{C_i} = C_i + \Phi(D_i) - \Phi(D_{i-1})$$

$\Phi(D_i)$ → Potential of data structure after i^{th} operation

$\Phi(D_{i-1})$ → Potential of data structure before i^{th} operation.

Before we begin, the potential of the data structure is $\Phi(D_0)$.

Now $\bar{C}_1 = C_1 + \Phi(D_1) - \Phi(D_0)$

$\bar{C}_2 = C_2 + \Phi(D_2) - \Phi(D_1)$

$\vdots$

$\bar{C}_n = C_n + \Phi(D_n) - \Phi(D_{n-1})$

$$\sum_{i=1}^{n} \bar{C}_i = \sum_{i=1}^{n} C_i + \Phi(D_n) - \Phi(D_0)$$

Q/1

if $\Phi(D_n) \geq \Phi(D_0) \Rightarrow \sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \bar{C}_i$ ---①

To ensure this thing, start with $\Phi(D_0) = 0$
because all rest D_i will be $R^+ \geq 0$

⇒ Now we will come back to our stack Problem

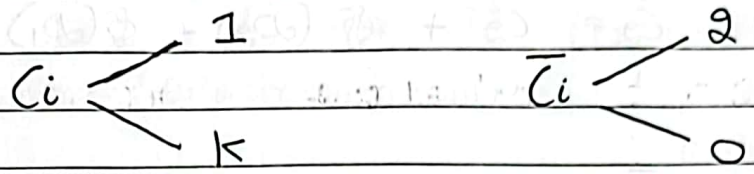
- Push ⇒ $\bar{C}_i = 1 + \Delta\Phi = 1 + 1 = 2$
- Multipop (k) ⇒ $\bar{C}_i = k - k = 0$

For above, potential function that we have taken is

Φ : The total no. of elements in stack.

So $\Phi(D_0) = 0$ here

Now we observe



$$\bar{C}_i \leq 2$$

$$\therefore \sum_{i=1}^{n} \bar{C}_i \leq 2n$$

And we know $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \bar{C}_i \leq 2n$

We are not interested in amortized cost, we are interested in actual cost. But calculation of A.C is easier.

$$\therefore \sum_{i=1}^{n} C_i \leq 2n$$

∴ instead of n^2 operations, we are doing atmax $2n$ operations.

NOTE→ We define Potential function such that cost of costly operation decreases and/but cost of cheap operation goes up slightly

2 BINARY INCREMENT

```
1   0 0 0 0  } oflipped to 1        Increment(A) {
2   0 0 0 1
3   0 0 1 0                          i = 0
4   0 0 1 1
5   0 1 0 0                          while (A[i]==1)
6     :                                { A[i++] = 0 }
      :
      :
    1 1 1 1  } worst case            A[i] = 1 }
  1 0 0 0 0  } (4 1's flipped)
               (i.e. log n)
```

i.e. calling increment func n times ✓

So, now if we do n no. of additions/ operations, then how many flips we have to do ??

In any i^{th} operation, the no. of flips we do is $\log i$ (max)

So, in worst case, we do $\log n$ flips.

So, for n operations, complexity $O(n \times \log n)$

* But now, we will proove that no. of flips done $\leq 2n$

General Analysis :-

See the LSB column, there $1 \to 0$ is done in every alternative iteration.

See the first node.

flips: $1 \to 0$ K; $0 \to 1$ 1

See the 1^{st} column, there $1 \to 0$ is done after every 4^{th} iteration.

and we go on

So, what we get is in doing n operations

$0 \to 1 \Rightarrow n$ times

$1 \to 0 \Rightarrow n/2 + n/4 + n/8 + \ldots \leq n$

Hence total no. of flips done $\leq n + n \leq 2n$

Amortized Analysis :-

costly operation = $1 \to 0$

cheap operation = $0 \to 1$

flips: $1 \to 0$ K; $0 \to 1$ 1

So, potential function should be such the potential of cheap function can inc. slightly but costly operation should decrease considerably. (i.e. decrease by K)

So, we define potential function.

Φ : No. of 1's in the binary no.

$0 \to 1 \to$ In any iteration, it is done only onee (see prgm) and when we do it potential increases by 1.

$1 \to 0 \to$ In any iteration, it is done atmost say k times (while loop) and when we do it each time, potential decreases by 1. So in doing k times, potential dec. by k.

actual cost ↓ ; $\Delta\Phi$ ↓

flips $\begin{cases} 1 \to 0 & k - k = 0 \\ 1 \to 0 & 1 + (1) = 2 \end{cases}$

So, $\bar{C}_i \leq 2$

$$\Rightarrow \sum_{i=n}^{n} \bar{C}_i \leq 2n$$

And $\sum_{i=1}^{n} C_i \leq \sum_{i=1}^{n} \bar{C}_i \leq 2n.$

So, instead of nlogn operations, we do atmost 2n operations.

Imp Note :- This Binary increment concept can be applied to building binomial heaps. If we are suppose given a binomial heap, we have its binary rep. also, and if we have to add one more node/tree we will have to increment its binary count. (Because merging is basically binary addition)

So, similar analogy in both

So, calling n insertions in binomial heap is actually a linear time algorithm.

3 Dynamic Table A.A. :-

Dynamic Table :- We don't know the size of table we require.

So, we are given certain nos. and we insert this in table.

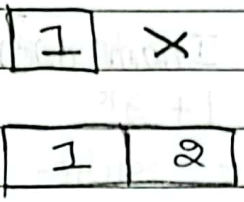
1st element → Create a table of size 1 and insert the element

1

No. of operations = 1

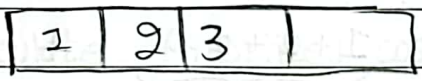
2nd element → Now, we will create a new table, copy the previous elements and insert a new o

Each time when array is full, we create a new array of **double the size**



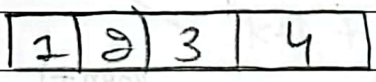
No. of operations = 1 + 1 (1 ↑ copying, 1 ↑ inserting)

3rd element → As our table is full, we create a new table of double the size of previous.



No. of operations = 2+1

4th element → Just add



No. of operations = 1

We go on like this and we will observe that cheap operation is done more frequently than costly operation.

Lets see cost of i^{th} addition / insertion

$$C_i = \begin{cases} 1 & \text{if } i != 1+2^k \quad k \in [0,1,2,\ldots] \\ i & \text{if } i == 1+2^k \end{cases}$$

In worst case, if I am adding n^{th} element, n may be equal to $1 + 2^k$

then in worst case, we may end up doing $O(n^2)$ operations

General Analysis :-

$$\sum_{i=1}^{n} C_i \leq \underbrace{1+1+1\ldots}_{n} + \underbrace{2^k + 2^k + \ldots}_{k \to 0 \text{ to } \log n}$$

$$\sum_{i=1}^{n} C_i \leq n + \sum_{k=0}^{\log n} 2^k$$

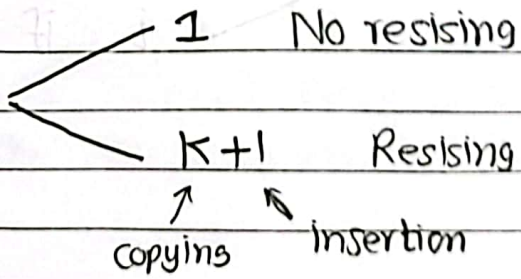
$$\leq n + 2^{\log n + 1}$$

$$\leq 3n$$

So, we end up doing at most $3n$ operations.

Amortized Analysis :-

We have 2 cases, in one case we do resizing and in other case, we are not doing resizing.



Now we define potential function.

Φ : Filled entries - Empty slots.

Note :- Our array is atleast half filled at any point
so Filled entries $\geq$ Empty slots

& $\Phi(D_0) = 0$ (initially empty)

Now, we calculate Amortized cost.

$\bar{C}_i$ $\begin{cases} 1 + 2 = 3 \\ k+1 + \left[\left[k+1 - (k-1)\right] - k - 0\right] \end{cases}$

$= 3$

Now in any case $\bar{C}_i = 3$

$$\sum_{i=1}^{n} \bar{C}_i = 3n$$

$$\therefore \sum_{i=1}^{n} C_i \leq 3n.$$

NOTE :- There can be more than one potential func. for the same problem.

Example, in this case, we can also take Potential function as

Φ : 2× Filled elements − Size of array.

≥ 0

$\Phi(\infty) = 0$

C_i → 1

C_i → k+1

$\bar{C}_i$ → 1 + 2 = 3

$\bar{C}_i$ → $k+1 + \left[2(k+1) - 2k - (2k-k) \right] = 3$

$\therefore \sum_{i=1}^{n} \bar{C}_i = 3n$

$\therefore \sum_{i=1}^{n} C_i \leq 3n$

V.V Imp

AMORTIZED ANALYSIS COMES INTO PLAY WHEN WE DO A SEQUENCE OF OPERATIONS.

8 May

1

Fibonacci Heaps :-

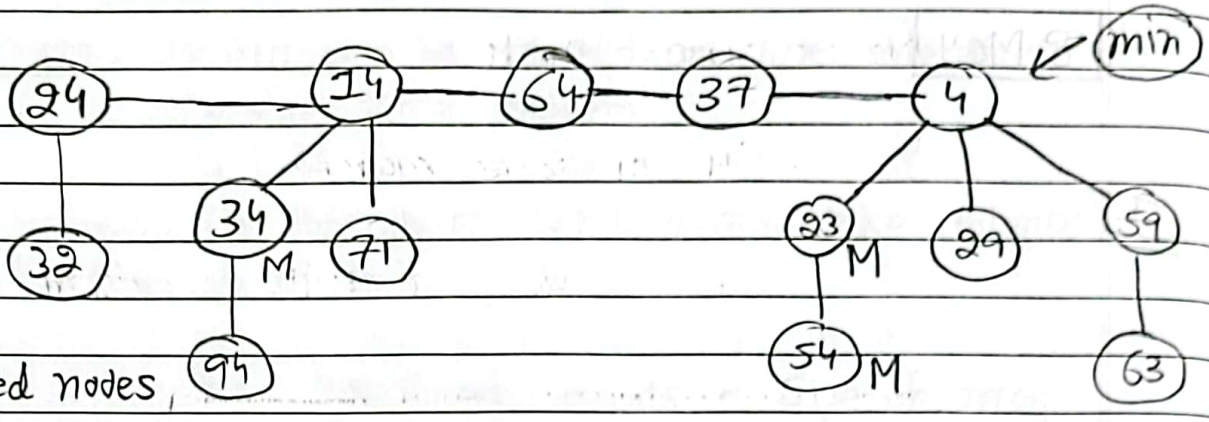
Fibonacci Heaps is a collection of Trees which satisfy heap property. (may not be binomial tree)

Here trees can have any shape and even we can have multiple trees with some no. of nodes (i.e. multiple B_K's for some K's)

So, all the rigid conditions we had in case of binomial heaps is dropped here.

Policy Followed :- Be Lazy when you are allowed to be lazy, when you are forced to do something, do more than what is required

Roots are connected by doubly circular L.L.



M → marked nodes,

When fibonacci heap is given, what basically given is pointer to (min).

Some Definitions :-

- Rank (x) → Number of children of the node x.

  Rank (64) = 0
  Rank (14) = 2

- Rank (H) → Max rank of any node in the fibonacci heap.

- Trees (H) → No. of trees in H

  Trees (H) = 5 in above example

- Marks (H) → No. of marked nodes of H

  Marks (H) = 3 in above example

- $n \rightarrow$ No. of nodes in H

  $n = 14$ in above example.

Now, For Fibonacci Heaps, Potential function is defined as

$$\Phi : \text{Trees}(H) + 2^* \text{Marks}(H)$$

The potential of the above example is $5 + 2^*3 = 11$.

⇒ Now we do various Operations on Fibonacci Heaps

I Insert/Add Operation :-

Create a new node

Add to the root list

Change the min pointer if required.

$O(1)$ → constant no. of pointer changes

$\Delta\Phi = 1$ here

II Union Operation :-

Root list is a doubly circular linked list

Join 2 doubly L.L.

$O(1)$ $\Delta\Phi = 0$ here

Rem

Actual cost can be as bad as $O(n)$

III Delete Min Operation :-

We have the min pointer, so we delete that node and add all the children to root list.

Now, How do we update the minimum ?? Don't be lazy now ☺

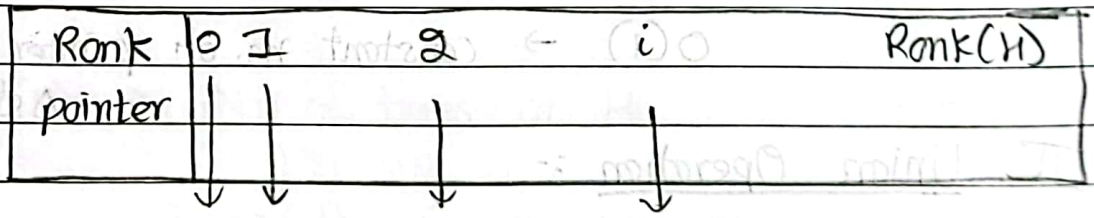
Consolidation :-

We need to traverse the root list to find the minimum. But we will do something more than this.

> What we do more is that if we see two trees with the same rank, merge them

But one more problem is that while we are traversing and we reach a node whose rank is say 10, so how will we know that there is another tree with same rank ??

So, we maintain a table



While iterating, we keep on storing pointers to node with rank i. We will get to know if 2 nodes have same rank

⇒ Merging 2 trees basically means make one of them as the left child of the other

Summary :

Delete Min

- Cut the children and amend them to the root list
- Consolidation

→ We do this because in future if we have to do any more operations on fibonacci heap, they will get easier

Will show that Both are $O(Rank(H))$

Will show that $Rank(H)$ is $O(\log n)$

They are basically A.C.

So, if I add n elements and form a doubly circular LL and then call Delete Min function, in worst case it can take $O(n)$ time (as we have to traverse)

But if we do several such operations, A.A can be done.

Consider 1st sub-operation :-

i.e. cut the children and amend them to root list

Amortised cost = Actual cost + $\Delta \Phi$

$= O(Rank(x)) + Rank(x)$

↑ x

∴ Amortized cost is $O(Rank(H))$

Because $Rank(x) \leq Rank(H)$

Consider 2nd Sub-operation :-

i.e. consolidation

Consider a merge operation

Amortized cost = $1 + (-1) = 0$

Merge operation is free

So, the Amortized cost of consolidation is No. of Trees left after consolidation $\leq Rank(H) + 1$

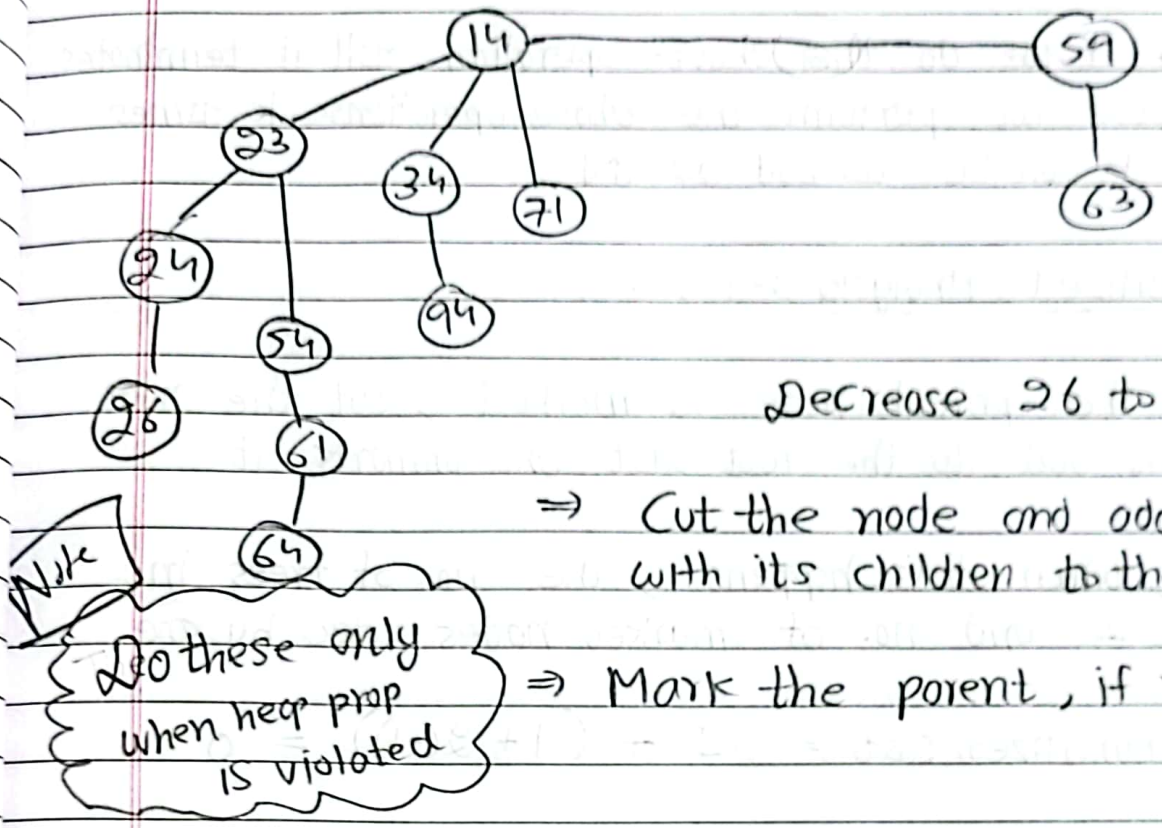
$\therefore O(Rank(H))$

3 Height of Tree in fibonacci Tree can be as bad as $O(n)$

Decrease Key Operation :-

We will not call bottom up heapify

We have to do it in $O(1)$.



Decrease 26 to 18 :-

⇒ Cut the node and add it along with its children to the root list

⇒ Mark the parent, if not marked.

If the parent ~~root~~ node is marked, cut that parent node also then and add it the root list and unmark it.

This process terminates when we reach the root list or- when we reach a unmarked node.

If we reach any unmarked node, just mark it and stop.

NOTE :- If after decreasing the value, heap do prop is not violated, then we don't ~~know~~ anything.

Now, if we do the above operation till it terminates Suppose we perform the above operations k times and k can be as bad as N.

Amortized Analysis :-

Case → If the parent node is marked, cut the node and add to the root list and unmark it.

So, when this happens, the no. of trees inc. by 1 and no. of marked nodes dec. by one.

$$\text{Amortized cost} = \underset{\uparrow}{1} + (1 + 2(-1)) = 0$$

Actual cost of cutting and adding which is 1.

So, this operation is free

Case → If the parent node is unmarked, we just mar it.

No. of trees inc. by 1 and no. of marked nodes here inc. by 1.

$$\text{Amortized cost} = 1 + (3) = 4$$

But this operation will be performed only once.

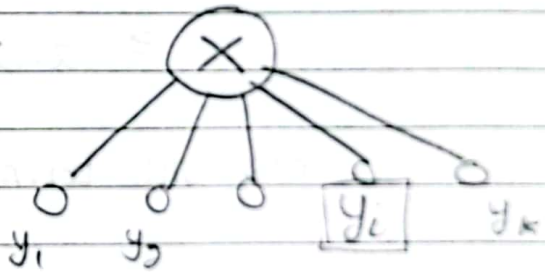
Hence considering both coses, we can say that A.C. of Delete key op. is O(1).

4 We know that Delete Min operation can be done in O(Rank(H)).

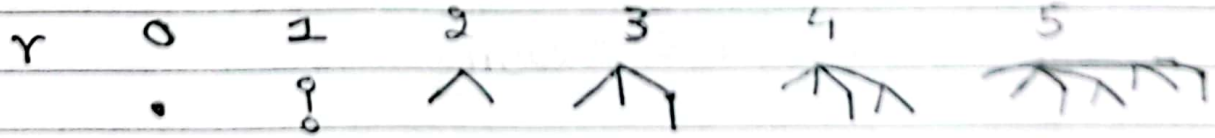
To Proove :- $\boxed{Rank(H) \leq 2 \log n}$

Th^m :- Fix a point in time, Let x be a node, and $y_1, y_2, \ldots y_k$ denote its children in the order in which they were linked to x. Then

For $i > 1$, $Rank(y_i) \geq i-2$



Now, lets define F(r) as Min. no. of nodes in a Fibonacci Heap of rank r.



r	0	1	2	3	4	5
F(r)	1	2	3	5	8	13

We observe $F(r) = F(r-1) + F(r-2)$

where $F(0) = 1$ & $F(1) = 2$

This is the fibonacci sequence.

⇒ Now, lets take any arbitory fibonacci heap whose rank is r and no. of nodes there is n.

Now $n \geq F(r)$ ($\because$ $F(r)$ is defined as minimum)

$n \geq F(r-1) + F(r-2)$

$\geq 2F(r-2)$

$\geq 2.2.F(r-4) \geq 2^k F(r-2k)$

This will terminate when $r-2k = 0 \Rightarrow k = \frac{r}{2}$

$\therefore n \geq 2^{r/2} F(0)$

$$\boxed{n \geq 2^{r/2}}$$

$$\Rightarrow \boxed{r \leq 2 \log n}$$

5 Now we will see that height of fibonacci heap can be linear unlike in case of Binary & Binomial Heap.

To Proove :- Height of Fibonacci Heap can be O(n)

Search

```
bool Search (struct List *node, int X){
 while (node) {
  if (node->data == X)   return true;
  node = node->next;
 }
 return false;
}
```

Add at begging

```
struct List *addatbeg (struct List *node, int i){

 struct List *temp = (struct List *) malloc (sizeof (struct List));
temp->data = i;
 temp->next = node;
 return temp;
}

head=addatbeg(head,i);
```

Add at the beginning

```
Void Addatbeg (struct List **node, int i){
 struct List *temp = (struct List *) malloc (sizeof (struct List));
temp->data = i;
 temp->next = *node;
  *node = temp;}

Addatbeg(&head,i);
```

Add at the beginning ?

```
Void Addatbeg (struct List *node, int i){
 struct List *temp = (struct List *) malloc (sizeof (struct List));
temp->data = i;
 temp->next = node;
  node = temp;}

Addatbeg(head,i);
```

Delete from at the Beginning

```
void Deleteatbeg (struct List **node){
struct List *temp = *node;
 if (*node)  {
  *node = (*node)->next;
 free (temp);}}


Deleteatbeg(&head);
```

Delete at the end

```
voidDeleteatend (struct List **node){
  if (*node)    {
    if ((*node)->next)   {
  struct List *t = *node;
while (t->next->next)      t = t->next;
  free (t->next);  t->next = NULL;  }
  else *node = NULL;    }}
```

Recursive Reverse

```
 struct List *recreverse (struct List *head){
   if (head && head->next) {
      struct List *temp = head, *t;
      if (head->next)   {
/   head = recreverse (head->next);
     t = head;
     while (t->next)     t = t->next;
     t->next = temp;
temp->next = NULL; }    } return head;}
```

Iterative Recursion

```
voidreverse (struct List **head){
 struct List *p = NULL, *c = *head, *n;
  while (c)   {
   n = c->next;
    c->next = p;
    p = c;
    c = n;   }
 *head = p;}
```

Robin Karp's Algorithm

```
p=0;t=0;b=1;
for i =0 to m
{p=(p*2+P[i])%q;t=(t*2+T[i])%q;
b=(b*2)%q;}

while(i<n){
if (p==t) there may be a match;
t=(t*2+T[i]-T[i-m]*b)%q;
 if (t<0) t=t+q;i++;}
if (p==t) there may be a match;
```

Robin Karp's Algorithm

```
for j=0 to m
if(P[j]!=T[i+j]) break;
If (j==m) there is a match;
```

Floyd's Cycle Finding Algorithm

```
bool Cycle (struct List *node){
struct List *t1,*t2;t1=t2=node;
  while (t2)    {
t2=t2->next;
     if (t2==t1)    return true;
      t1 = t1->next;
     if(t2) t2=t2->next; }

return false;}
```

```
voidBubbleSort (int A[], long longint n)
{
Long long int i, j;
int t;
for (i = 0; i < n - 2; ++i)
for (j = 0; j < n - 1 - i; ++j)
if (A[j] > A[j + 1])
          {
               t = A[j];
               A[j] = A[j + 1];
               A[j + 1] = t;
          }
}
```

```
void RQSort (int A[], long long int l, long long int r)
{
      if (l < r)
      {
            long long int k, p, t;
            p = rand () % (r - l + 1) + l;
            t = A[l];
            A[l] = A[p];
            A[p] = t;
            k = Partition (A, l, r);

            RQSort (A, l, k - 1);
            RQSort (A, k + 1, r);

      }
}
```

```
voidInsertionSort (int A[], long longint n)
{
longlongint i, j, max;
int t;
for (i = 1; i < n; ++i)
   {
     j = i - 1;
     t = A[i];

while (j > -1 && A[j] > t)
          A[j + 1] = A[j--];

A[j + 1] = t;
   }
}
```

Quick Sort

```
RQSort (int A[], l, r){
if (l < r)    {
p = rand () % (r - l + 1) + l;
Swap (A[l],A[p]);
 k = Partition (A, l, r);
 RQSort (A, l, k - 1);
 RQSort (A, k + 1, r);    }}
```

Muralidhara V N IIIT Bangalore

Merge Sort

```
MergeSort (int A[], l, r){
 if (l < r)   {
    mid = (l + r) / 2;
    MergeSort (A, l, mid);
    MergeSort (A, mid + 1, r);
    Merge (A, l, r);
 }}
```

Muralidhara V N IIIT Bangalore

Quick Sort

```
Partition (int A[], l, r){
pivot = A[l];  i = l + 1;  j = r;
  while (i <= j)    {
  while (i <= j && A[i] <= pivot) i++;
   while (i <= j && A[j] > pivot)   j--;
```

Muralidhara V N IIIT Bangalore

Quick Sort

```
if (i < j) {
Swap(A[i],A[j]);  i++;  j--;}
  }

i--;

A[l] = A[i];  A[i] = pivot;  return i;}
```

Muralidhara V N IIIT Bangalore

Merge Sort

```
Merge (int A[], l, r){
p = 0;  mid = (l + r) / 2;
 i = l;  j =mid + 1;

while (i < mid+1 && j < r + 1)    {
 if (A[i] <= A[j])   B[p++] = A[i++];
 else  B[p++] = A[j++];    }
```

Muralidhara V N IIIT Bangalore

Merge Sort

```
while (i<mid+1)    B[p++] = A[i++];
while (j<r+1)    B[p++] = A[j++];

 i = l;  p = 0;  while (i < r + 1)
  A[i++] = B[p++];}
```

Muralidhara V N IIIT Bangalore

Randomised Find Rank

```
 RFindRank (A[], l, r, rank){
  if (l < r)    {
    p = rand () % (r - l + 1) + l;
  Swap(A[l],A[p]);
   k = Partition (A, l, r);
       if (rank == r - k + 1)  return k;
       else if (rank < r - k + 1) return
    RFindRank (A, k + 1, r, rank);
  else return RFindRank(A,l,k-1,rank-r+k-1);
     } return r;
```

Muralidhara V N IIIT Bangalore

Operation	Linked List	Binary Heap	Binomial Heap	Fibonacci Heap
Add	O(1)	O(log n)	O(log n)	O(1)
Delete Min	O(n)	O(log n)	O(log n)	O(log n) — A.C
Min	O(1)	O(1)	O(1)	O(1)
Decrease Key	O(1)	O(log n)	O(log n)	O(1) — A.C
Delete	O(n)	O(log n)	O(log n)	O(log n) — A.C
Build Heap	O(n)	O(n)	O(n)	O(n)
Union	O(1)	O(n)	O(log n)	O(1)

```
voidSelectionSort (int A[], long longint n)
{
longlongint i, j, max;
int t;
for (i = 0; i < n - 1; ++i)
 {
        max = 0;
        for (j = 1; j < n - i; ++j)
        if (A[j] > A[max])
                max = j;

        if (max < n - 1 - i)
           {
                t = A[max];
                A[max] = A[n - 1 - i];
                A[n - 1 - i] = t;
            }
  }
 }
```

```
bool BinarySearch (int A[], long lon
{
        long long int mid;
        while (l <= r)
        {
                mid = (l + r) / 2;
                if (A[mid] == X)
                        return true;
                else if (A[mid] > X)
                     r = mid - 1;
                else
                     l = mid + 1;
        }

return false;
}
```