Date: / /
Page No.:

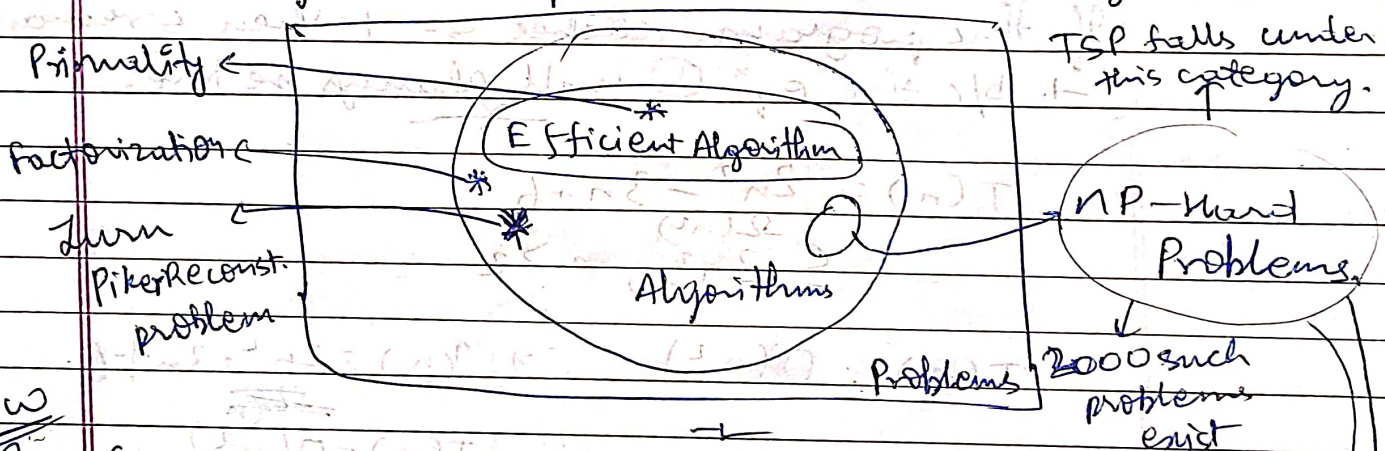
Ch- Intro to Algorithm

→ Algorithm is the method of solving a problem by a finite sequence of instructions.

If an output of a Q is infinite then the algo. doesn't exist for that problem. However, if the output is finite then the algorithm may or may not exist.

→ Hartling problem
→ Hilbert's 10 problem
→ no Algorithm exists. ~~not a solvable problem~~

(TSP) Travelling salesman problem → Brute force algorithm.



H.W

Q. Given n write a program which can give all the permutations of n

Q→ Factorization problem

→ primality = n → given 'n' state whether it is a prime or not.

Date: / /
Page No.:

We will assume that our computer can perform $\approx 10^9$ operations/sec (bops → billion operations per sec) & has 10^6 space

```
#include <stdio.h>
int main() {
    int i, j, k, n, s;
    n = 10000; s = 2;
    for (i=0; i<n; ++i)
        for (j=0; j<n; ++j)
            for (k=0; k<n; ++k)
                s = 2*s + 1 ---(1)
    printf("%d", s);
```

If this program reaches $s = -1$ then s remains -1 b/c the eqn (1) will always be true.

$$T(n) = \underbrace{2n^2 - 3n + 6}_{\Omega(n^2)} \leq 2n^2, \quad n > 2$$

$T(n) = O(n^2)$ ⇒ $T(n) = 2n^2 + 3n + 6$

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

$\leq 3n^2, \quad n \geq 5$

$n^2 > 3n + 6, \quad n \geq 5$

$T(n) = 2n^2 - 3n + 6 \geq n^2 \quad n \geq 3$

$n^2 - 3n + 6 \geq 0$; $T(n) = 2n^2 + 3n + 6$

$\geq 2n^2 \quad n \geq 0$

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

4 3 3'1 $100 \times 10^{-1} = 10^{1} = 0.1$

Date: / /
Page No.:

Polynomial time Algorithm

Th:→ A polynomial of degree k is $O(n^k)$

Pf.) Suppose $f(n) = b_k n^k + b_{k-1} n^{k-1} + \cdots + b_1 n + b_0$

Let $a_i = |b_i|$

$$f(n) \le a_k n^k + a_{k-1} n^{k-1} + \cdots + a_1 n + a_0$$

$$f(n) \le \sum_{i=0}^{k} a_i n^i \le n^k \sum_{i=0}^{k} a_i \left(\frac{n^i}{n^k}\right) \le n^k \sum a_i$$

$$\le c n^k$$

→

Sorting

$a_0\ a_1 \cdots a_{n-1}$; $a_0 \le a_1 \le \cdots \le a_{n-1}$

Bubble Sort

4, 3, 6, 2, 1, 5

A fn" is less than (x, y) if $x < y$ then true else false

$i \rightarrow [0 \text{ to } (n-1)]$

$j \rightarrow 0$ to $n-1$

if $(a_j > a_{j+1})$ { $t = a_j$; $a_j = a_{j+1}$; $a_{j+1} = t$ }

after each loop the largest no. will be at the last

$T(n)$ = no. of comparisons $= (n-1) + (n-2) + (n-3) + \cdots + 1$

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

constant if the no. of inputs is same

Ashika Vanshe

Assignment op. is L to R or R to L

Date: / /
Page No.:

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

$\Theta(n^2)$

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

$T(n)$ is $\Theta(n^2)$; $n \geq 0$

$\Omega(n^2)$

$T(n) = \frac{n(n-1)}{2} \geq \frac{n^2}{4}$; $n \geq 2$

$n(n-1) \leq 2n^2$

$n^2 \geq -n \quad n \geq 1$

swaps are costlier than comparisons

$$\frac{n^2}{2} \leq T(n) \leq n^2$$

Selection Sort

$i \rightarrow 0$ to $n-1$

$m = 0$

$j \rightarrow 1$ to $n-1$

if $(a_j > a_m)$ $m = j$

if $(a_j < a_m)$ $t = a_j$

$a_j = a_m$

$a_m = t$

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

$0 \leq$ no. of swaps $< n$

Insertion Sort

It is comparatively more efficient than selection & bubble sort.

Bubble sort Selection " Insertion "	$O(n^2)$
Merge sort Randomized Quick sort Heap sort	$O(n \log n)$

Date: / /
Page No.:

Divide and conquer

Search

Linear search — $O(n)$

weighted average $= \sum_{i=1}^{n} \left(\frac{1}{n}\right) i$ ← probability

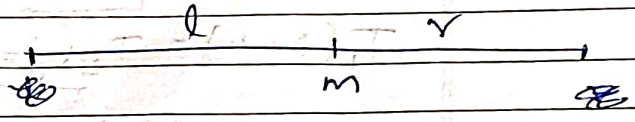
$= \frac{1}{n} \sum_{i=1}^{n} i = \frac{1}{n} \frac{n(n+1)}{2}$

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

Binary search

Both with recursion and loop

$a_0 \le a_1 \le \cdots \le a_{n-1}$




```
BS(A, l, r, x)
if (l < r)
{  m = (l+r)/2
   if (A[m] == x) return m;
   else if (A[m] < x) BS(A, m+1, r, x)
   else BS(A, l, m-1, x)
}
```

$A[m] = x$

if $(A[m] < x)$

$A[l] \le A[m] < x$

$A[0] \ge A[m] > x$

We have to do these until $\frac{n}{2^k} < 1$

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

$\log n = \log_2 n$

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

$\le 2 + 2 + T(n/4) \le 2 + 2 + 2 + T(n/8)$

$\le 2k + T(n/2^k)$ → 2^k

$= 2 \log n + T(0) \le 2 \log n$

no. of operations

$O(\log n)$; $O(\log n)$

$10^3 = 2^{10}$

10^{3+1} 2^{10+1}

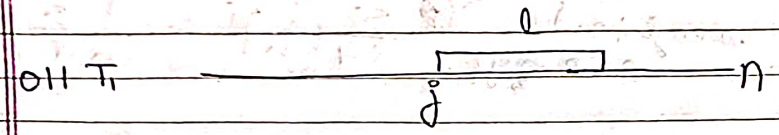
Date: / /
Page No.:

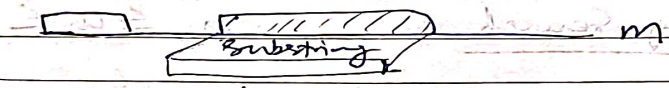
x	.
x	o
o	

Bigs binary search

→ x —

(*) Longest Common Sequence





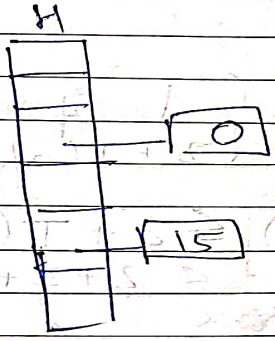
$T_1(i+k) = T_2(j+k)$; $k = 0$ to $l-1$

$n \approx m \approx 10^3$

$T_1(i+k) = T_2(j+k)$; $k = 0$ to $l-1$

$l = 10 \Rightarrow O(n) \times O(n-l) = O(n^2) \times O(l) \Rightarrow O(n^3)$

H(x)



Ex 1 $O(n-l)$
$O(n)$

(*) $T_1(i+k) = T_2(j+k)$

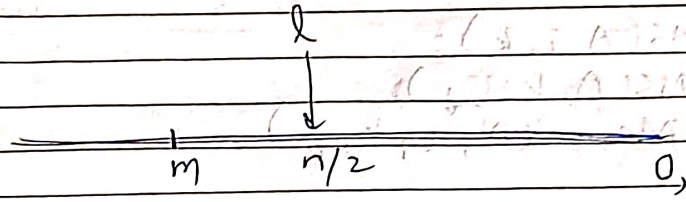
$O(n) + O(n) \rightarrow O(n)$

Date: / /
Page No.:

$l \rightarrow 1$ to n

$F \rightarrow O(y)$

$O(n) \times O(n) \rightarrow O(n^2)$



BS(l, r)

$k = \frac{l+r}{2}$

If (SS(k)) BS(k+1, r)

else BS(l, k-1)

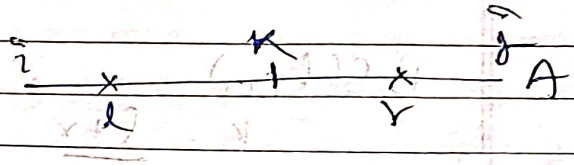
$O(\log n) \times O(n)$

$\hookrightarrow O(n \log n)$

Date: / /
Page No.:

```
MS(A, i, j)
if (i < j)
{
    k = (i+j)/2;
    MS(A, i, k);
    MS(A, k+1, j);
    Merge(A, i, k, j)
}
```

```
Merge (A, i, k, j)
{
  l = i; r = k+1; p = 0;
  while(l ≤ k && r ≤ j)
  {
     if (A[l] ≤ A[r])
          B[p++] = A[l++];
     else  B[p++] = A[r++];
  }
  while (l ≤ k)  B[p++] = A[l++];
  while (r ≤ j)  B[p++] = A[r++];

  p = 0, l = i
  while(l ≤ j)   A[l++] = B[p++];
}
```



$$T(n) = 2 \cdot T(n/2) + 2n$$
$$= 2\left[2\,T(n/2) + 2\frac{n}{2}\right] + 2n$$
$$= 4 \cdot T(n/4) + 4n$$
$$= 4\left[2 \cdot T(n/8) + 2n/4\right] + 4n$$

Date: / /
Page No.:

$= 8.T(n/8) + 6n$

$= 16.T(n/16) + 8n = 2^k T(n/2^k) + 2kn$

$T(1) = 1 \Rightarrow \frac{n}{2^k} = 1 \Rightarrow 2^k = n \Rightarrow k = \log n$

$= n.T(1) + 2 \log n \cdot n = O(n \log n)$

—x—

(*) **Master's Theo.:-**

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

(2) $T(n) = 2.T(n/2) + n$ $\quad O(n \log n)$

(3) $T(n) = 2T(n/2) + 1$ $\quad O(n)$

(4.) $T(n) = T(n/2) + n$ $\quad O(n)$

(5.) $T(n) = 2.T(n/4) + 1$

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

—x—

RQS(A, i, j)

i ——x—— j

if (i < j)

{

$O(n \log n)$ $\quad p = rand() \% (j - i + 1) + i$

$O(n^2)$

gives a random no. b/w i & j

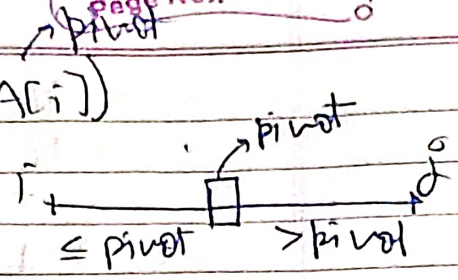
$t = A[i]$

swap(i, p) $\rightarrow A[i] = A[p]$

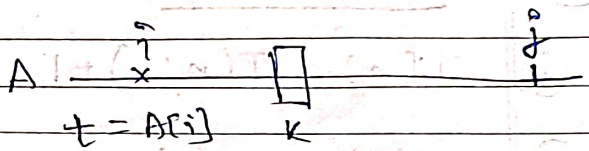
$A[p] = t$

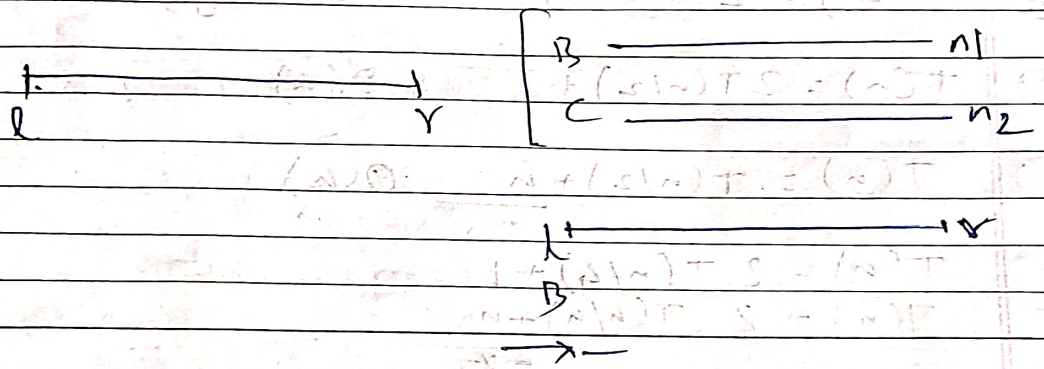
Date: / /
Page No.:

k = Partition (A, i+1, j, A[i])



RQS (A, i, k-1)
RQS (A, k+1, j)
}





Partition (A, i, j, pivot)
l = i ; r = j ;
while (l ≤ r)
{ while (l ≤ r & A[l] ≤ pivot) l++ ;
while (l ≤ r & A[r] > pivot) r-- ;
if (l ≤ r) { t = A[l] ; A[l] = A[r] ; A[r] = t ;
l++ ; r-- ; }
}
k = r-1 ; A[i-1] = A[k] ;
A[k] = pivot ;
return k ;

Date: / /
Page No.:

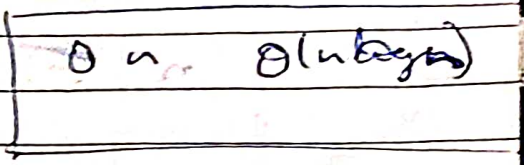
$T(n) = n + T(n_1) + T(n_0)$

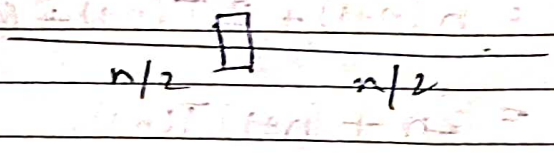
$= n + T(n_1) + T(n - n_1 - 1)$; $n_1 + n_0 + 1 = n$

$= n + T(n-1)$

$= n + n-1 + n-2 + \cdots + 1$

$= \theta(n^2)$

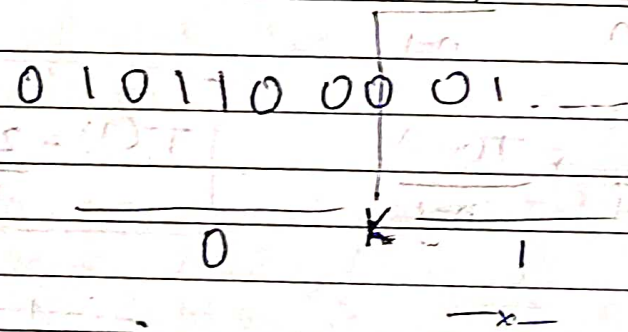




$T(n) = \theta(n) + n + 2T(n/2)$

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

$= \theta(n \log n)$



$$T(n) = \sum_{i=0}^{n-1} \frac{1}{n} \left[T(i) + T(n-i-1) + n+1 \right]$$

$$= n+1 + \frac{1}{n} \sum_{i=0}^{n-1} T(i) + T(n-i-1)$$

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

$T(0)$	$T(n-1)$
$T(1)$	$T(n-2)$
$T(2)$	$T(n-3)$
⋮	⋮
$T(n-1)$	$T(0)$

Date: / /
Page No.:

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

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

$$n\,T(n) = n(n+1) + 2T(n-1) + \left(2\sum_{i=0}^{n-2} T(i)\right)$$

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

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

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

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

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

$$T(1) = \frac{2}{2} + T(0)$$

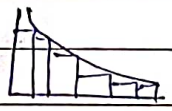
$$\frac{T(3)}{4} = \frac{2}{4} + \frac{T(2)}{3}$$

$$\frac{T(2)}{3} = \frac{2}{3} + \frac{T(1)}{2}$$

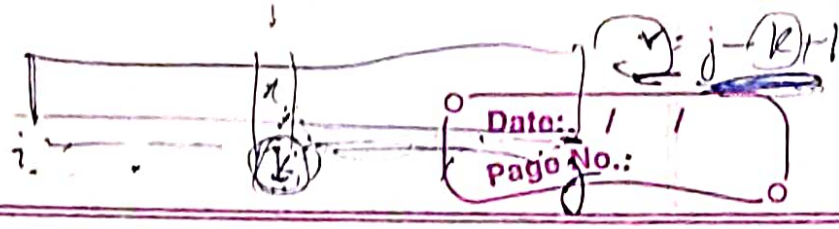
$$\frac{T(1)}{2} = \frac{2}{2} + T(0)$$

$$\sum_{i=2}^{n+1} \frac{1}{i} \le \int_{2}^{n+1} \frac{1}{x}\,dx$$

$$\le 2\log(n+1)$$



$$\frac{T(n)}{n+1} = 2\sum_{i=2}^{n+1} \frac{1}{i} \le 2\log(n+1)$$



$T(n) \leq 2(n+1)\log(n+1)$

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

If two guys have rank 1 then there will be no rank 2 guy but rank 3.

a_0 ——— a_{n-1}

** **Rank(x) = 1 + no. of elements > X**

Find Rank(r) = (Rank(a_i) = r)

$O(n \log n)$ $a_0 \leq a_1 < \cdots < a_{n-1}$

n 1

```
r = 1
i → 0 to n
if (a_i > x) r++;
O(n)
```

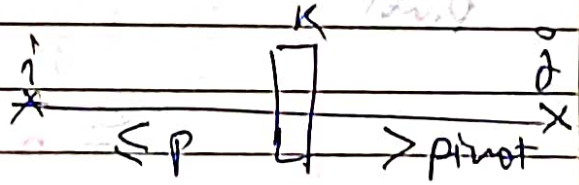
number at rank r is a_{n-r}

$1 \leq r \leq n$

```
RFindRank(A, i, j, r)
if (i ≤ j)
{
    p = rand() % (j - i + 1) + i
    swap(A[p], A[i])
    k = Partition(A, i+1, j, A[i])

    if (r = j - k + 1) return k;
    else if (r < j - k + 1) return RFindRank(A, k+1, j, r)
    else return RFindRank(A, i, k-1, r-j+k-1)
}
else return -1;
```

$O(r)$

$O(n^2)$

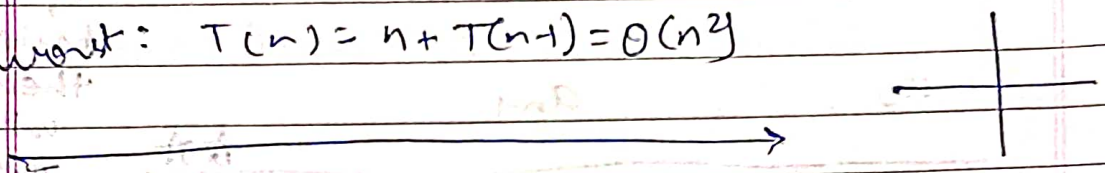


Rank([A[k]]) = i + j - k

Date: / /
Page No.:

$$T(n) \leq n + \max_{0 \leq i \leq n-1} \{T(i), T(n-i-1)\}$$

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



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

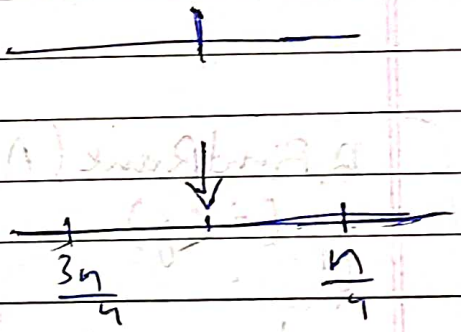
$$= n + \frac{n}{2} + T\left(\frac{n}{4}\right) = n + \frac{n}{2} + \frac{n}{4} + T\left(\frac{n}{8}\right)$$

$$= n\left(1 + \frac{1}{2} + \frac{1}{4} + \cdots\right)$$

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

Good Pivot

$$T(n) \leq n + T\left(\frac{3n}{4}\right)$$



$$\leq n + \frac{3n}{4} + \frac{9n}{16}$$

$$\leq n \cdot \frac{1}{1 - 3/4} = 4n$$

$$T(n) \leq 2n + T\left(\frac{3n}{4}\right) \leq 2n \cdot 4 = 8n$$

$$= 2n + 2n \cdot \frac{3}{4} + 2n \cdot \frac{9}{16} + 2n \cdot \left(\frac{3}{4}\right)^3$$

$$= 2n \left[\sum \left(\frac{3}{4}\right)^i\right]$$

Date: / /
Page No.:

$$T(n) = 2n + 2.\frac{3n}{4} + T\left(\frac{9n}{16}\right)$$

$$T(n) = n + \sum_{i=0}^{n-1} \frac{1}{n} \max\{T(i), T(n-i-1)\}$$

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

$T(0)$ $T(n-1)$
$T(1)$ $T(n-2)$

$T(n) < cn$ $i < n$ $T(i) < ci$ — strong induction

$T(n-2)$ $T(1)$
$T(n-1)$ $T(0)$

$$T(n) = n + \frac{2}{n} \sum_{i=n/2}^{n-1} c.i$$

$$= n + \frac{2c}{n} \sum_{i=n/2}^{n-1} i = n + \frac{2c}{n} \times \frac{3n^2}{8}$$

$T(n) \leq cn$ $T(i) < ci$ $i < n$

$T(n) < cn + c.\frac{3n}{4} \leq cn$ $1 + \frac{3c}{4} < c$

$1 \leq \frac{c}{4}$; $c > 4$

$T(i) \leq 4i \;\forall\; i < n$

$T(n) \leq n + 3n \leq 4n$ $O(n)$

RQS → QS
$O(n \log n)$

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

Prime no.

Date: / /
Page No.:

Randomized algo.

- Las Vegas
  - RQS
  - RFindRank.
- Monte Carlo

which are correct with high probability

Primality $n = 1000$ $1 - \frac{1}{n}$

$a_0 - - - - - a_{n-1}$

Rank(x) $< \frac{n}{2}$ $\quad O(\log n)$

$\frac{1}{2}$ $\quad \frac{1}{n}$

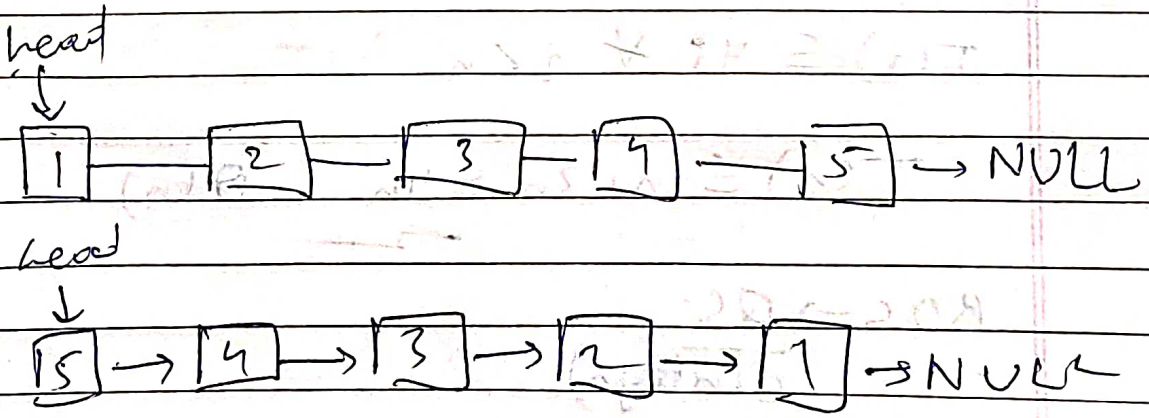
$a_1 - - a_k \quad 1 - \frac{1}{2^k}$

~~$k \geq \log n$~~ $\quad k = \log n$

If $n = 10^9$ then probab. of getting wrong as $\frac{1}{10^9}$

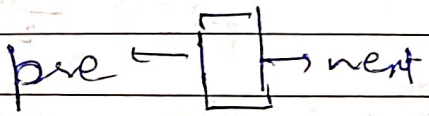
X is a global array

Reversing a list



Date: / /
Page No.:

```
Reverse(struct node **head)
{ struct node p, c, n : p = NULL;
                        c = head;

while(!c)                                   O(n)
{
  n = c->next;
  c->next = p
  p = c; c = n; }                  **head = p;
```



Doubly linked list

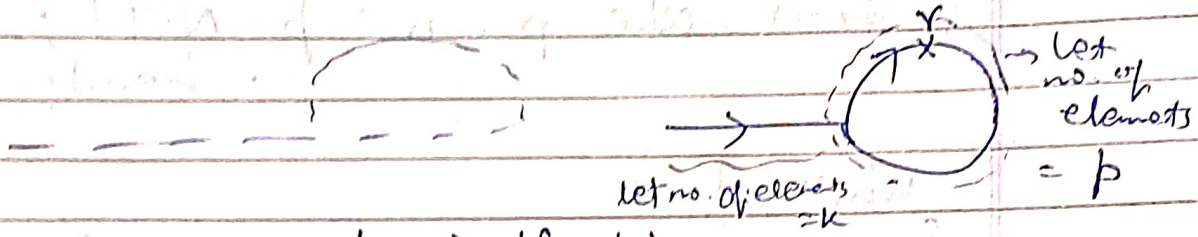
Circular → { Singly, Doubly }

Floyd's Cycle finding Algorithm

```
p1 = head ; p2 = head.

while(!p2)
{
  p1 = p1 -> next;
  p2 = p2 -> next;
  if (!p2) p2 = p2 -> next;
  else return false;
  if (p1 == p2) return true;
}return false
```

Date: / /
Page No.:

Proving that it will terminate before 2n



n → no. of nodes in the LL.
m → " " iterations in FCFA

FCFA ↗ yes ↘ No

$\frac{n}{2} < m < 2n$

$n = k+p$, if $k=0$, $r=p$

$k \neq 0$

Let r; $0 \leq r < p$ $\quad k+r = qp$

$m = 2(k+r)$ P1 & P2 both are at r.

$= k+qp+r$

$2m = 4(k+r) = k+3qp+r$

$m = 2(k+r) \leq 2(k+p) \leq 2n$

$k = q_1 p + r_1$

$0 \leq r_1 < p$

$k+p-r_1 = q_1 p + p$

$k+r = qp$

X, n>0 $\quad X \in \mathbb{R}$, $n \in \mathbb{Z}^+$

read

find X^n

```
y=1
i→0 to n
y = y*x
```

O(n)

```
Pow(x,n)
if (n==0) return 1
else if((n%2)==0)
{return pow(x^2, n/2)}
else return x.pow(x^2, n/2)
```

Date: / /
Page No.:

n — even: $2k \to (x^2)^k = x^{2k} = x^n$

n — odd: $2k+1 \to x(x^2)^k = x \cdot x^{2k} = x^{2k+1} = x^n$.

$$T(n) = \begin{cases} 1 & \text{if } n \text{ is even} \\ 2 & \text{if } n \text{ is odd.} \end{cases}$$

$T(n) \le 2 + T(n/2) \le 2 \log n$

$T(n) \ge 1 + T(n/2) \ge \log n$

k is a no. which depends on n.

$\log n + \lfloor k(n) \rfloor$

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

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

```
Pow(x, n)
y = 1
while (n > 0)
{ if (n % 02 == 1)
    y = y * x
  x = x^2
  n = n/2
}
return y;
```

$\Theta(\log n)$

x^{13}

x	y	n
x	1	13
~~x^2~~		~~6~~
x^2	x	6
x^4	x	3
x^8	x^5	1
x^{16}	x^{13}	0

$\frac{y}{2^k} < 1$

$$n = \sum_{i=0}^{k} b_i 2^i \qquad b_i \in \{0, 1\}$$

Date: / /
Page No.:

$x^n = x^{\sum b_i 2^i} = \prod_{i=0}^{k} x^{b_i 2^i}$; $x^{a+b} = x^a \cdot x^b$

$x^{b_i 2^i}$ — $b_i = 0 \longrightarrow x^{0 \cdot 2^i} = 1$

$b_i = 1 \longrightarrow x^{2^i}$

$b_3 b_2 b_1 b_0$

0 1 0 0 0 1 1 1 0 0 1

$y = \prod_{i=0}^{k} x^{b_i 2^i}$ $\quad b_i \to 1 \quad y = y \times x^{2^i}$; $b_i \to 0$

— x —

$N \to \underline{10^5 \text{ digits}}$ $\quad N = 10^{10^5}$

$B = 3 \times 10^5$ $\quad O((\log N)^2)$

$X \to 3$ digit $\quad \underline{251}$

$y = X^N$

3651
1325

$N_b \to$ ..... ↓

$N = \sum 2^i b_i$

— x —

$N \to 10^5$ digit

$\lfloor \sqrt{N} \rfloor$ $\quad \lfloor \log_2 N \rfloor$ $\quad \lfloor N^{1/k} \rfloor$; $2^k \le N < 2^{k+1}$

while (is less (y, N)

$y = y \times 2$

Date: / /
Page No.:

$k^2 \leq N < (k+1)^2$

$a_0 \; a_1 \; \text{---} \; a_{n-1}$

Given x, find i, j such that $a_i + a_j = x$

I

```
i → 0 to n
j → i to n
if (a_i + a_j == x) return true;
return false
```

$O(n^2)$

$a_0 \leq a_1 \leq \text{---} \leq a_{n-1}$

II

```
{Sort(A) → O(n log n)
```

$x = a_i + a_j$

$a_j = x - a_i$

```
i → 0 to n
j = BS(A, x - a_0, 0, n-1)
```

$O(n \log n)$

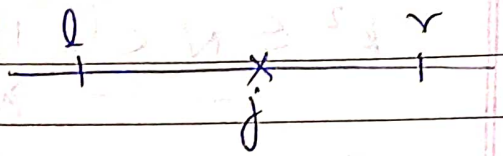
$O(n \log n) + O(n \log n)$

III

```
l = 0; r = n-1
while (l ≤ r)
{ if (a_l + a_r == x) return true;
  else if (a_l + a_r < x) l++;
  else r-- }
return false;
```

$O(n)$

Date: / /
Page No.:



$l \le r$

$a_j \le a_r$

$a_l + a_j \le a_l + a_r < X$

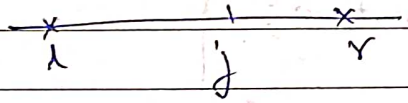
$a_l + a_j < X$

$a_j + a_l \neq X$

$a_l + a_r < X \quad l{+}{+}$

$a_l \le a_j$

$a_l + a_r \le a_j + a_r$



$a_l + a_r > X \quad r{-}{-}$

$\Rightarrow a_j + a_r > X$

$a_l \le a_j$

$X < a_l + a_r \le a_j + a_r$

$\therefore X$

Q. $a_0 \; - - - \; a_{n-1}$

$a_i + a_j = a_k$

Sort (A) $\rightarrow \theta(n \log n)$

$K \rightarrow 0$ to n

$l = 0 \; ; \; r = n-1$

while $(l \le r)$

if $(a_l + a_r == a_k)$ return true

else if $(a_l + a_r < x)$ $l{+}{+}$

else $r{-}{-}$;

return false;

$\theta(n^2)$

"open problem"

Date: / /
Page No.:

→ $a_0 \; - - - - - - - - \; a_{n-1}$

$\min |a_i - a_j| \;,\; i \neq j$

```
min = +∞
i → 0 to n
    j → i+1 to n
    if (abs(a_i - a_j) < min)
        min = |a_i - a_j|
```

$\Theta(n^2)$

Brute Force

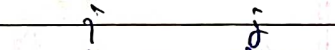
$\Theta(n \log n)$

Sort(A)

```
min = +∞
i → 1 to n
    if (|a_i - a_{i-1}| ≤ min)
        min = |a_i - a_{i-1}|
```

$\Theta(n)$

Actual



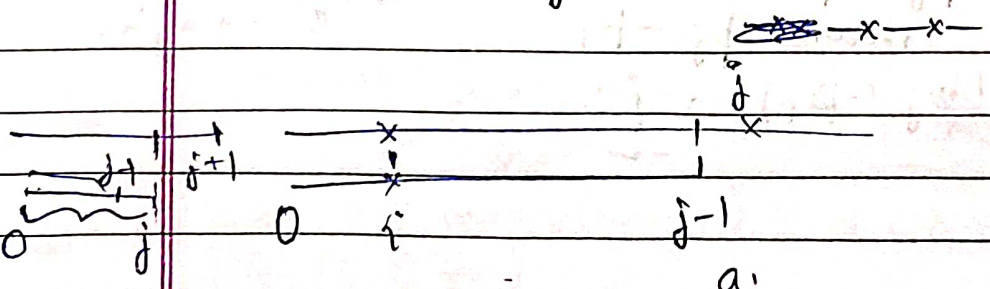
$a_0 \; - - - \; a_{n-1}$

$\max |a_j - a_i|$

$j > i$


```
max = 0
i → 0 to n
    j → i+1 to n
    if (a_j - a_i > max)
        max a_j - a_i
```

$\Theta(n^2)$



a_j

$i = \arg\min_{k<j} a_k$

0 $j \to$

$\Theta(n)$

```
i → 0;
j → 1 to n
    if (a_j - a_i > max)
        max = a_j - a_i
    if (a_i > a_j) i = j
```

Date: / /
Page No.:

a_0 — — — a_{n-1}

p_0 — — — p_{n-1} p_n

$p_0 = 0$

$p_1 = a_0$

$p_2 = p_1 + a_1$

$p_3 = p_2 + a_2$

$p_i = p_{i-1} + a_{i-1}$

$p_n = p_{n-1} + a_{n-1}$

$p_3 = a_0 + a_1 + a_2$

$p_i = a_0 + a_1 + \dots + a_{i-1}$

```
p0 = 0
for i -> 1 to n+1
    pi = p(i-1) + a(i-1)
```

O(n)

a_0 — — —

b_0 ———

Max $p_j - p_i$

$i = 0 \quad j > i$

$i < j$

$p_j - p_i = a_0 + a_1 + \dots + a_{j-1} - (a_0 + a_1 + \dots + a_{i-1})$

$= a_i + a_{i+1} + \dots + a_{j-1}$

```
j -> 1 to n+1
  if (pj - pi > max)
      max = pj - pi
  if (pj < pi) i = j;
```

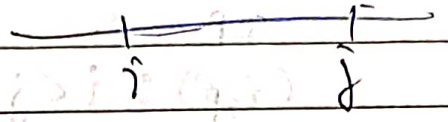
```
i = 0 | sum = a0
j -> 1 to n+1
    s = s + a(j-1)
```

$i < j$

Date: / /
Page No.:

Maximum sum of any subarray:

$[i, j)$




```
i → 0 to n
    j → i to n
        k → i to j
            s = s + a_k
        if (s > max)
            max = s;
```

$O(n^3)$


```
i → 0 to n
    s = 0
    j → i to n
        s = s + a_j
```

$O(n^2)$

```
j → 0 to n
    sum = sum + a_j
    if (sum > max)
        max = sum
        i_max = ...
    if (sum < 0)
        sum = 0
```

$O(n)$

i_{max} ; j_{max} =

```
max_sub_array(arr, n) ; → curret_max = global_max = arr[0];
{ for (i = 0 to n)
  { if (curret_max + arr[i] > arr[i])
        curret_max += arr[i]
    else
        curret_max = arr[i]
    if (curret_max > global_max) global_max = curret_max }
  return global_max }
```

$O(n)$

Date: / /
Page No.:

$a_0 \quad a_1 \quad \text{---} \quad a_{n-1}$

IP

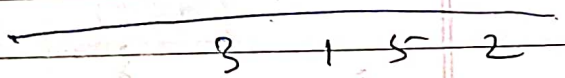
$(i,p) : i<j \quad a_i > a_j$

```
c = 0
i → 0 to n
j → i+1 to n
  if (a_i > a_j) c++;
return c;
```

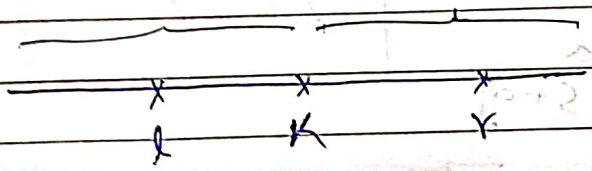
$\theta(n^2)$

$0 \le n(IP) \le \frac{n(n-1)}{2}$

$a_i > a_j$



(3, 1)
(3, 2)
(5, 2)




```
if (A[l] ≤ A[r]) l++;
else {r++; c = c+k-l+1;}
```

```
MS(A, i, j)
if (i<j)
{ k = (i+j)/2
  return (MS(A,i,k) + MS(A,k+1,j) + Merge(A,i,k,j))
  return 0;
}
```

same as that of merge sort plus this time

$T(n) = 2T(n/2) + \theta(n) = \theta(n \log n)$

Date: / /
Page No.:

no. of contributions of inversion pair

$(k-i+1) \times (j-k)$

n_1 1, 2, ——— 100

n_2 101, ..
102 ..

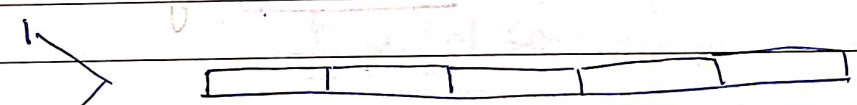
n_k 200 ..

$n = \sum n_i$; $k = \frac{n}{2}$

$O(kn)$

$k = \frac{n}{2}$

$O(n^2)$; $n = 10^5$



$n_1 + n_1$ $n_1 + n_2 - 1$

$k < \log n$

$O(n \log n)$

$O(kn)$

$k = 10$

$n = 10^5$

$n_1 + n_2 + n_3 + n_4 - \cdots = n$

k $k/2$ $k/4$ $\therefore O(nk)$

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

B n_1 n_2 ——— n_k

B n_1 n_2 ——— ——— n_k

n_1+n_2 n_3+n_4

A ——— ——— ——— ——— ———

i, k, j

i, k k+1, j

Date: / /
Page No.:

n_i	3	6	1	4	7	2	5	1
p_i	3	9	10	14	21	23	28	29

0 1 2

r = 0 3 9 10

Merge(A, p_{i-1}, $p_r - 1$, $p_{r+1} - 1$)

Stable Sorting

Radix sort, Count sort

```
d b = 1
while (d--)
    i → 0 to n
    c[(A[i]/b) %10]++;
    i → 1 to 10
    c[i] = c[i] = c[i-1]
    i → n-1 to 1
    B[--c[(A[i]/b) %10]] = A[i]

    i → 0 to n
    A[i] = B[i] : b = b x 10
```

O(dn)

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

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

Date: / /
Page No.:

MASTER THEOREM FOR DIVIDING FUNCTIONS

a & b should be constants

there shouldn't be log n in f(n)

f(n) ≥ 0

① $\log_b a$ ② k

$T(n) = aT(n/b) + f(n)$

$a \geq 1 \quad b > 1$

$f(n) = \Theta(n^k \log^p n)$

Case 1: if $\log_b a > k$ then $\Theta(n^{\log_a b})$

Case 2: if $\log_b a = k$

if $p > -1$ $\Theta(n^k \log^{p+1} n)$

if $p = -1$ $\Theta(n^k \log\log n)$

if $p < -1$ $\Theta(n^k)$

Case 3: if $\log_b a < k$ if $p \geq 0$ $\Theta(n^k \log^p n)$

if $p < 0$ $O(n^k)$

not Θ.

1.) a) $a = 2 \quad b = 2 \quad f(n) = (n^2 \log^0 n)$

$p = 0 \quad k = 2$

$\log_2 2 = 1 < 2$

$\Rightarrow \Theta(n^2 \log^0 n) = \Theta(n^2)$

b.) $a = 1 \quad b = 2 \quad f(n) = (n^0 \log^1 n)$

$p = 1 \quad k = 0$

$\log_2 1 = 0$ $\Theta(n^0 \log^{1+1} n)$

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

Date: / /
Page No.:

$\log_3 3 = 1$ $\quad k = 0$

$n^0 \log^1 n$

$O(n^1)$

$S = 1 \quad 3 \quad 6 \quad 10 \quad -- \quad T_n$

$S = \quad 1 \quad 3 \quad 6 \quad -- \quad T_{n-1} \; T_n$

$T_n = 1 + 2 + 3 + 4 + --$

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

$\log_b a = \log_4 3 = 0.--$

$n^1 \log^1 n \quad k = 1 \quad p = 1$

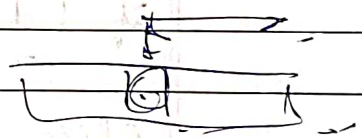
$n^1 \log n$

$\log_b a = \log_2 2 = 1$

$k = 1 \; p = 1 \quad n^1 \log^2 n$

test1.pdf

$K \rightarrow RFR$



$T(k/2) + n < T(n/2) + nR$

$\log_2 1 = 0 \quad k = 1 \quad p = 0 \quad O(n)$

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

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

$\frac{n-p}{2^p} = 1 \Rightarrow n - p = 2^p$

$\Rightarrow n = 2^p + p$

$\frac{n-2}{2^p} \quad np$

$T(n) = \quad \log_2 2 = 1 \quad k=1 \quad p=0$

$n \log n$

Date: / /
Page No.:

$\log_2 1 = 0 \quad k=1 \quad p=0 \quad n \lg^0 n = n$

Q.) 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 1 1 0 0

We have to find a binary substring such that it has equal no. of 1s & 0s and of all such substrings possible find that substring has that has max length.

Soln) 1 0 1 0 1 1 1 0 1 0 0 0 1 0 0 0 1 1 0 0
1 0 1 0 1 2 3 2 3 2 1 0 1 0 1 -2 -1 0 -1 -2

Initially $T[c] = -1$

Now

if $(T[c] == -1)$
$T[c] = i$

else
if $(i - T[c]) > max$
$max = i - T[c]$

But start $c = n$ b/c you get SEG fault since $c \in [-n, n]$

∴ Add n so $c \in [0, 2n]$

$(a \wedge b) \wedge c = a \wedge (b \wedge c)$

$a \wedge b \wedge c = a \wedge c \wedge b$

$a \wedge 0 = a \; ; \; b \wedge b = 0$

Date: / /
Page No.:

Q.) A = [2, 2, 3, 3, 4, 4, 5, 6, 6]

i.e. given an array with every no. occurring in pair & no in pairs are consecutive except one single digit which is 5 here. ~~then~~ find that no. (5 here).

Sol^n $\Theta(\log n)$

Q.) The same Q. as above but this time the no. in pair are not consec.

Sol^n $\Theta(n)$

Take XOR of all no.

—x—

Q.) XOR of l to r

a_1 a_2 a_3 a_4

Q.) Food Eating comp.

c people are partic.

a competitor can eat only t amount of food/sec.

what is the min. time in ~~which~~ which whole food is complete.

there is a mark on each dish which represents the quantity of food in it.

Date: / /
Page No.:

Q.) cube root of n

soln

```
while(r-l > acc){

}
```

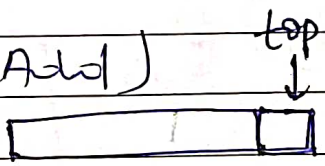
acc = 10^{-6}

$\rightarrow \theta\left(\log\left(\frac{n}{acc}\right)\right)$

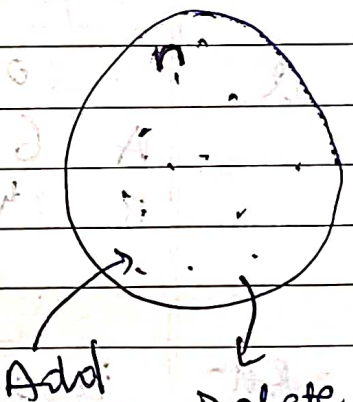
—X—

DDS (Dynamic data set problem)

- Stack FILO/LIFO
  - Push O(1) (Add)
  - Pop O(1) (delete)

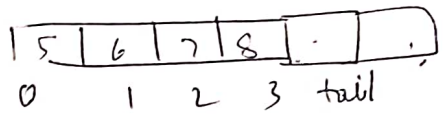


- Queue FIFO/LILO
  - Enqueue O(1) (Add)
  - Dequeue O(1) (delete)




```
POP(S)
  top--;
  return (S[top+1])
```

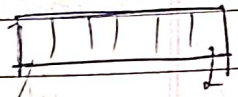

```
PUSH(S, X)
  S[++top] = X
```



Date: / /
Page No.:

If we know that the max. no. of no. in stack is always 10, but no. of insertion possible = 25 the overflow may happen for which we use %10.

```
Dequeue (Q)
    head++
    return Q[head-1];

Enqueue (Q, int x)
    Q[++tail] = x;
```



The no. are from [head, tail)
included excluded

Queue:
- Enqueue
- Dequeue head tail

You should prefer add at the end & delete from ~~add at the beg, delete at the end~~ the beg.

Q.

	0	1	2	3	4	5
A	6	13	7	8	4	9
B	4	2	4	4	-1	-1

Sol-

```
i -> 0 to n
    j -> i+1 to n
        if (A[j] < A[i]) B[i] = j; break;
if (j == n) B[i] = -1
```

$\Theta(n^2)$

else if ✓ else if

Date: / /
Page No.:

top = -1
i → 0 to n
if (top < 0) S[++top] = i
else { if (A[S[top]] < A[i]) S[++top] = i

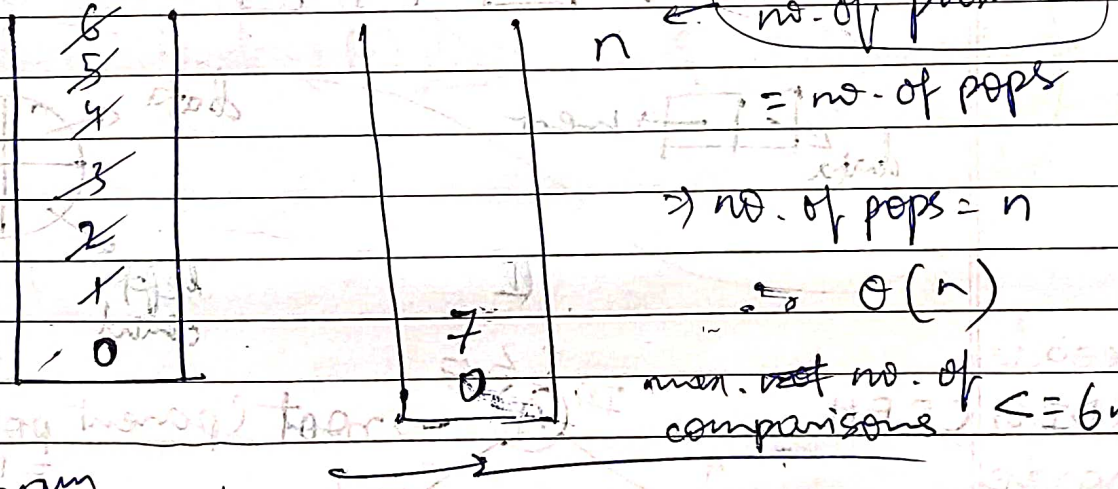
In the stack we are only storing the indices

else
{ while (top ≥ 0 & A[S[top]] ≥ A[i])
B[S[top--]] = i
S[++top] = i; }

while (top ≥ 0)
B[S[top--]] = -1

0	1	2	3	4	5	6	7	8	9	10
6	8	11	14	10	13	9	7	1	3	2

8 7 4 4 6 6 ~~7~~ 8



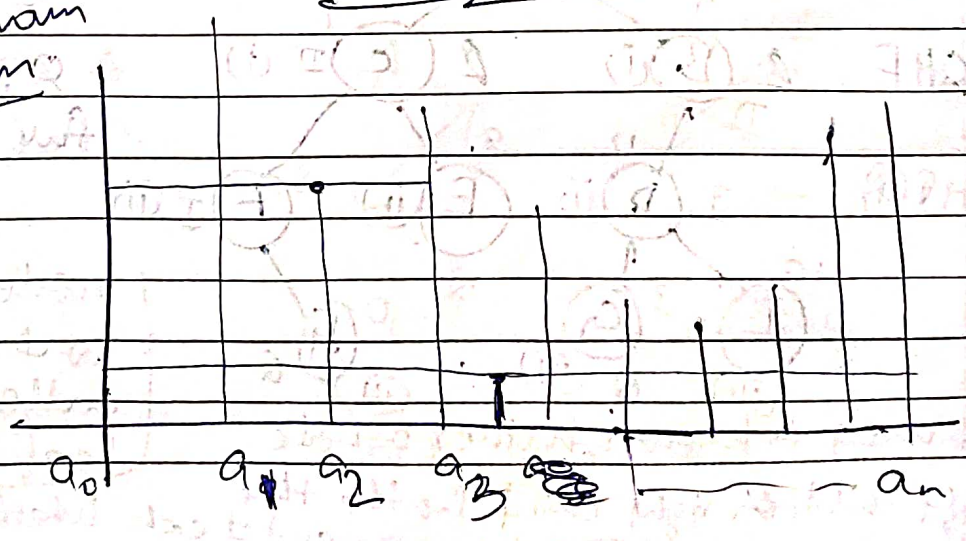
no. of pushes = no. of pops

⇒ no. of pops = n

∴ O(n)

max. ~~not~~ no. of comparisons <= 6n

Histogram problem



वो खड़ा है He is standing

Date: / /
Page No.:

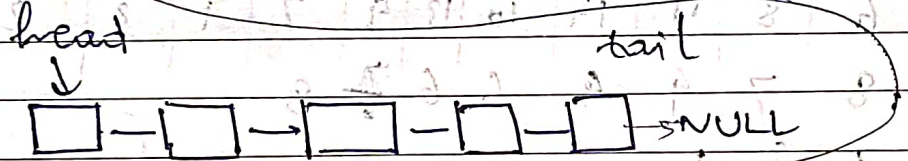
```
max = 0
i → 0 to n
    j → i+1 to n
        min = a[j]
        k → i+1 to j+1
            if (a[k] < min)
                min = a[k]
        if (min * (j-i) > max)
            max = (j-i) * min
```

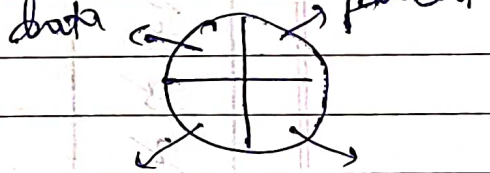
$O(n^3)$

$O(n^2)$

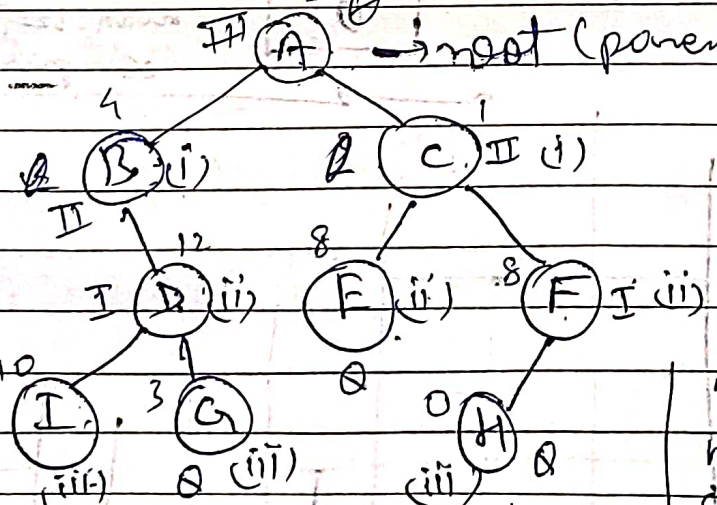
BINARY TREE

for linked list





Preorder
ABDIGCEFH
inorder
BIDGAECHF
postorder
IGDBEHFCA



nodes with no child are called leaf nodes.

If you want to know whether a node is left child or right child. Go the the parent pointer and check whether left or right

no. of children allowed for each node in a binary tree → {0, 1, 2} = S

For full binary tree → S = {0, 2}

Date: / /
Page No.:

1.) Preorder (node)
if (node)
{ Visit (node)
Preorder (node → lc)
Preorder (node → rc)
}

—x—

2.) Inorder (node)
if (node)
{ Inorder (node → lc)
Visit (node)
Inorder (node → rc)
}

—x—

3. Postorder (node)
if (node)
{ ~~Inorder (node → lc)~~ Visit

Post order (node)
if (node != NULL)
{ Postorder (node → lc)
Postorder (node → rc)
Visit (node);
}

O(n)

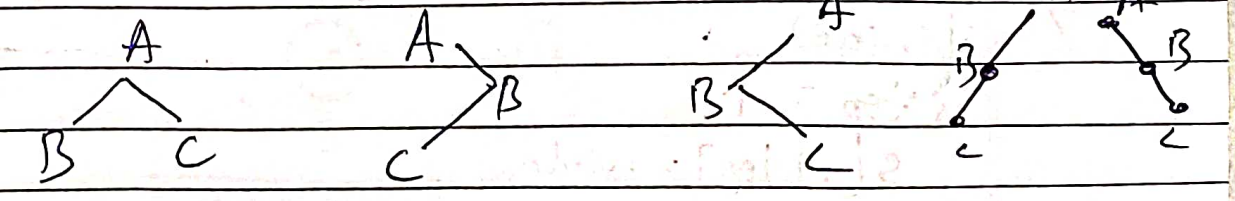
n = total no. of nodes

Nodes excluding leaf nodes are called internal nodes

For degenerate binary tree
S = {0, 1}

no. of internal nodes = 1

—x—



Pre:	ABC	ABC	ABC	ABC	ABC
Inord:	BAC	ACB	BCA	CBA	ABC
Post:	BCA	CBA	CBA	CBA	CBA

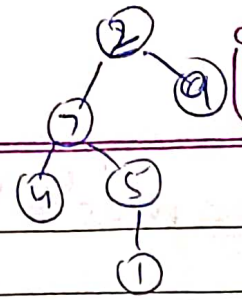
```
Preorder(node)
if(node)
{ top=-1
  s[++top] = node;
  while(top>=0)
     visit(s[top--]); node=s[top+1]
```

If we are given inorder traversal and either a preorder or postorder only then we can uniquely identify a binary tree

Iterative traversals

1)
```
Preorder (node)
if (node)
{ top=-1
  s[++top] = node;
  while (top>=0)
     visit(s[top]) ; node = s[top--]
     if (node->rc)  s[++top] = node->rc
     if (node->lc)  s[++top] = node->lc
}
```

2)
```
Inorder(node)
if(node)
 {top = -1;
  s[++top] = node
  B[top] = F
  while(top>=0)
  if(B[top]==T)  visit(s[top--])
  else{ node = s[top--]
        if(node->rc){s[++top]=node->rc; B[top]=F}
        s[++top]=node; B[top]=T
        if(node->lc){s[++top]=node->lc; B[top]
```



Date: / /
Page No.:

3.) Postorder(node)

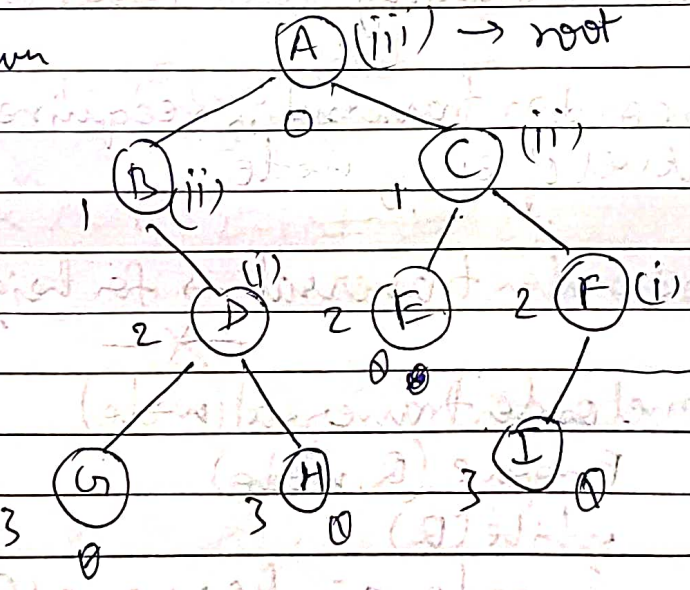
```
if (node)
{ top = -1
  S[++top] = node
  B[top] = F
  while (top >= 0)
    if (B[top] == T) visit S[top--]
    else { B[top] = T; node = S[top--]; S[++top] = Node
           if (node->rc) S[++top] = node->rc; B[top] = 0;
           if (node->lc) S[++top] = node->lc; B[top] = 0;
         }
}
```

→ x ←

Binary Tree

when you come from top-down you will find levels of node.

Vice versa gives you height of the nodes



Level order traversal: A B C D E F G H I → from left to right

$n = n_0 + n_1 + n_2$

- n_0 → no. of nodes with 0 child
- n_1 → no. of nodes with 1 child
- n_2 → no. of nodes with 2 children

Date: / /
Page No.:

→ level of a node = 1 + level of the parent

level of root node = 0

→ height of node = 1 + max (height of lc & rc)

height of leaf node = 0

→ structure of each node contains:-
a.) data b.) ~~lc~~ parent c.) lc d.) rc
e.) l = level of node f.) h = height of the node

→ height of a tree = height of the root node
max. height in a tree = max. ~~height~~ level of a tree

→ The sum of height & level for each node along the path containing max. level will be same.

Preorder traversal is reequired to compute the level of every node

Postorder traversal → for height

(*)
```
levelordertraversal(node)
    Enqueue(Q, node)
    while(Q)
    { node = Dequeue(Q)
      visit(node)
      if(node→lc) Enqueue(Q, node→lc)
      if(node→rc) Enqueue(Q, node→rc)
    }
```
O(n)

Date: / /
Page No.:

→ Perfect binary tree → a full binary tree where all the leaf nodes are at the same level

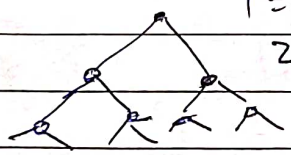
→ Complete binary tree → If you ignore the nodes in the last level then it should form a perfect binary tree & the last nodes should be filled from left to right.

n = total no. of nodes in the tree
h = height of the tree

Q.) if n is given then what is the max. h possible?

Soln Considering degenerate tree → $h \leq n-1 \Rightarrow \boxed{h+2 \leq n+1}$ → (1)

Q.) If h is given then max. no. of nodes possible?



$1 = 2^0$ (at level 0)
2^1 (at level 1)
$4 = 2^2$ (at level 2)
2^i
2^h (at level h)

$$\Rightarrow n \leq \sum_{i=0}^{h} 2^i = 2^{h+1} - 1$$

$$\Rightarrow \boxed{n+1 \leq 2^{h+1}} \quad - (2)$$

From (1) & (2) $\boxed{2^{h+1} \geq n+1 \geq h+2}$ ★★

Date: / /
Page No.:

From (2) $\boxed{h \text{ is } \Omega(\log n)}$

If h is $O(\log n)$ also then it is a balanced tree

For c1:- $\log n \leq h \leq 5 \log n$
then this tree is balanced binary tree

perfect trees are always balanced vice-versa may or may not be true.

l = no. of leaf nodes
i = " " internal "

Th^m $\boxed{l \leq i+1}$

Proof:- Strong induction.

Base cases

$n=1$ $i=0, l=1$ $1 \leq 1$ ✓

or

$n=2$ $i=1, l=1, i \leq 2$ ✓

$n=3$

$l=2$ $i=1$ $2 \leq 2$ ✓

$l=1$ $i=2$ $1 \leq 3$ ✓

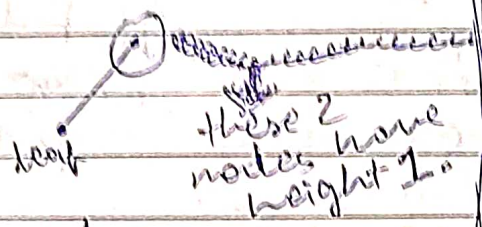
Date: / /
Page No.:

By strong induction we assume that given any tree $< n$ nodes the h^{th} is true.

we will find a leaf node, go to its parent & see whether it has another child or not.

height = 1

Case1 (it doesn't have)



we ~~dete~~ delete these node

$n_{new} = n-1$

$i_{new} = i_1$; $l_{new} = l_1$

$l_1 = l$; $i_1 = i-1$

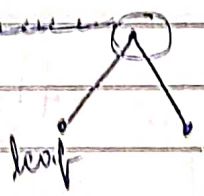
By induction

$l_1 \leq i_1 + 1$

$\Rightarrow l \leq i - 1 + 1$

$\Rightarrow l \leq i \Rightarrow l \leq i+1$

Case2 (it has)



we ~~delete~~ delete both of these nodes

$n_{new} = n-2$

$i_{new} = i_1$ $l_{new} = l_1$

$i_1 = i-1$ $l_1 = l-1$

By induction

$l_1 \leq i_1 + 1$

$\Rightarrow l-1 \leq i-1+1$

$\Rightarrow l \leq i+1$

$n+1 \leq 2^{h+1}$; $l \leq i+1$

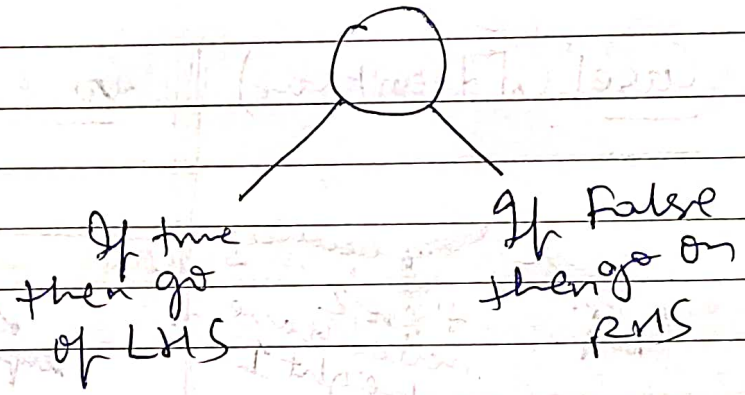
$l+l \leq i+l+1 = n+1 \leq 2^{h+1} \Rightarrow 2l \leq 2^{h+1}$

$\Rightarrow \boxed{l \leq 2^h \Rightarrow h \geq \log l}$ ★

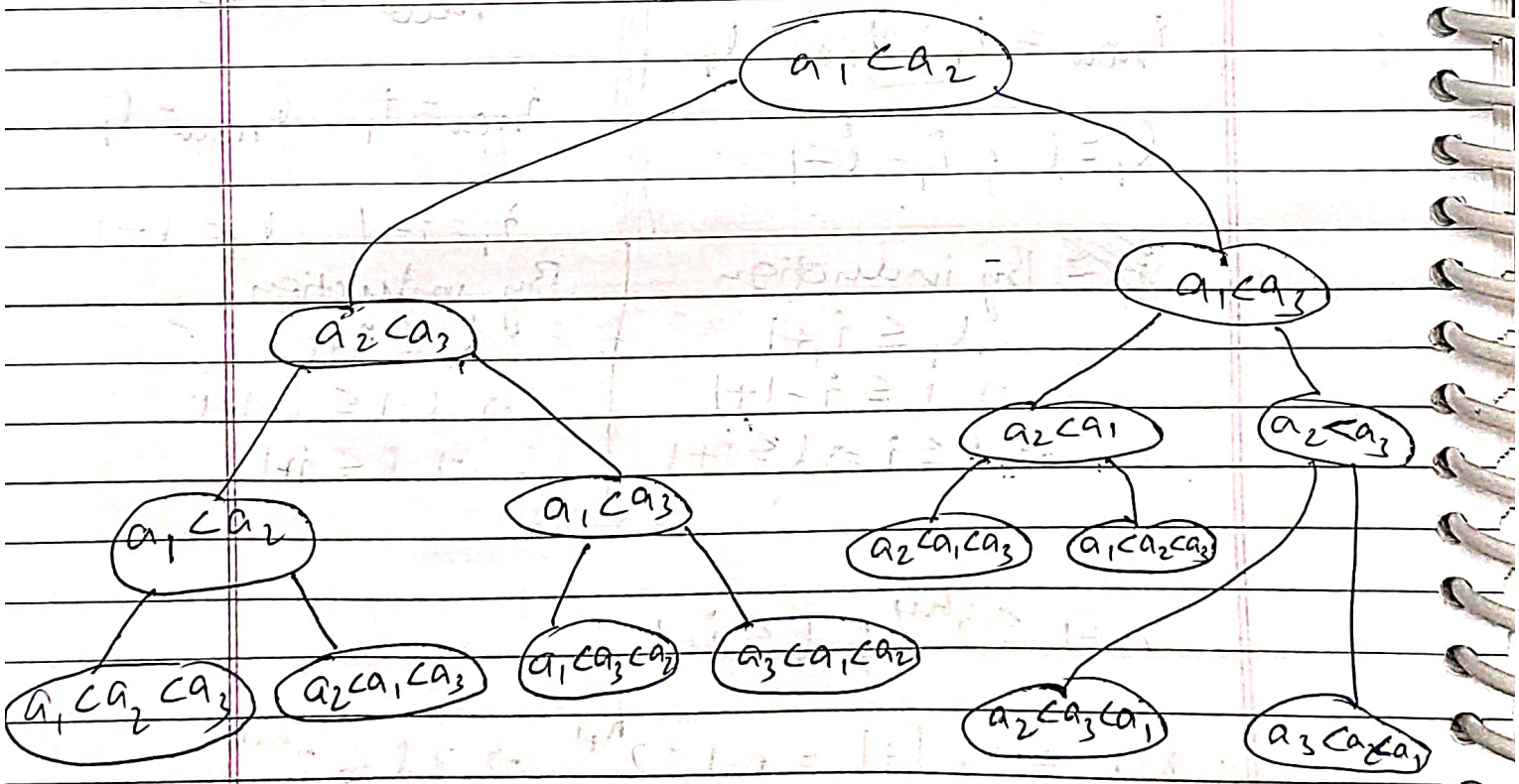
Date: / /
Page No.:

Decision Tree

In each node an algorithm asks a binary question which will have T/F as its ans.



Decision Tree for bubble sort on no. $a_1\ a_2\ a_3$



The total no. of possible permutations for a_1, a_2, a_3 $= 6 <$ the no. of leaf nodes in above tree since some are repeating.

n = no. of numbers given for sorting

Date: / /
Page No.:

Now, the worst case no. of comparisons done by a sorting algo is h

$\therefore h > \log l > \log(n!)$

To show $\log n! > \frac{n}{2} \log \frac{n}{2}$

which will imply that any comparison based sorting algo will have lower bound as $n \log n$

i.e. $\Omega(n \log n)$

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

$\Rightarrow \log n! \leq n \log n$

(not useful since we have to show this)

$n! = \prod_{i=1}^{n} i \geq \prod_{i=\frac{n}{2}}^{n} i \geq \left(\frac{n}{2}\right)^{n/2}$

$\Rightarrow \log n! > \frac{n}{2} \log \frac{n}{2}$ (Q.E.D)

means → if you have an implementation of priority queue you can use the same structure for stack as well as queue.

Date: / /
Page No.:

Generalization of these 2 data structures →

Stack - FILO / LIFO
- Push ( ) | O(1)
- Pop ( )

Queue - FIFO / LILO
- Enqueue | O(1)
- Dequeue

→ Priority Queue -
- Enqueue ( ) - Add ( )
- Dequeue - Delete min ( ) | Delete Max ( )

↕ 2 option deleting the min priority or max priority

DDS

Add

delete O(log n)

Since stack & queues can be done in O(1)

we are only going to study minimum priority queue

↓

Ex:→ of a priority queue → binary heap

Although we can do Add in O(1) & dequeue in O(n)
It is not advantageous b/c if in n
$\frac{n}{2}$ is Add & $\frac{n}{2}$ is delete
the order is $O(n^2)$

So better to do logn for both add & delete

~~If you delete the min n times~~

delete min will be queue
delete max " " stack.

Date: / /
Page No.:

deletion of min can't be of $\Theta(1)$?

Since if you delete min priorities n times then the array formed will be increasing, this means you sorted an array & its order is $\Theta(1) \times n = \Theta(n)$

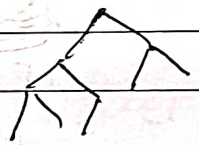
not possible b/c we have already seen that any comparison sorting algo. has $\Omega(n \log n)$

MIN BINARY HEAP

Binary heap

Property-1

It is a complete binary tree (CBT)



BT (Binary Tree) → h is $\Omega(\log n)$

CBT we have to show

h is $O(\log n) \rightarrow O(\log n)$

i.e. the CBT that we have has to be a balanced binary tree.

For general binary tree

$l \leq i+1$

$2l \leq l+i+1$

$\leq n+1$

$\Rightarrow l \leq \frac{n+1}{2}$

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

$h+1 \geq \log(n+1)$

h is $\Omega(\log n)$

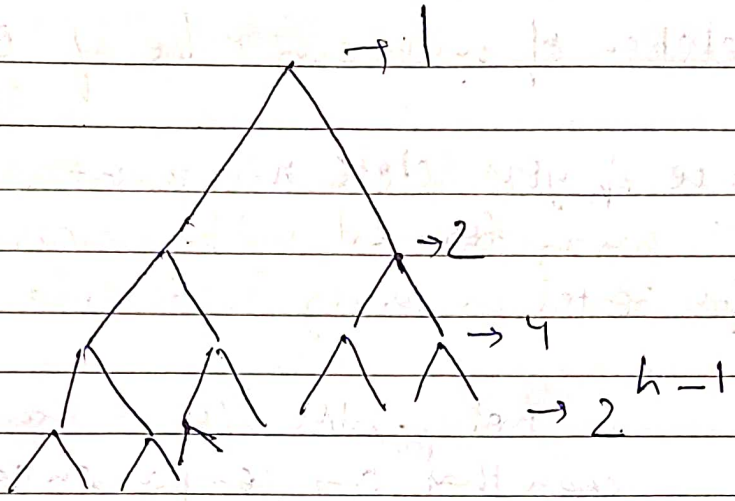
$h \leq n+1$

↓

For CBT it's not possible

Date: / /
Page No.:

Proof



$$\sum_{i=0}^{h-1} 2^i = 2^h - 1 \qquad \Rightarrow n \geq 2^h$$

$$\Rightarrow \boxed{h \leq \log n} \quad \text{for CBT}$$

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

for CBT

$$\boxed{l \leq \frac{n+1}{2}} \text{ ; for any BT}$$

For CBT

n	l
1	1
2	1
3	2
4	2
5	3
6	3
⋮	⋮
h	$\frac{h+1}{2}$

$$\Rightarrow \boxed{l = \frac{n+1}{2}}$$

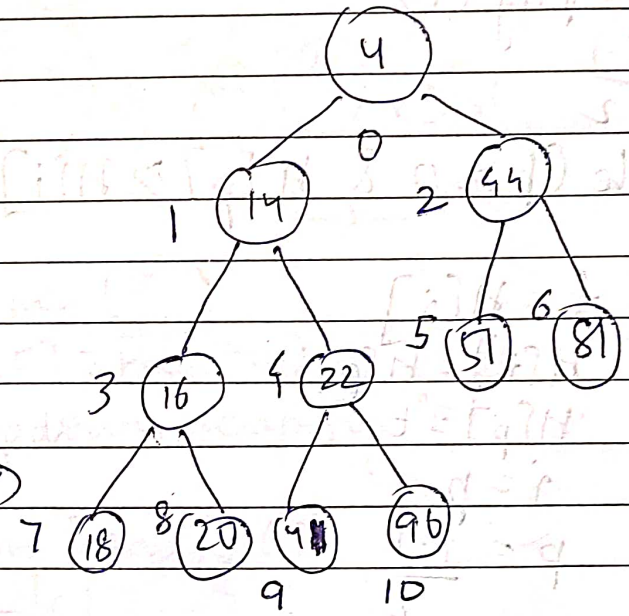
For CBT

Date: / /
Page No.:

Property 2 Heap :-

priority(node) ≥ priority(node→parent)

∀ node in the heap



The largest no. is among the leaf nodes

By level order traversal we form array H

	0	1	2	3	4	5	6	7	8	9	10
H:	4	14	44	16	22	51	81	18	20	41	96

For now we will assume that this entire complete binary heap can be represented just through an array

i.e. each index will represent a node containing data of that node

now if we are given a node represented by index i

Date: / /
Page No.:

Rep:

index of its LC = 2i+1

RC = 2i+2

Parent = $\left(\frac{i-1}{2}\right)$

```
BottomHeapify(i)
    p = (i-1)/2
    while (p >= 0 && H[p] > H[i])
    {
        t = H[i]
        H[i] = H[p]
        H[p] = t
        i = p
        p = (i-1)/2
    }
```

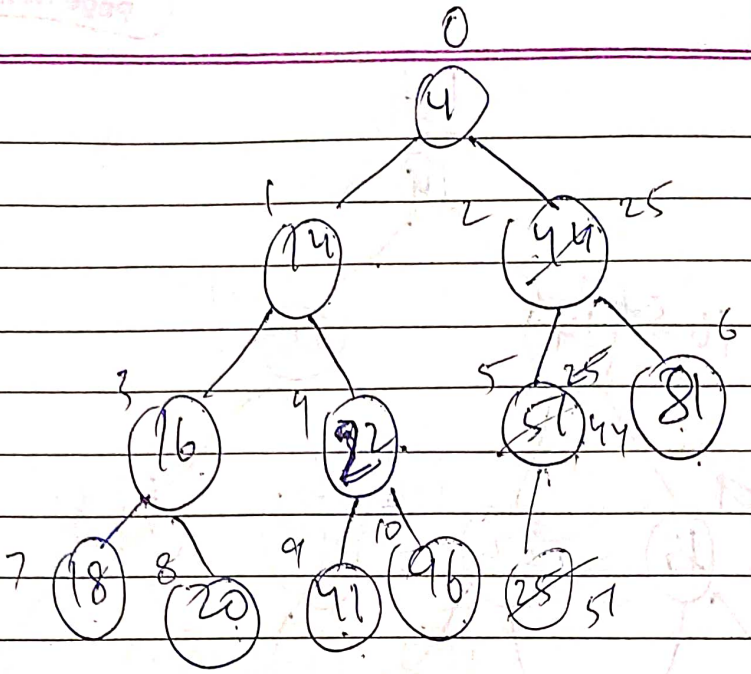
This while loop can run for almost height of the tree

$\therefore h \leq \log n$

$\Rightarrow O(\log n)$

```
Add(x, H)
    H[n++] = x;
    BottomUpHeapify(n-1)
```

Date: / /
Page No.:



→ Delete min ( )

```
H[0] = H[--n]
Topdown heapify (0)
```

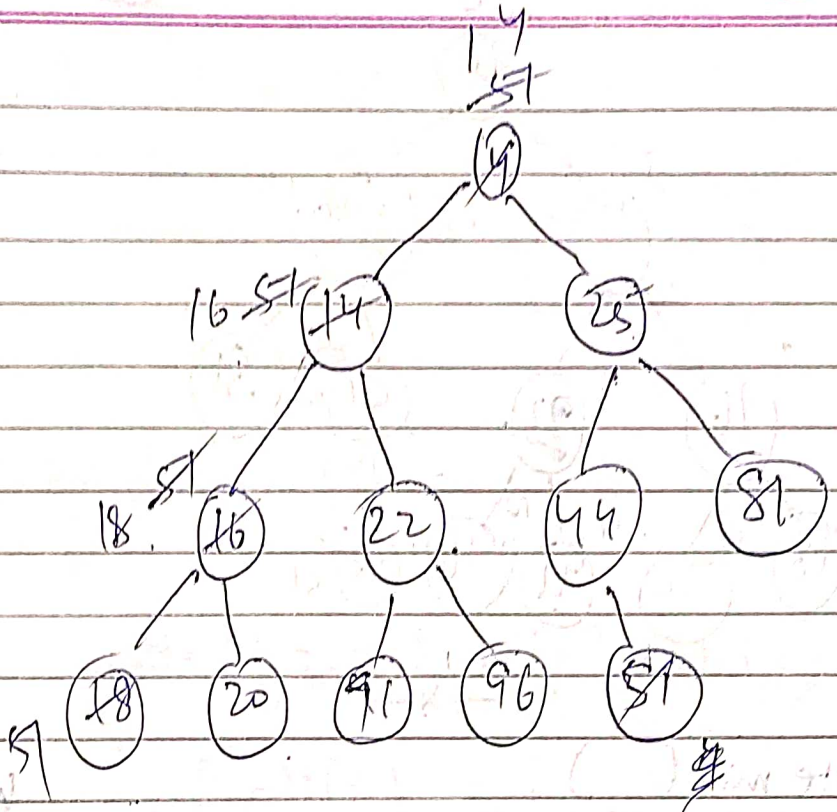
→ Topdown heapify (i)

```
while( 2i+2 < n)
{ if (H[2i+1] < H[2i+2])
      l = 2i+1
  else l = 2i+2;

  if (H[i] > H[l])
      { t = H[i]; H[i] = H[l]; H[l] = t }
      [ i = l;
  else break;
}

if (2i+1 < n & H[i] > H[2i+1]
{ t = H[i]; H[i] = H[2i+1]; H[2i+1] = t }
```

This while loop can travel for almost the level or the tree = height of the tree

$h \leq \log n$

$\Rightarrow O(\log n)$



→ update (i, x) // H[i] = x

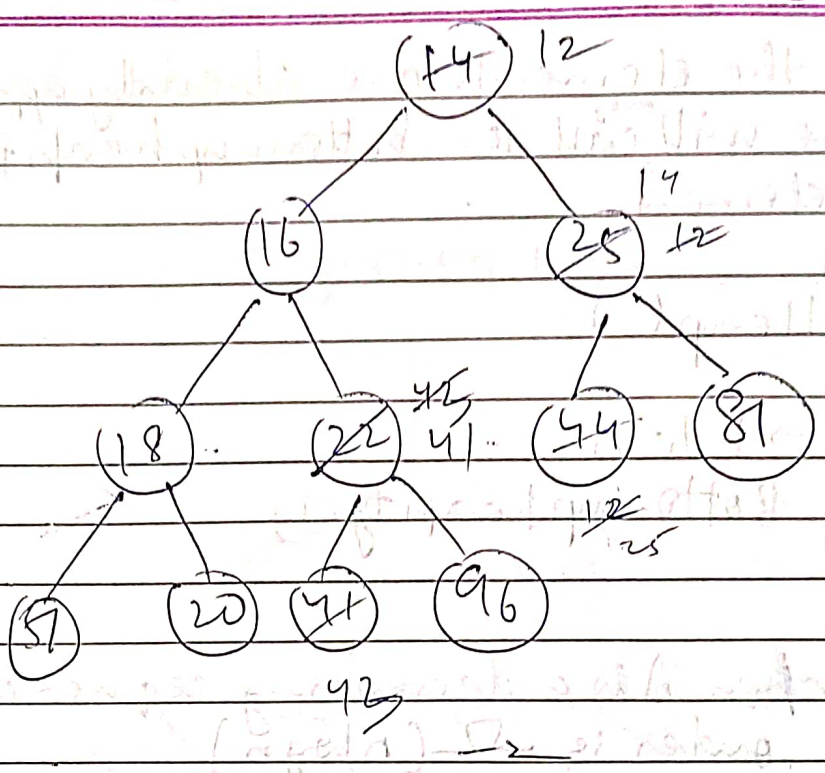
```
if (H[i] < x)
{ H[i] = x
  Topdown heapify (i)
}
else if (H[i] > x)
{ H[i] = x
  Bottom up heapify (i)
}
```

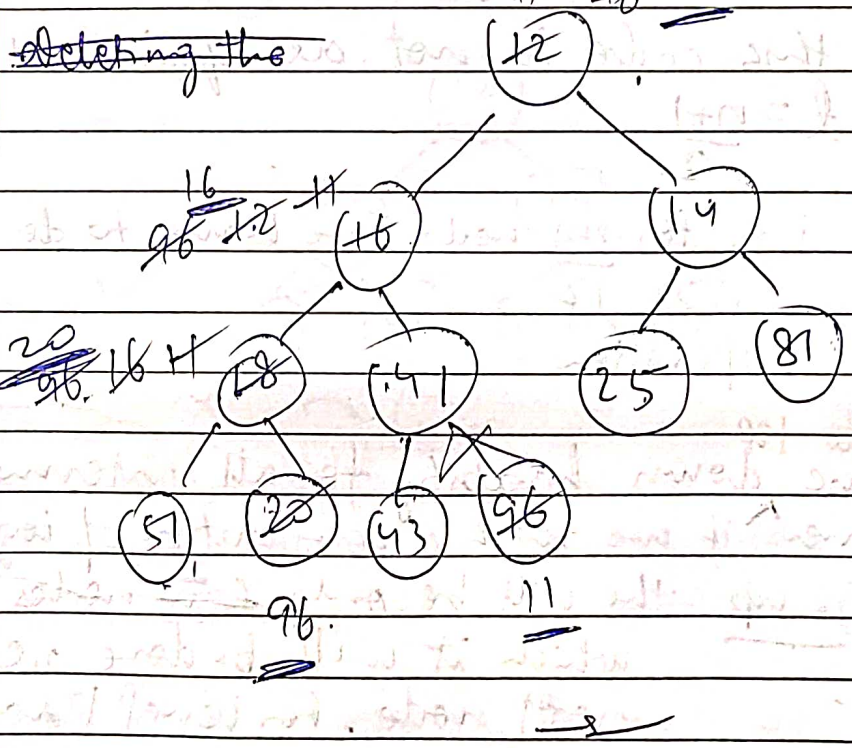
this case is called increase key

this case is called decrease key

$O(\log n)$

Date: / /
Page No.:





Buildheap → Here what we do is we are given an array & we have to construct a binary heap of that. (A)

In Add fn" each time we appended a node & called the bottomup heapify fn".

$n-\left(\frac{n}{2}+1\right)$

$\therefore 2n-n-1 = n-1$

Date: / /
Page No.:

Here the elements are already appended so we will only call the bottom up heapify for each element.

BuildHeap( )

i → 1 to n
BottomupHeapify(i)

But when A is a decreasing sequence then order is $\Omega(n \log n)$

So this code is not as efficient

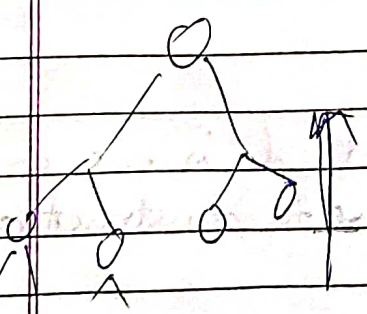
$\because l = \frac{n+1}{2}$

i.e. for $\frac{n+1}{2}$ nodes we have to do logn operation

∴ we do top down heapify for all internal node so even if we reach a computation of logn as we go up the will be only 1 nodes for which it will be done i.e the root node. For level 1 nodes order will be log(n-1). For level 2 nodes order will be log(n-2) & so on.

∴ Now the order of Buildup heapify will be $O(n)$

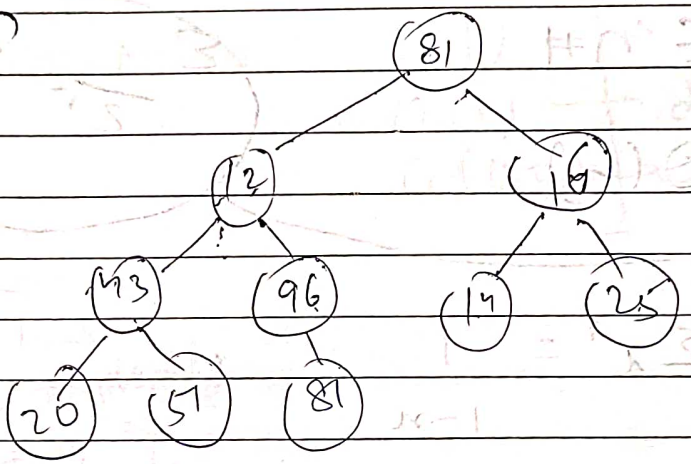
no. of nodes at height i $= \frac{n+1}{2^{i+1}}$

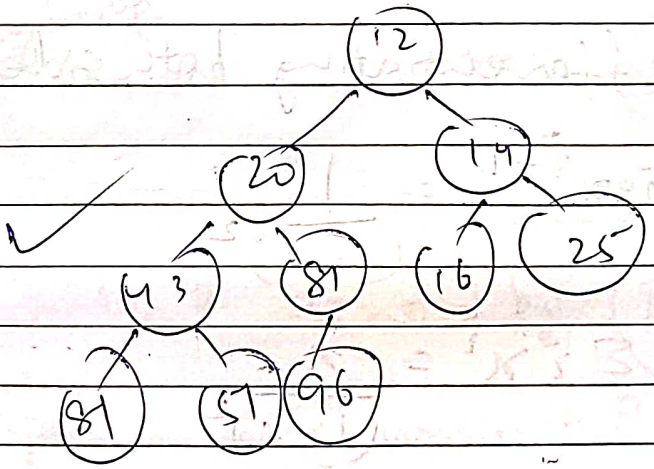


Date: / /
Page No.:

→ BuildHeap( )
$i \rightarrow \frac{n}{2}$ to -1
Topdown Heapify( )

$\theta(n)$





For 2nd code of Buildheap the no. of comparisons done for a node at height i = i & the no. of such nodes at height i = $\frac{n+1}{2^{i+1}}$

Date: / /
Page No.:

$$\Rightarrow \sum_{i=0}^{h} i \cdot \frac{n+1}{2^{i+1}}$$

$$= \frac{n+1}{2} \sum_{i=0}^{h} \frac{i}{2^i}$$

$$\therefore \leq n+1$$

$$\Rightarrow \theta(n)$$

$\star$ $\sum \frac{i}{2^i} \leq 2$

Proof :- $\sum x^i = \frac{1}{1-x}$

differentiating both sides

$$\sum i x^{i-1} = \frac{1}{(1-x)^2}$$

$$\Rightarrow \sum i x^i = \frac{x}{(1-x)^2}$$

$$\Rightarrow \sum i/2^i = \frac{1/2}{1/4} = 2$$

include those 2 heaps in 1 & do buildheap for that

Date: / /
Page No.:

Merge / Union $O(n)$ → In case of Binary Heap

Binomial Heap ~~B.H~~ $O(\log n)$

F.H $O(1)$ → union.

$O(1)$ → Add

$O(1)$ → Decreasekey

$O(\log)$ → Delete

every other operation has the same complexity as that of binary heap

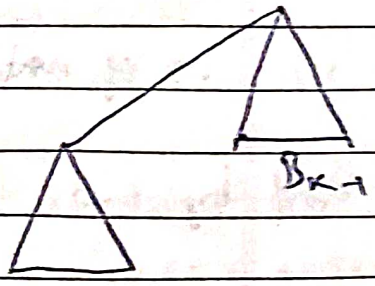
Binomial Tree's

of order k denoted by B_k

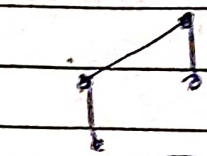
If you want to construct B_k then make 2 copies of B_{k-1} & take one as the child of other one

$k=0$ B_0 → one node

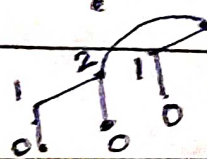
$k=1$ B_1 → 2 copies of B_0 with one as child of other



$k=2$ B_2

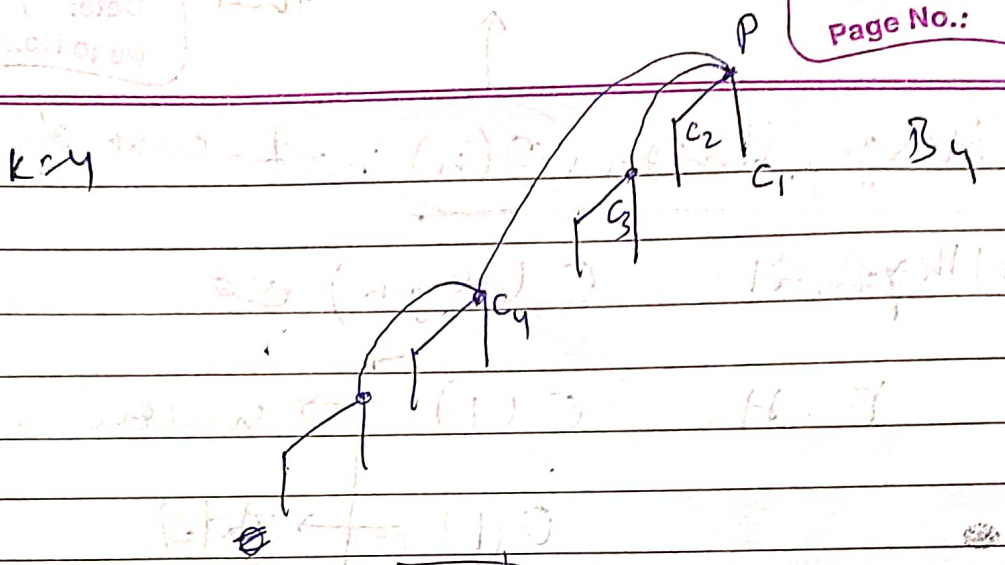


$k=3$ B_3

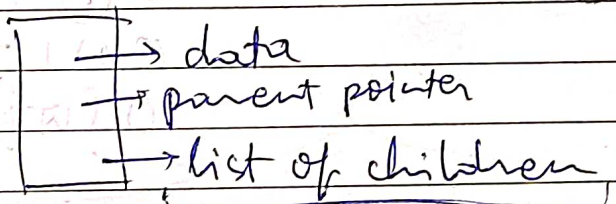


→ The no. written represent the height of each node.

Date: / /
Page No.:



Structure



basically a linked list

for node denoted by P C_1, C_2, C_3 & C_4 are the children stored in a linked list

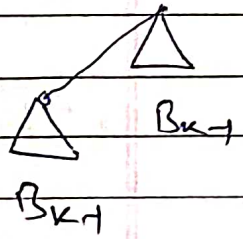
BINOMIAL TREE is NOT the same as binary tree.

Properties of a binomial tree

Prop 1.) no. of nodes in binomial tree → $N(B_k) = 2^k$

Proof by ind. $N(B_0) = 1$

$$N(B_k) = N(B_{k-1}) + N(B_{k-1})$$
$$= 2^{k-1} + 2^{k-1}$$
$$= 2^k$$



Date: / /
Page No.:

Prop. 2 Height

To determine the height of a node take the max. height of all its children & add one to it.

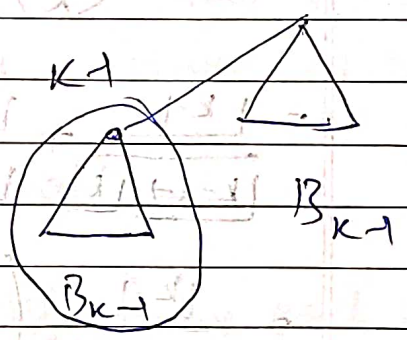
All the leaf nodes ~~with~~ will have height = 0.

Height of binomial tree (B_k) = k

Proof by induction

$H(B_0) = 0$

~~$H(B_0) = 0$~~

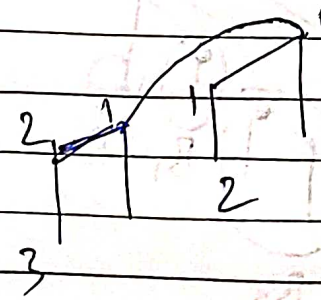


$H(B_k) = 1 + H(B_{k-1})$

$= 1 + k - 1 = k$

Prop 3 Level

Level of a node = (level of its parent node) + 1



→ Levels of each node

~~In a binomial tree~~ of order k the no. of nodes which are at level l

	l	N(k, l)
$^3C_0$ ←	0	1
$^3C_1$ ←	1	3
$^3C_2$ ←	2	3
$^3C_3$ ←	3	1

Date: / /
Page No.:

$$\therefore \boxed{N(k,l) = {}^{k}C_{l} = \binom{k}{l}} \quad \star$$

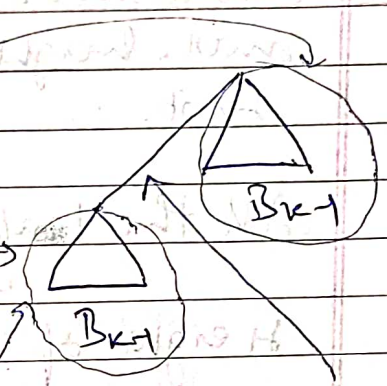
Proof by induction

$$\therefore N(k,l) = N(k-1,l) + N(k-1,l-1)$$

$$= \binom{k-1}{l} + \binom{k-1}{l-1}$$

$$= \frac{\lfloor k-1}{\lfloor k-l-1 \rfloor \lfloor l} + \frac{\lfloor k-1}{\lfloor l-1 \rfloor \lfloor k-l}$$

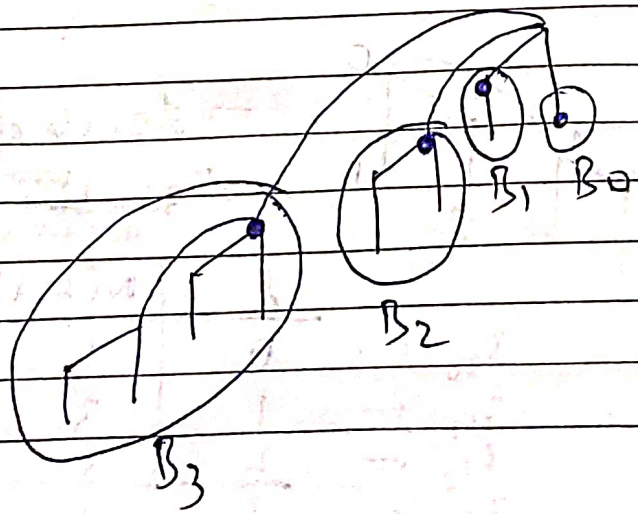
$$= \frac{\lfloor k-1}{\lfloor k-l \rfloor \lfloor l} \left(k-l+l \right) = \frac{\lfloor k}{\lfloor k-1 \rfloor \lfloor l} = \binom{k}{l}$$



When we make this joint level of each node in this tree ↑es by one while level of each node in other tree remains the same.

Prop. 4 The parent node of a binomial tree of order k will have the parents of $B_0, B_1, \ldots B_{k-1}$ as its children

Ex:- B_4

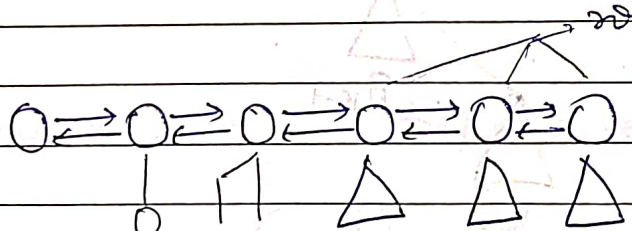


Date: / /
Page No.:

Binomial heap

Prop (1) Collection of Binomial Trees of different order, but for each order k you can have atmost one copy of B_k

Prop (2)



↳ a doubly linked list containing each node, the root of a binomial tree.

∵ B_1 has occurred once it can't occur again. Similarly for others

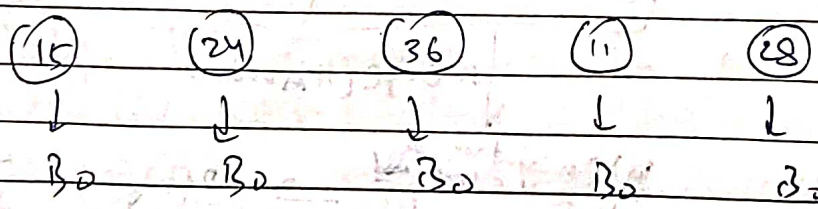
Order ↑es from left to right

Heap property should be satisfied by each node

↓

value in a node ≥ value in its parent node

Ex:-

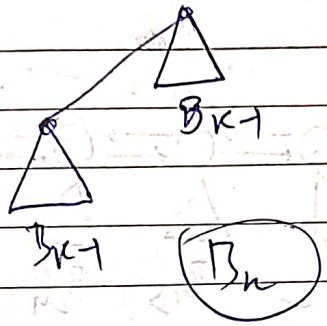


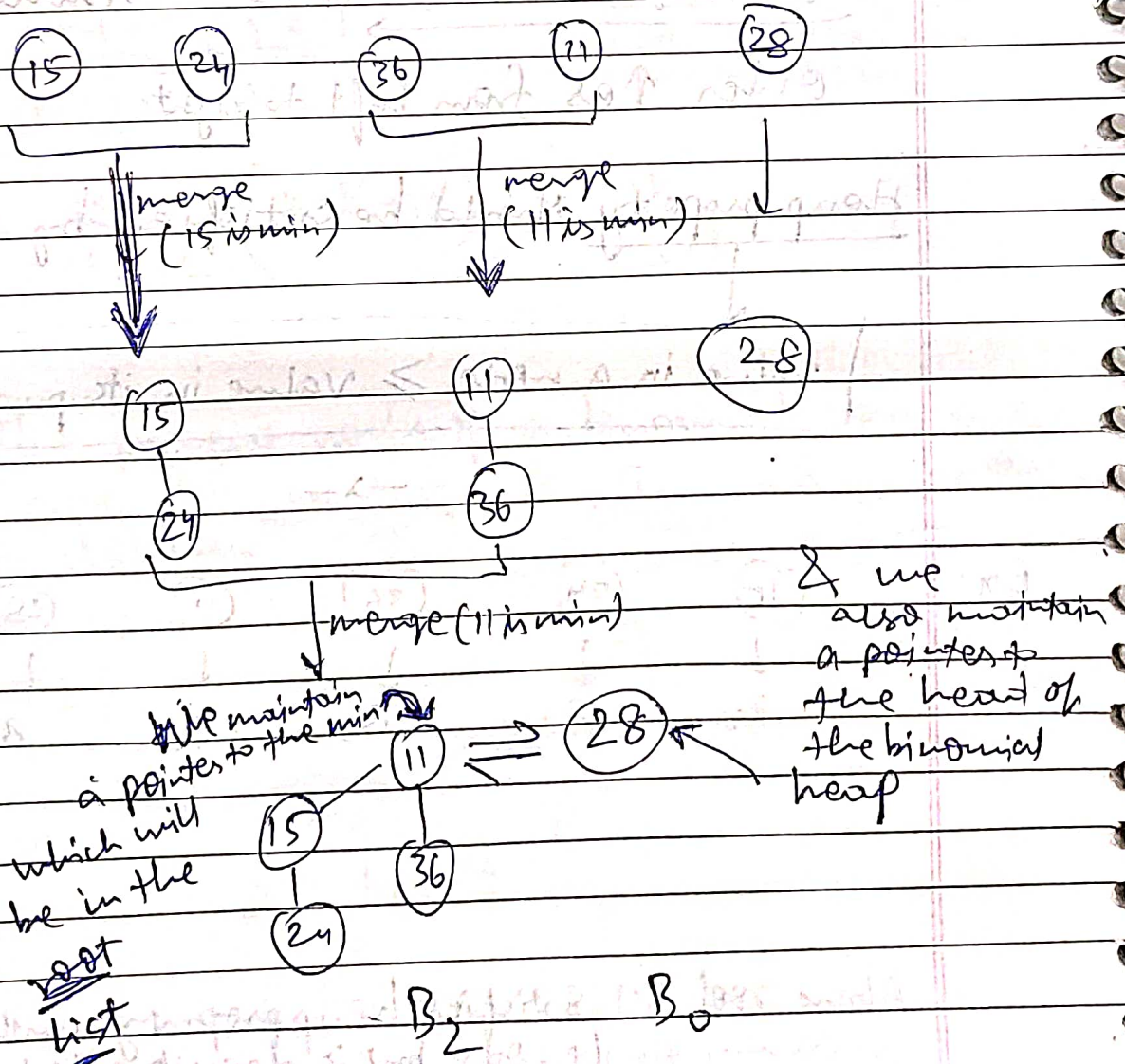
Above root list satisfies heap property with each node containing single node but it doesn't satisfy prop. 1. B_0 is occuring many times.

Date: / /
Page No.:

So what we do is that we take 2 cons. trees & merge them.

when we say merge we have to keep the following dig. in mind





Date: / /
Page No.:

Q. Given any no. n, is it always possible to construct a binomial heap?

Ans Lets look at the binary representation of n.

$$n = \sum b_i 2^i \quad ; \quad b_i = \{0, 1\}$$

whenever $b_i = 1$ take a copy of B_i

B_i will have 2^i elements

& the summation of all of these elements = n

This also tells us that there is a unique representation of collection of binary trees for given n. i.e. if $n = B_0 + B_3 + B_7$ then there can't be any other rep. However there can be variety of B_0's, B_3's & B_7's

We have a build heap fn, given a set of n no. we already got an algorithm to build a binomial heap

The algo. is we make all B_0 make them in a linked list. Now we look at the order of 2 consecutive ones, merge them & make it as one. Just repeat this process, If there is one left out that is the one that is going to be there at the end

i.e. as long as you see 2 binomial trees of same order you merge them.

Date: / /
Page No.:

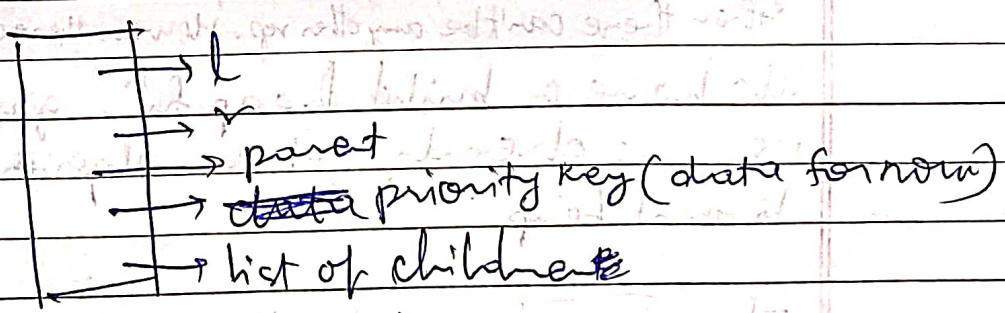
At 1st we look at n nodes, ~~the 2nd time~~
the 2nd time ——— $\frac{n}{2}$ "

3rd " ——— $\frac{n}{4}$ "

$$n + \frac{n}{2} + \frac{n}{4} + \cdots \leq 2n$$

So, this algo will create a valid binomial heap & this will happen in ~~to~~ linear time.

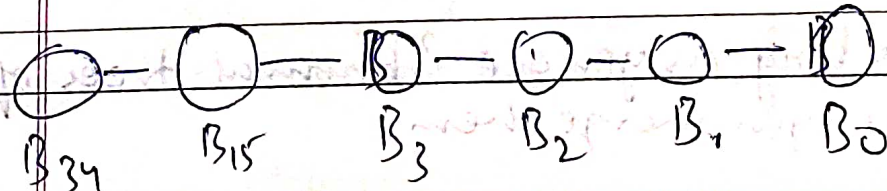
So now, the structure for each node will have 2 more items the left pointer(l) & a right pointer(r) for linked list. If the node is not a root node, those pointers will be NULL



OPERATIONS

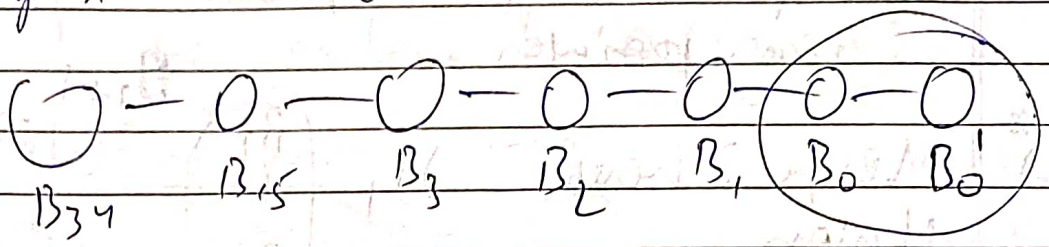
(1) Add a no.

without loss of generality lets assume the root list to be

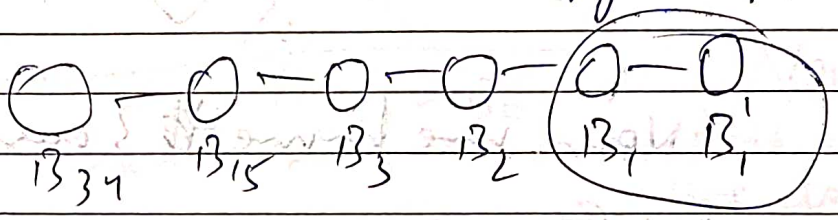


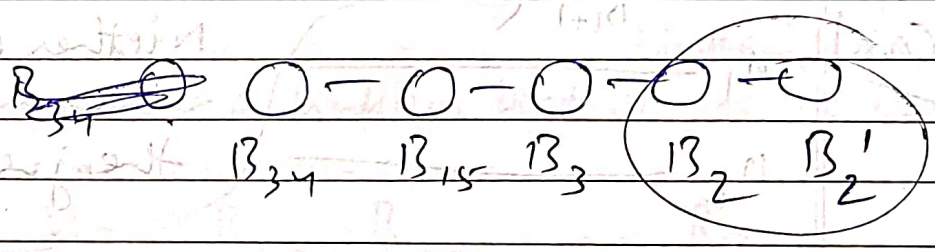
Date: / /
Page No.:

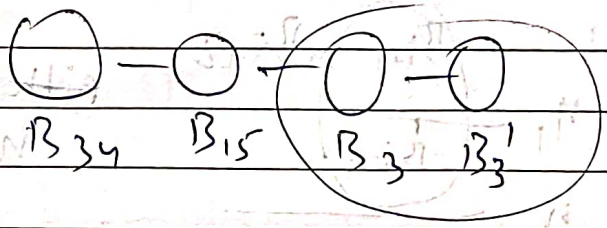
just add the no.

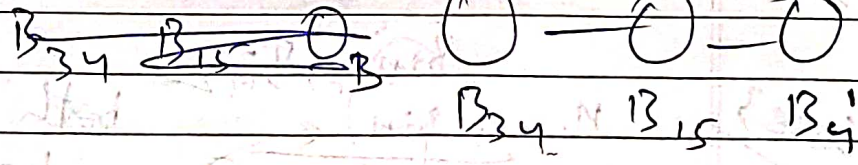


But it doesn't satisfy heap property so merge









Algo ends

Q.) So the complexity of this algo given there are n no. of nodes in the binomial heap.

Q.) How many nodes are there in the root list?

Ans It is equal to the no. of 1's in the binary rep. of n which is at most $\log_2 n$

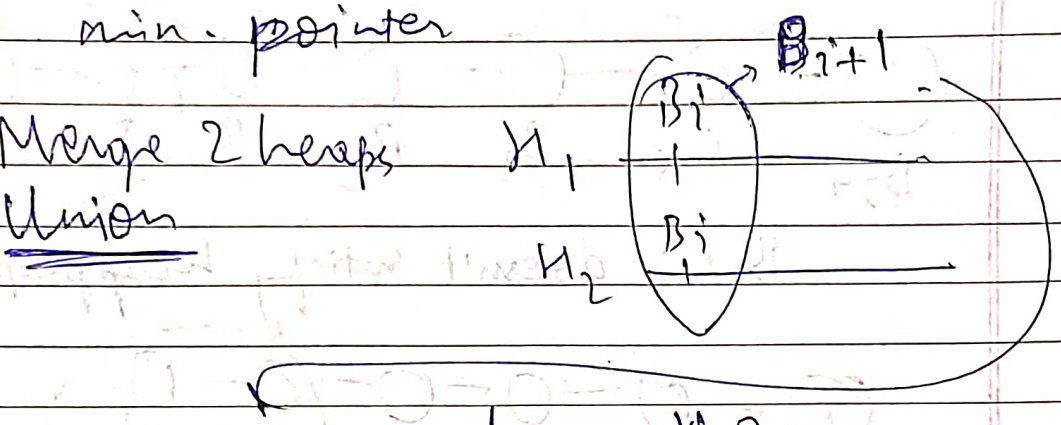
$\therefore$ $O(\log n)$

Date: / /
Page No.:

Keep in mind to update the min. pointer

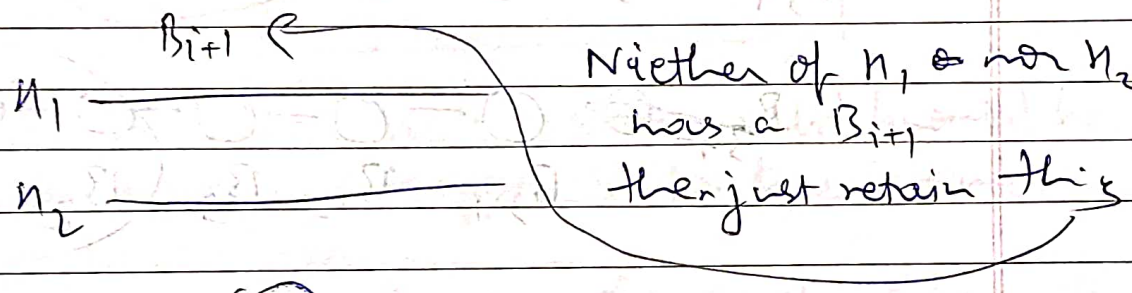
2.) Merge 2 heaps

Union



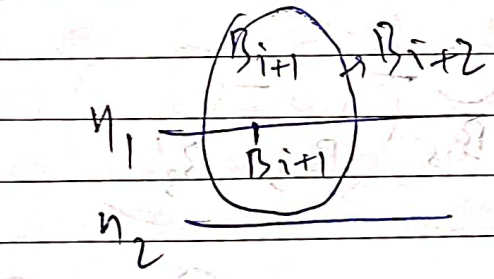
Now we have 3 cases

Case1



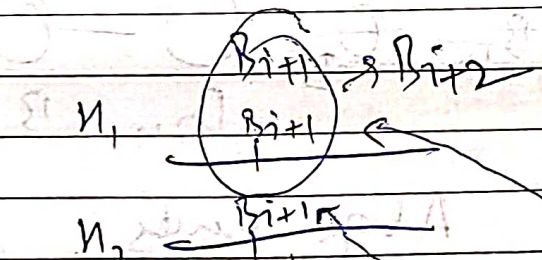
Neither of H_1 nor H_2 has a B_{i+1} then just retain this

Case2



Only one of H_1 or H_2 has B_{i+1}

Case3



both H_1 & H_2 have B_{i+1} merge these 2 & retain this

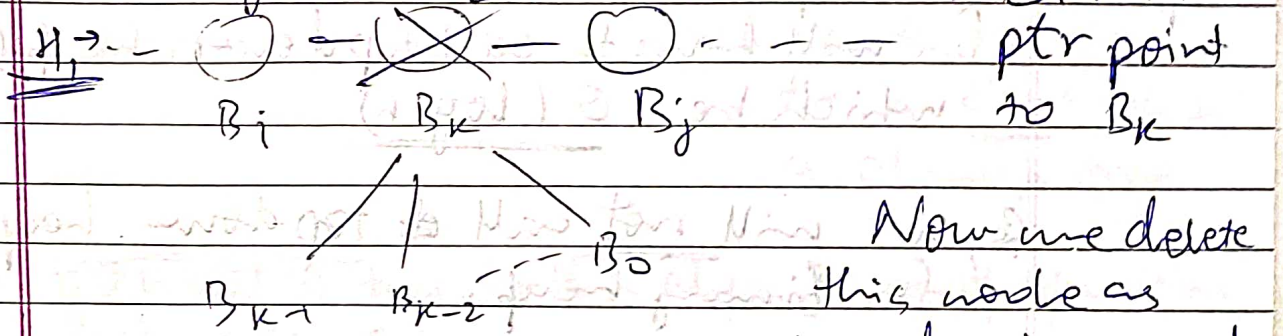
This process is analogous to binary addition

$$\begin{array}{r} 1\ 1 \\ 1\ 0\ 1\ 1\ 0 \\ 1\ 0\ 1\ 1\ 1 \\ \hline 1\ 1\ 1\ 0\ 1 \end{array}$$

Date: / /
Page No.:

If n is the no. of nodes in the heap the ordere is $O(\log n)$

3.) deleteing_min



Now we delete this node as we do in general linked list

make a linked list of its children

$H_2 \rightarrow B_{k-1}\ B_{k-2} - - - B_0$

& merge H_1 & H_2

This will happen in $O(\log n)$

4.) decrease key

decrease the value of the required node & called 'bottomup heapify'

order will be $O(\log n)$

5.) delete a particu. node

decrease the value of that node less than its root node and call decrease_key fun. & the delete the root that tree.

6.) Increase key

~~because the one parent has~~ one parent has multiple children & for increase key we will have to call top down heapify which be $O(\log^2 n)$

So, we will not call top down heapify for binary heap.

∴ for increasing the value of a node we will delete that node & add a new node to the heap with that value.

—X—

AMORTIZED ANALYSIS

We have an algorithm which starts with an empty stack & does n operations where each of the operation is one of the following:-

1.) Push ( ) → O(1) 2.) Pop ( ) → O(1)

3) Multipop(k) → pops k elements from top of the stack
↳ O(k)

Q.) what is the complexity of this Algo.?

Ans. The way we've been analyzing this algorithm is the no. of operation i.e. (n × complexity of costliest of the 3 operations)

Date: / /
Page No.:

$\Rightarrow n \times \theta(k) \rightarrow \theta(nk) \rightarrow O(n^2)$

k can be as bad as 'n'

However, the no. of operation that we do is $\leq n^2$ which is always true

but what is true for this algo. is no. of operations that we do $\leq 2n$

For ex. we can take that we did $(n-1)$ pushes as 1st $(n-1)$ operations, the nth operations being popping $(n-1)$ elements

$\Rightarrow$ order of operations $= (n-1) \times 1 + 1 \times (n-1)$
$= 2(n-1) < 2n$

To explicitly see

no. of pushes $\leq n$

no. of pops $\leq$ no. of pushes $\leq n$
b/c you can't pop an element unless it is pushed

order of operations = no. of pushes + no. of pops
$\leq n$ $\leq n$

$\leq 2n$

Date: / /
Page No.:

There is an alternative way to analyze these algorithms which is called amortized analysis

Amortized analysis

	Actual cost	Amortized cost
Push	1	2
Pop	1	0
Multipop	K	0

~~The way to interp~~

Actual cost

The way to interpret this is lets say that these pushes are very costly operations, you have to do it on the server & every push & every pop, you will be charged ₹1.

i.e. Everytime you want to post you pay ₹1
" " " " pop " " "

Amortized

Now, you ~~have~~ are given an offer

when you want to push you pay ₹2
but the pop is free for you

just like when you buy a datapack, all the calls are free for you.

Date: / /
Page No.:

But during this process the Service provide (SP) should not go bankrupt

So what will happen when you ~~go~~ ask for a push SP will collect ₹2 from you, keep ₹1 with him & pay ₹1 to serve to push. Now when you call a pop, SP uses this ₹1 without charging anything from you.

since the no. of pushes ≥ no. of pops

⇒ the amount of money with SP is never less than 0

It is very important to realize that the stack is empty at 1st. B/c if the stack ~~is bit~~ already has like 1 billion elements at 1st then you can call pop so many times that SP will go bankrupt.

So what this amortized analysis does is that;

looking at Actual cost this k is a problem k in worst case can be n

so what we did is that we moved ~~n~~ from actual cost to amortized cost so that the cost of costliest operation becomes zero.

We are interested in finding the sum of the actual cost but if we want to calc. sum of the amortized cost, it's easy

Date: / /
Page No.:

So what we will do here is that

	Actual cost	Amortized cost
Push	1	2
Pop	1	0
Multipop	1	0

c_i — actual cost of ith operation

$\overline{c_i}$ — Amortized cost of ith operation

so what we see here is

$$\overline{c_i} \leq 2$$

So, it's easy for us to find the sum of Amortized costs

$$\sum_{i=1}^{n} \overline{c_i} \leq 2n$$

The way we should define amortized cost is s.t.

$$\sum_{i=1}^{n} c_i \leq \sum_{i=1}^{n} \overline{c_i}$$

Date: / /
Page No.:

i.e. given this actual cost we have to define an amortized cost s.t. sum of the actual cost is less than or equal to sum of the amortized cost.

finding sum of actual cost is difficult for us (kn). So if we can define amortized cost in a way mentioned above

we can say sum of the actual cost $\leq 2n$
∵ finding the sum of amortized cost is easy

—x—

(Q.) Given the actual cost, how do we come up with amortized cost with above mentioned properties)

—x—

~~The~~ Lets consider the general framework when this ~~co~~ actually happens, it doesn't happen all the time. How did we get this $\{2, 0, 0\}$ from this $\{1, 1, k\}$?

There's something called a potential fn^n (ϕ)

ϕ is defined from a data structure (D) to R^+

i.e. $\phi : D \rightarrow (R^+)$ → set of ~~value no~~ non negative no.

This potential fn^n takes a data structure as input & gives us a value

Date: / /
Page No.:

Now, the amortized cost is defined in terms of the potential fun^n.

the potential fun^n for the ith operation ($\bar{C}_i$) is defined as :-

→ will always be non (-ve).

$$\bar{C}_i = C_i + \phi(D_i) - \phi(D_{i-1})$$

$\phi(D_i)$ → difference b/w potential → can be (-ve)

D_i is the data structure that we have after the ith operation

D_{i-1} is the data structure that we have before the ith operation

$$\Rightarrow \bar{C}_1 = C_1 + \phi(D_1) - \phi(D_0)$$

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

$$\bar{C}_3 = C_3 + \phi(D_3) - \phi(D_2)$$

$$\vdots$$

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

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

If we can defined ϕ

s.t. $\phi(D_n) - \phi(D_0) \geq 0$ — (1)

Date: / /
Page No.:

then $\sum c_i \leq \sum \bar{c}_i$ — (2)

Now,
One way to ensure that (1) is true is, we to ensure that when we start the operation we ensure that the ϕ fn that we have is 0. & after that if we ensure that $\phi \geq 0$

then (2) becomes true.

Lets do this for the stack operation.

	Actual cost	Amortized cost
Push	1	2
Pop	1	0
Multipop	k	0

ϕ fn for the stack that we started with

ϕ: The no. of elements in the stack.

i.e. the ϕ fn will take a data structure (stack) & will give the no. of elements in the stack (value)

So, ∵ we start with an empty stack.

$\Rightarrow \phi(D_0) = 0$

Now, we look at the ith operation which can be any of 3 mentioned in the table above

Date: / /
Page No.:

Lets look at the amortized cost of push

push → (1) + (1) = 2

(1): Actual cost; (1): change in potential Φ

pop → 1 + (−1) = 0

Multipop → k + (−k) = 0

so it doesn't matter what is the ith operation.

$\hat{c}_i \leq 2$ ✓

$\Rightarrow \sum \hat{c}_i \leq 2n$

but we know that $\sum c_i \leq \sum \hat{c}_i$

$\Rightarrow \boxed{\sum c_i \leq 2n}$

Moral of the ~~story~~ story → If we have to do amortized analysis, we have to define a potential fn. The role of the potential fn is, if the actual cost is very small, its potential increases marginally. For ex:- in the push, it's an easy operation, it's cost is only 1., we define a potential fn, which ~~Pes~~ the potential by 1 or 2 it's fine but what should happen is that if we do a costly opern (here it's multipop) the potential should go down

Date: / /
Page No.:

considerably so that the potential difference will compensate for the costly operation that we are doing

—x—

We will implement this analysis on some more fn.

One of which being the binary increment fn

We have a binary no. which we want to increment every time. That binary no. can be of string length 105.

```
BI(A)
    i = strlen(A)-1
    while (A[i] == '1')
        A[i--] = '0';
    A[i] = '1';
```

If the binary input represent a decimal no. n then the while loop can atmost run up to an order of log n.

Now, suppose our binary input is 0000000000. These no. of bits can store upto 1023

So it's complete valid to call (BI) fn for this input 512 times

for each time the order can be atmost $(\log_2 512)$

⇒ Order of this code will be $512 \log_2 512$

Date: / /
Page No.:

Similarly if we call the BI fun^n n times for an x input binary of 0's s.t these ~~binary input~~ many bits of the binary input can accomodate n

then the order of the code will be ~~n log n~~ atmost $O(n \log n)$

strlen = 12 strlen = 8

x = 0 0 0 0 0 0 0 0 0 0 0 0 ; n = 1 1 1 1 1 1 1 1

x can accomodate n

indices → 0 1 2 3 4 5 6 7 8 9 10 11

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

0 0 0 0 0 0 0 0 0 0 0 1

1 0

1 1

⋮

0 0 0 0 1 1 1 1 1 1 1 1

$\rightarrow O(n \log n)$

When we perform ~~this~~ these operations the digit at 11th index changes (i.e. from 0 to 1 or from 1 to 0) n times

While the digit at 10th index changes $\frac{n}{2}$ times

Similarly ——— 9th ——— $\frac{n}{4}$ times

⋮

1

1

just to take the upper bound we take this as n

Date: / /
Page No.:

$$\Rightarrow \sum_{i=0}^{n} \frac{n}{2^i} = n\left(\frac{2 - \frac{1}{2^{n-1}}}{}\right) \leq 2n$$

~~$\Rightarrow$ It's order will be $O(n)$~~

$\Rightarrow$ The total no. of flips that we do $\leq 2n$

Now, lets analyze this by amortized analysis

		no. of flips	no. of 1's	
C_i	$0 \rightarrow 1$	1	+1	= 2
	$1 \rightarrow 0$	k	k-k	= 0
		Actual cost		Amortized cost

$\boxed{\Phi = \text{No. of 1's}}$

$\Rightarrow \boxed{\bar{C}_i = 2}$

$$\Sigma C_i \leq \Sigma \bar{C}_i = 2n$$

$\Rightarrow \boxed{\Sigma C_i \leq 2n}$

binary increment

In this case, (NOT in every case) $\Phi(D_i) > 1$ (strictly greater than 1) $\forall\ i > 0$

$\Rightarrow \Sigma C_i < 2n$ (strictly smaller than $2n$)

$\therefore$ the total complexity of this algo is $\Theta(n)$

Date: / /
Page No.:

(*) When building a binary heap using Add fun; with n inputs

B.H	
i → 0 to n Add(i)	Ω(n log n)

b/c there will be atleast $\frac{n+1}{2}$ nodes for ~~wit~~ which these order for Add will be log n. Hence, we need an alternate algo for building binary heap

~~However~~ However, we can use the above algo to build a binomial heap in linear time

Consider the analogy

$B_7\ B_6\ B_5\ B_4\ B_3\ B_2\ B_1\ B_0$
0 0 0 0 0 0 0 0

Binary increment is perfectly analogous to the Binomial heap add fun

~~Note~~ : since order of BI is O(n)
⇒ " " Build binomial heap is also O(n)

—x—

~~Eyman~~

Date: / /
Page No.:

→ Dynamic Table

You are given n numbers which you have to add to an array. The problem is that 'n' is not known from the beginning and we don't want to allocate 10^5 size of array for each time the input is given. Here comes the dynamic table into play.

Suppose the inputs are $a_1\ a_2\ a_3\ a_4\ a_5\ a_6\ a_7\ a_8\ a_9\ a_{10} \cdots a_n$

when a_1 is given
you form an array of size 1 & store a_1 [a_1]

when a_2 is given
you form an array of size 2×2^0, copy a_1 to current array, delete the previous array & add a_2 to current array.

when a_3 is given
you form an array of size 2×2^1, copy the elements of previous array, delete the previous array & add a_3 to current array.

when a_4 is given
you just add it to current arr.

when a_5 is given
you form a array of size 2×2^2, copy the elements of previous array, delete the previous arr & add a_5 to current arr.

Date: / /
Page No.:

for a_6, a_7, a_8 you just add the similarly you go on doing a for all n's.

Ex:- [1] × arr1

[1|2] × arr2

[1|2|3] arr3

[1|2|3|4] × arr3

[1|2|3|4|5| | | |] arr4

[1|2|3|4|5|6| | |] arr4

Now, the claim is that almost space taken by the array will be 2n while the time complexity $\leq 3n$

$$c_i = \begin{cases} 1 & \text{if } i \neq (2^k+1) \\ i & \text{if } i = (2^k+1) \end{cases}$$

for this example

$$\sum_{i=0}^{\log n} c_i = n + \sum_{i=0}^{\log n} 2^k$$

Date: / /
Page No.:

$$\Rightarrow \sum c_i = n + \sum_{i=0}^{\log n} 2^i \leq n + 2^{(\log n)+1} - 1$$

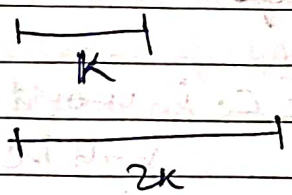
$$= n + 2n - 1 \leq 3n$$

Now, lets do the amortized analysis of it

ϕ = (no. of filled slots − no. of empty slots) in the array

Let no. of filled slots be x & y

	Actual Cost	Amortized Cost
Just Adding	1	1 + 2 = 3
Resizing & Adding (Costly operation)	k+1	k+1 + [k+1−(k−1)−k] = k+1 + (−k+2) = 3



$\bar{c}_i = 3 \quad \Rightarrow \sum c_i \leq \sum \bar{c}_i \leq 3n$

$$\Rightarrow \boxed{\sum c_i \leq 3n}$$

Date: / /
Page No.:

There can be multiple ϕ's that can be chosen to find the order, which satisfy the conditions of amortized analysis.

So, any ϕ can be defined as

ϕ: (2x no. of elements) − (size of the arr)

	Actual cost	Amortized cost
Just adding	1	1+2 = 3
Resizing & adding	k+1	$k+1+[2(k+1)-2k-[2k-k]]$ = 3

Amortized cost is not an average operation. This is infact the worst case, we are not taking average

which mean $\sum C_i$ in worst case will be $\leq$ Result which comes

Date: / /
Page No.:

FIBONACCI HEAP

Operations	Linked list	Binary Heap	Binomial Heap	Fibonacci Heap
insert	1	log n	log n	1
delete-min	n	log n	log n	log n
decrease-key	n	log n	log n	1
delete	n	log n	log n	log n
union	1	n	log n	1
find min	n	1	1	1

Everything written in the last column (Fibonacci heap column) is the amortized cost. These are not the actual cost

What it means is if you do a sequence of operations, for example if you do a k_1 insertions, k_2 deletions & k_3 decrease keys

then the complexity of algo. is:-

$$k_1 + k_3 + k_2 \log n$$

but if you start at just one decrease key or one delete min, it could be as bad as order n.

Date: / /
Page No.:

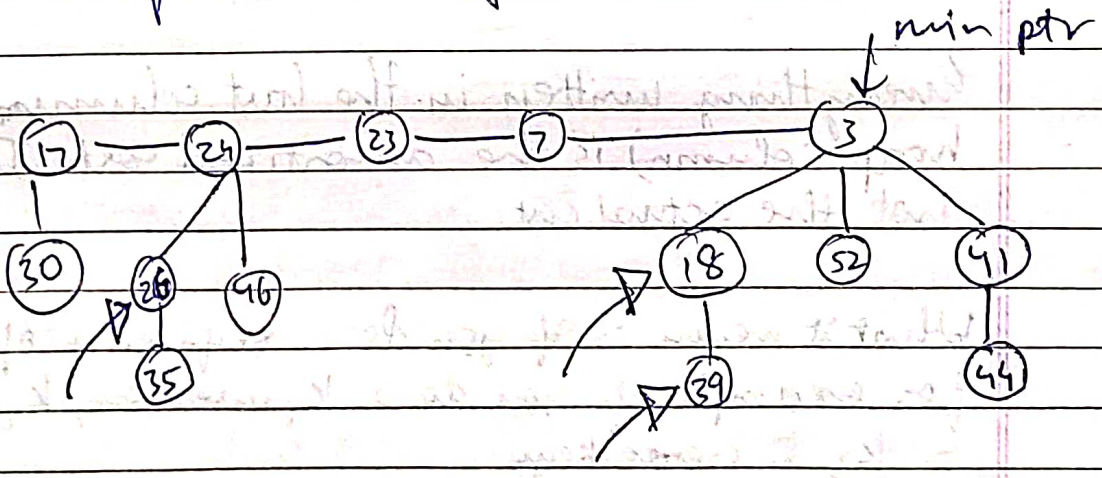
We're going to do amortized analysis of fibonacci heaps.

Fibonacci heaps are also a collection of trees where each node can have multiple children. ~~Heap~~ They also follow heap property

→ Similar to binomial heaps, but less rigid structure

- Binomial heap: eagerly consolidate trees after each insert
- Fibonacci heap: lazily defer (postpone) consolidation until next delete-min.

Ex:- of fibonacci heap



→ shows marked nodes

the structure of each node will contain a boolean item called marked nodes which will be either 0 (NOT marked) or 1 (marked)

Date: / /
Page No.:

Any node can have any no. of children. We only need to take care of one ptr. (the min ptr)

(*) Notations

i) n = no. of nodes in heap.
ii) rank(x) = no. of children of node x.
iii) rank(H) = max. rank of any node in heap H.
iv) trees(H) = no. of trees in heap H.
v) marks(H) = no. of marked nodes in heap H

For ex: given in previous pg

trees(H) = 5 marks(H) = 3 n = 14
rank(H) = 3

vi) level → level of root node is zero.
" " child is (1+ level of the parent)

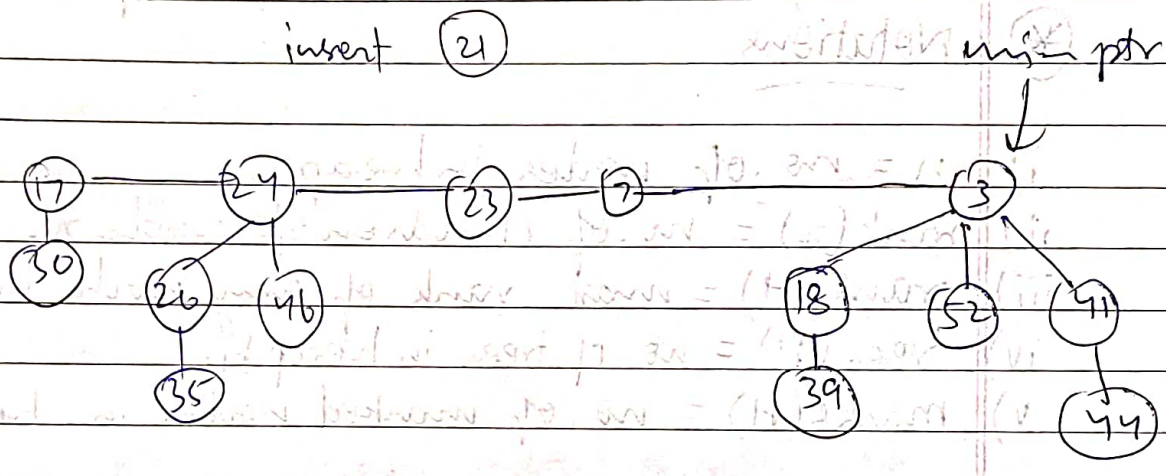
vi) height → height of a leaf node is 0
" " any other node is
[1 + max. (height of all its children)]

vii) height of a tree → height of root node
viii) " " a heap heap = max. (height of all trees)

For Fibonacci heap we define the potential function as:-

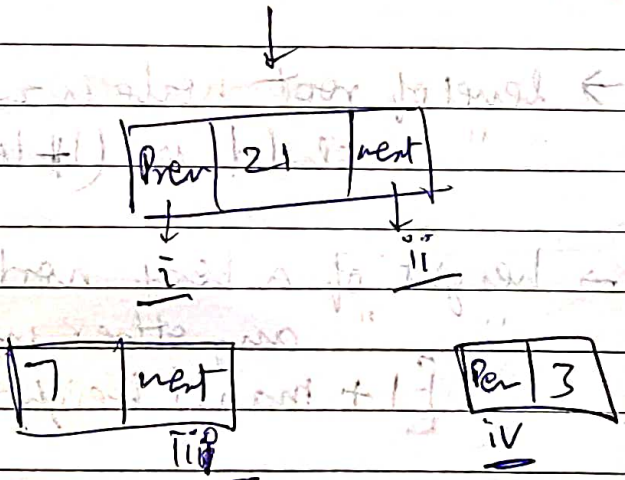
$$\boxed{\phi(H) = \text{trees}(H) + [2 \times \text{marks}(H)]}$$

1.) Insert



∵ we have been given only one ptr (min ptr), we will insert (21) adjacent to it

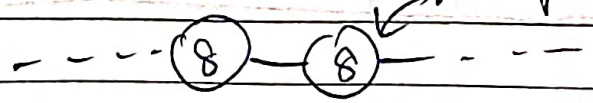
Now, the no. of operations (pointer changes) done will be 4.



∴ Actual cost for insertion = 4

Amortized " " " = 4 + 1 = 5

∵ duplication is acceptable in fibonacci, there can be a situation as follows:-



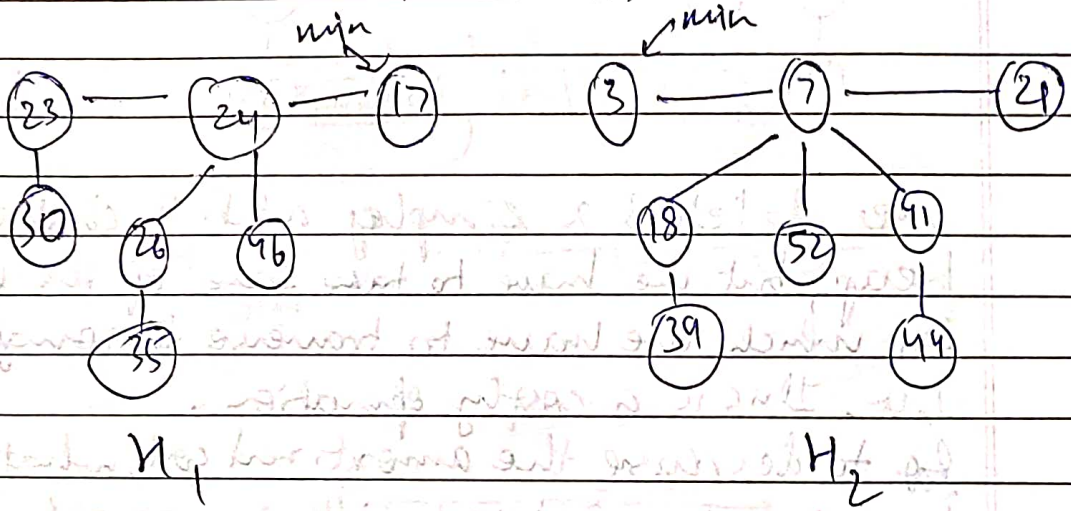
Now, when we say delete ~~min~~, it's ~~okay~~ to delete one of the min.

Date: / /
Page No.:

2.) Unions

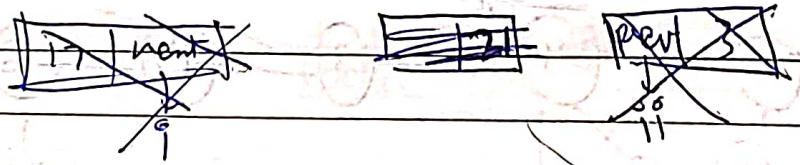
Combine 2 fibonacci heaps.

Representation :- Root lists are circular, doubly linked lists.



Just merge like you merge 2 linked lists.
& take care of the final min ptr.

the no. of operations (pointer changes) = 4.



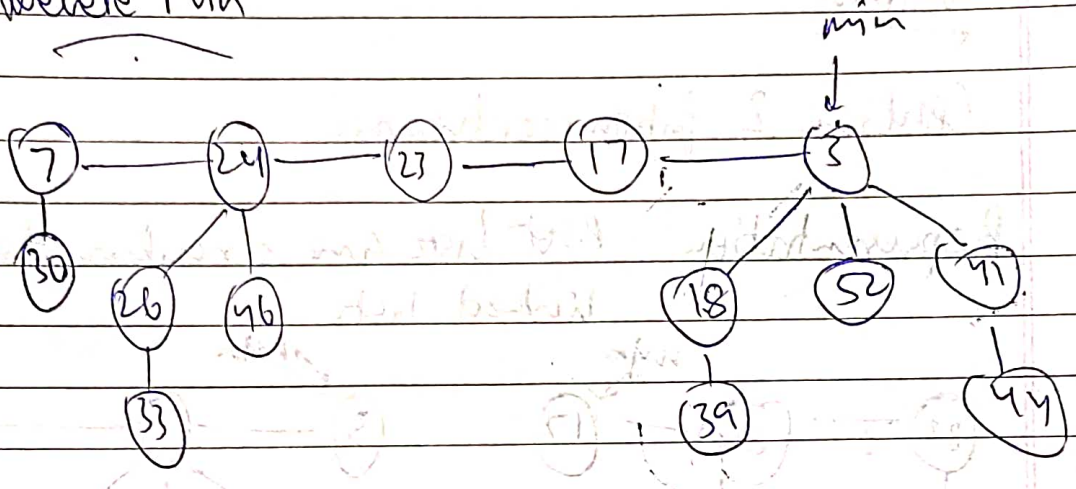
change in potential = 0

∵ no. of trees before = no. of tree after
no of marked nodes before = no. of marked nodes after

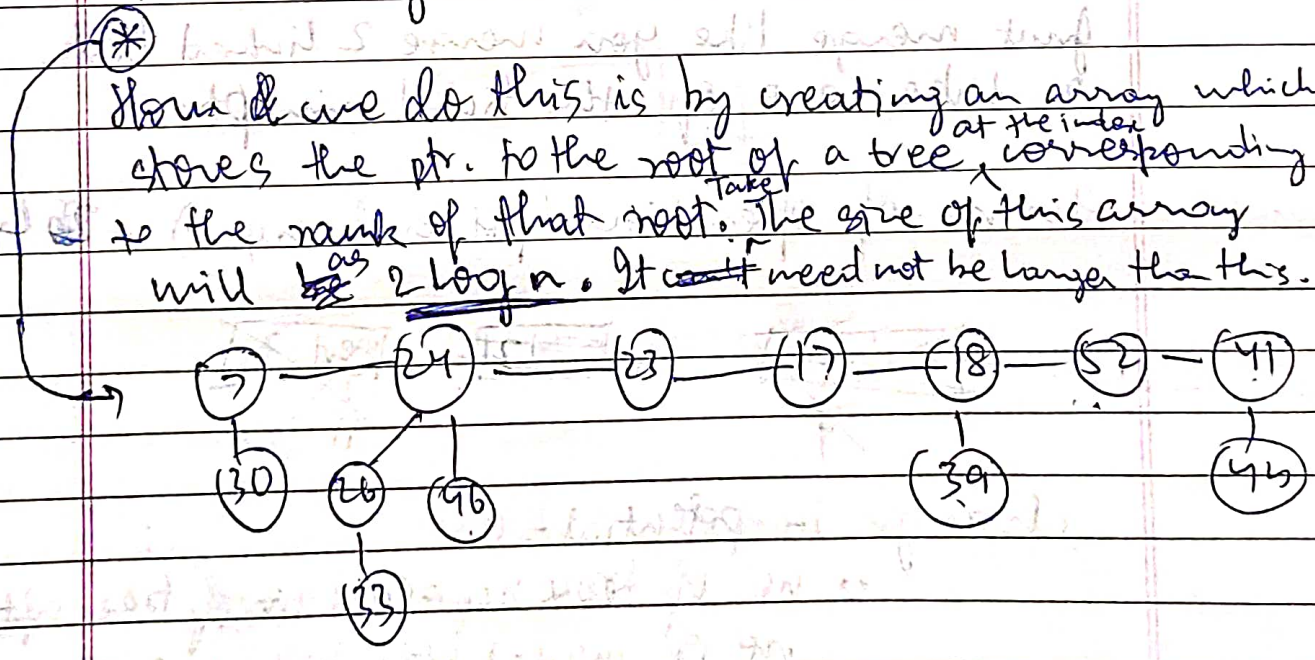
∴ Amortized cost = actual cost
= order 1

Date: / /
Page No.:

3.) Delete Min



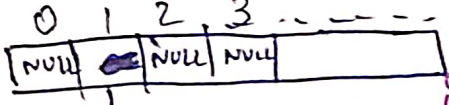
We delete 3 & simply add its children to the heap, but we have to take care of the min ptr for which we have to traverse through the list. This is a costly operation.
So to decrease the amortized cost, what we do is we merge 2 trees with same rank.

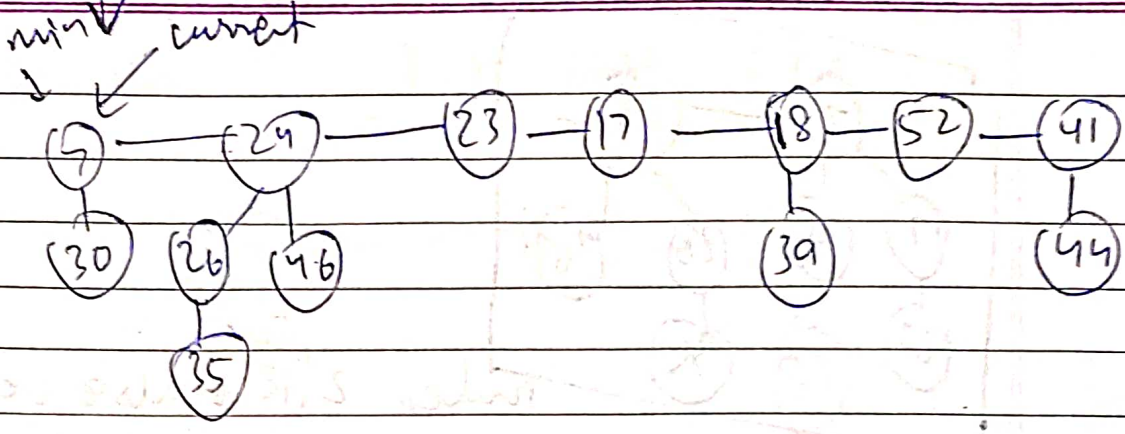


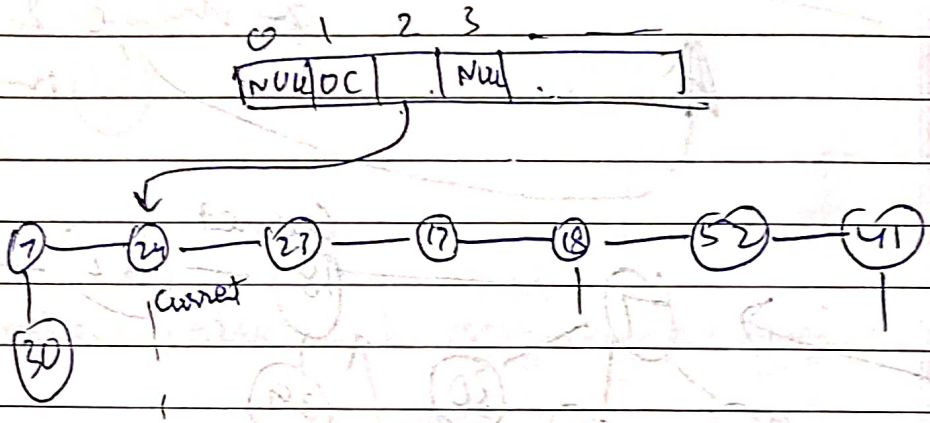
Now how we do this is by creating an array which stores the ptr. to the root of a tree at the index corresponding to the rank of that root. Take the size of this array as 2 log n. It need not be larger than this.

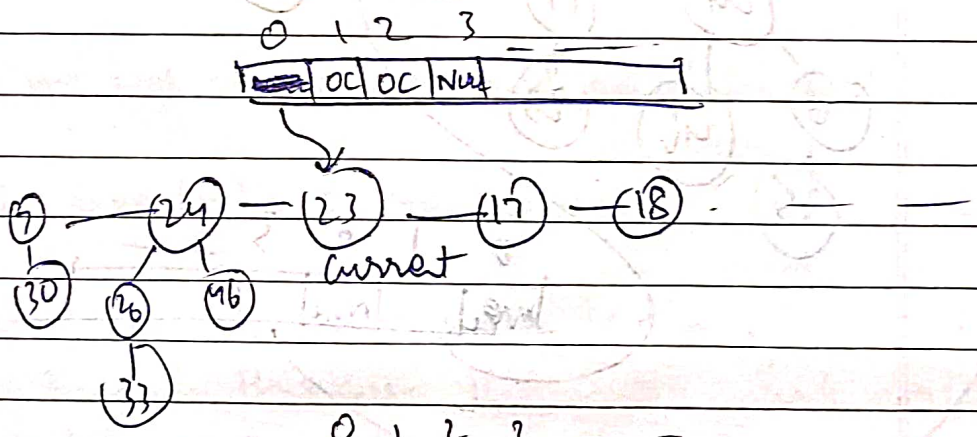
OC → occupied

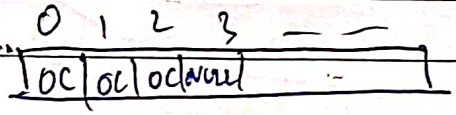
Date: / /
Page No.:

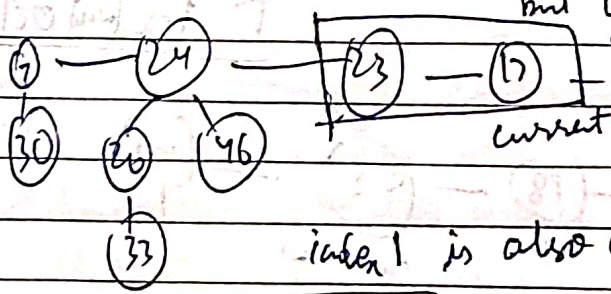




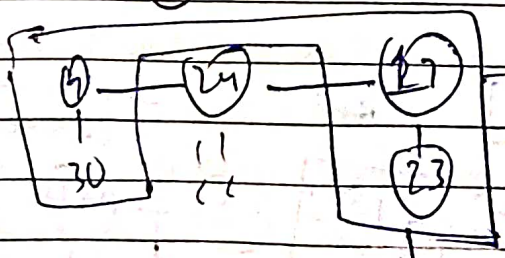






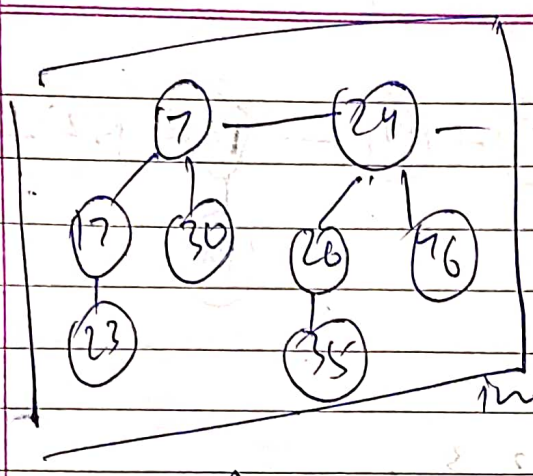


index 1 is also occupied

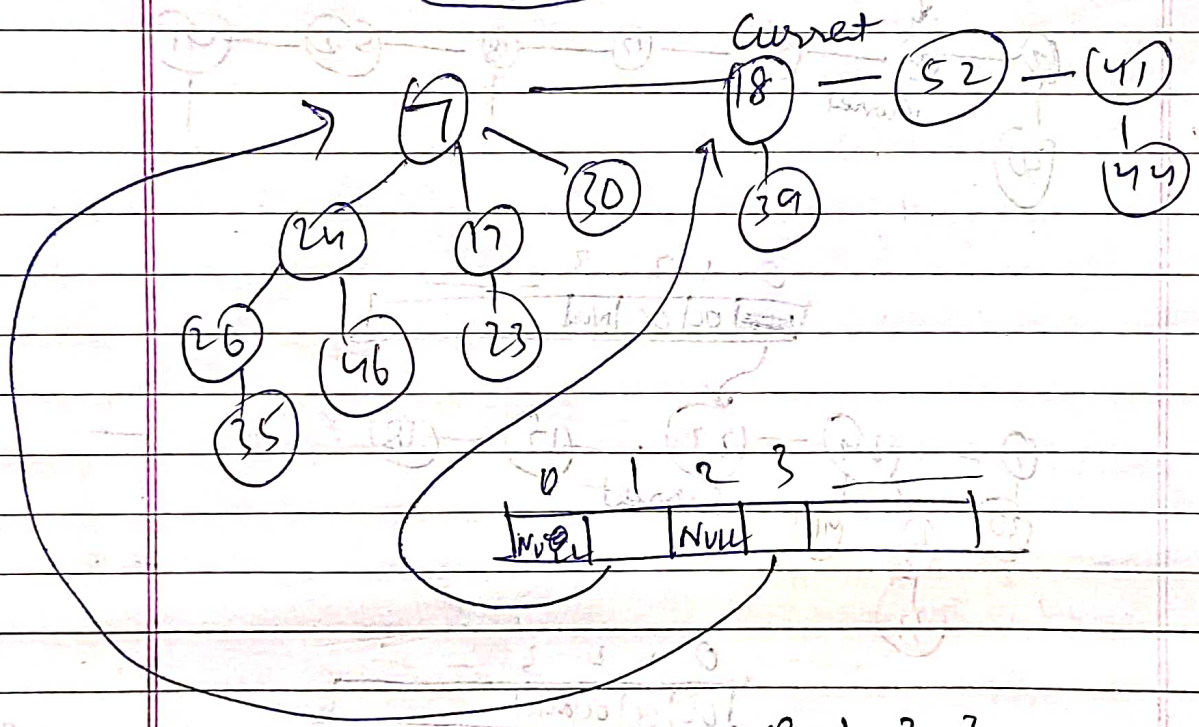


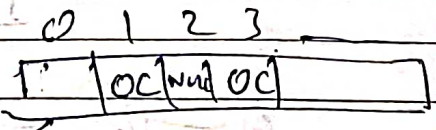
merge & make index 1 NULL

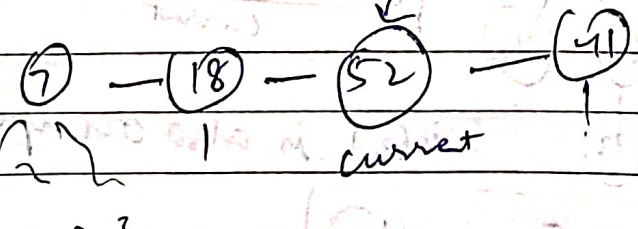
Date: / /
Page No.:

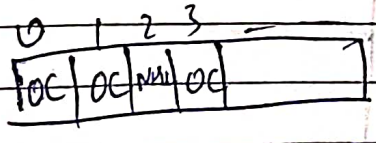


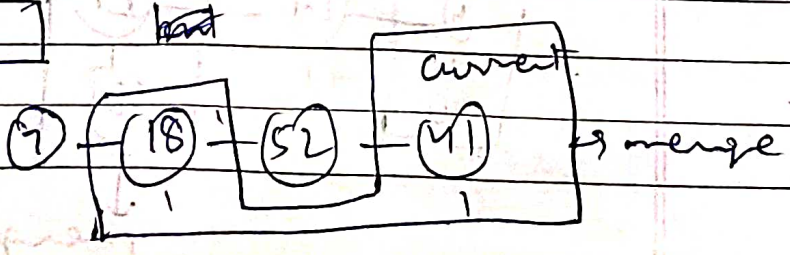
index 2 is ~~also~~ occupied
~~so~~ merge & make index 2 NULL







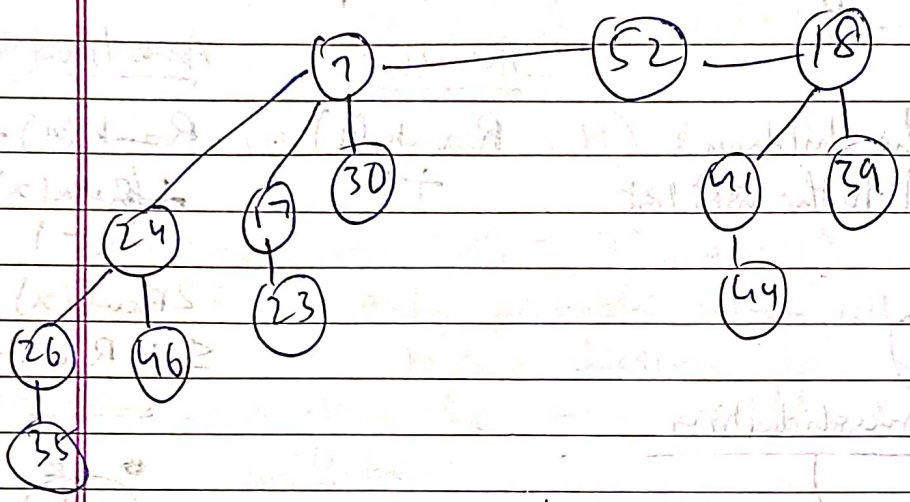




but 1 is ~~also~~ OC ~~so~~ make merge & make 1 NULL

Date: / /
Page No.:

Finally the F.H looks like



Marked nodes can be anywhere (even in the root list)

Now, we look at the amortized analysis of it.

$\phi(H) = trees(H) + 2 \cdot marks(H)$

In this case we don't have to worry about mark node, since in this case the no. of mark nodes don't change too much, what changes is the no. of trees.

If we are deleting a node which was marked then the no. of marked nodes reduce by 1, that will reduce the potential which is a good thing. If at all the potential reduces, it's a good thing, the amortized cost is ~~a~~ actually smaller.

But, if the node that we are deleting is not a marked node the marks(H) will not change, so the only thing that we have to worry about is the no. of trees.

$x + Rank(x) - x - 1$

Date: / /
Page No.:

∴ For a delete_min potential fⁿ analysis we'll just look at the no. of trees.

	Actual cost	Amortized cost
(I) → Cut the children & add to the root list	Rank(x) +1	Rank(x) +1 + (Rank(x) − 1) = 2Rank(x) ≤ 2 Rank(H) → (A)

Delete-min (If x is the node that we're deleting)

(II) consolidation

~~Merging the trees~~ merge (looking up the table and merging)

~~looking up the table~~ non-merge (looking up the table & ~~moving~~ storing the pointer to that root node) → this pointer should not get out of the table in further operations. The operation which is not a non-merge is merge.

Actual cost: we'll treat it as 1

Amortized cost: 1 + (−1) = 0

Total no. of non merge operations that you do will be exactly equal to the no. of trees at the end of all operations i.e. = Trees(H') ≤ Rank(H) + 1

Date: / /
Page No.:

H' is the final Heap

$$\text{Trees}(H') \leq \text{Rank}(H)+1 \quad \rightarrow (B)$$

Pf:- $\text{Trees}(H') \leq \text{Trees}(H)$

when $\text{Trees}(H') = \text{Trees}(H)$

only possible when all the trees in H are distinct.

then the max. possible trees in H will be →

T_1 T_2 T_3 — — — $T_{\text{Rank}(H)+1}$

Rank 0 Rank 1 Rank 2 Rank Rank(H)

i.e $\text{Trees}(H) \leq \text{Rank}(H)+1$

$\Rightarrow \text{Trees}(H') \leq \text{Rank}(H)+1$

∴ Total complexity of delete-min $O(\text{Rank}(H))$

→ (*)

Pf:- ⇒ Total no. of operations possible in delete-min $\leq 3\,\text{Rank}(H)+1$ (from (A) & (B))

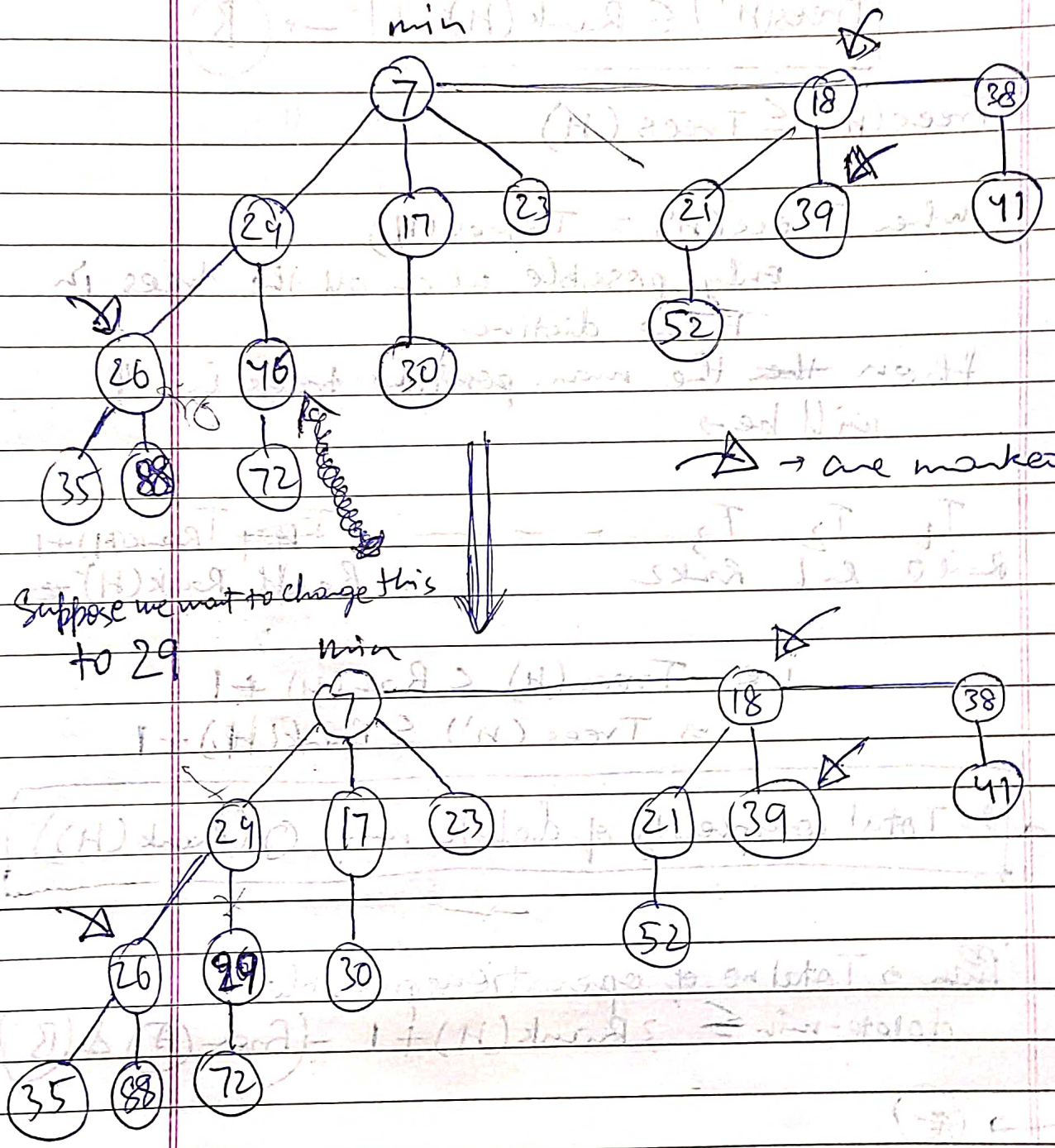
$\therefore O(\text{Rank}(H)) = O(\log n)$

In Fibonacci heap height can be as bad as 'n' So topdown & bottup; both cannot be called.

Decrease Key

Date: / /
Page No.:

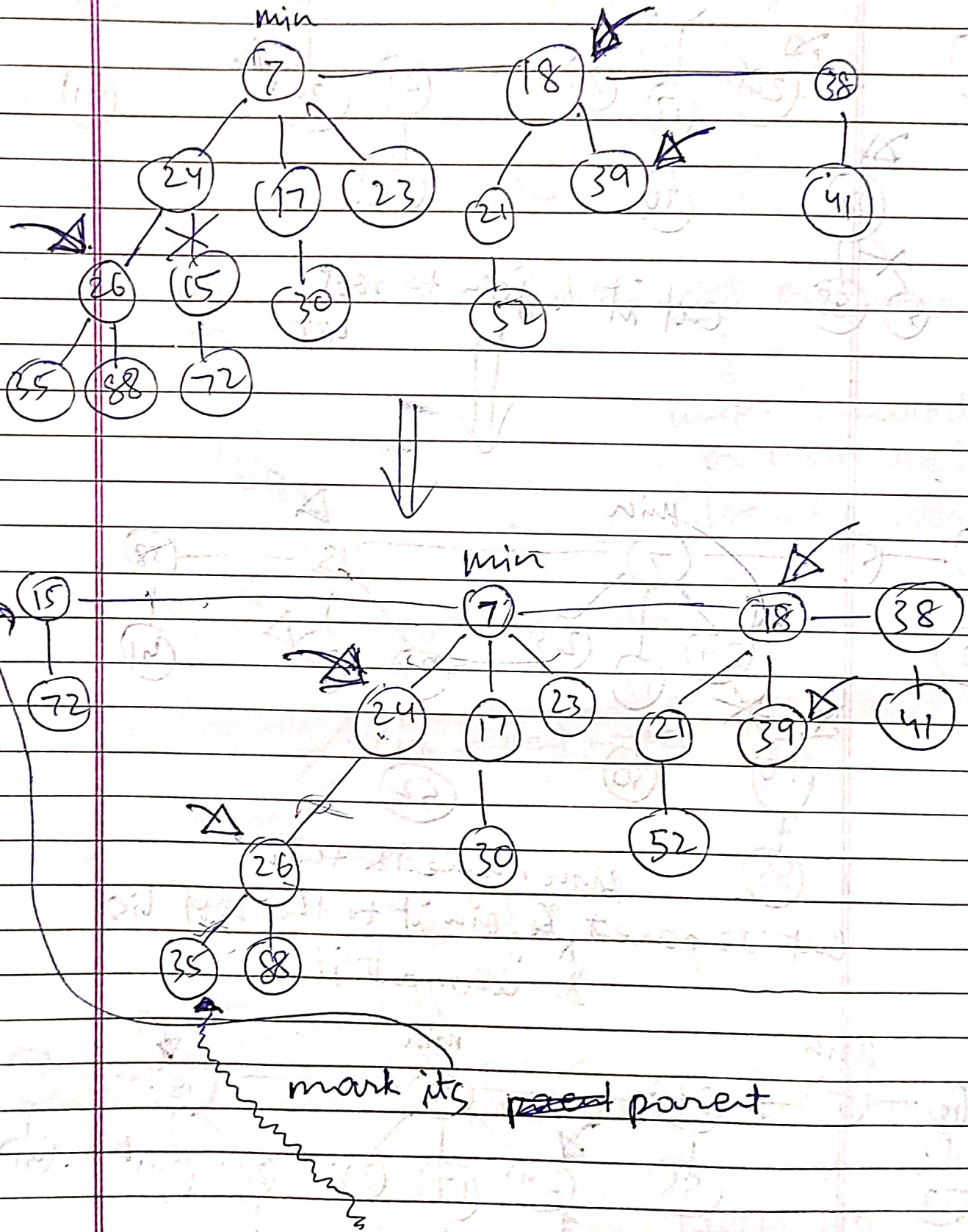
→ Marked nodes indicate that one of the children have been cut in the past



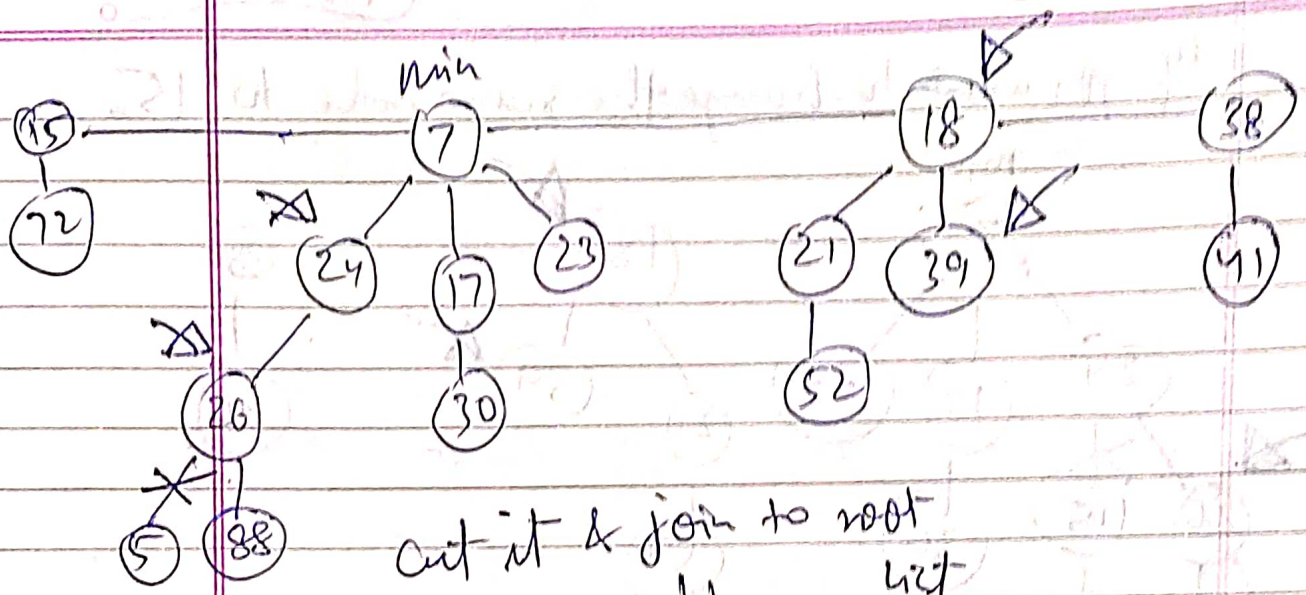
& stop

Date: / /
Page No.:

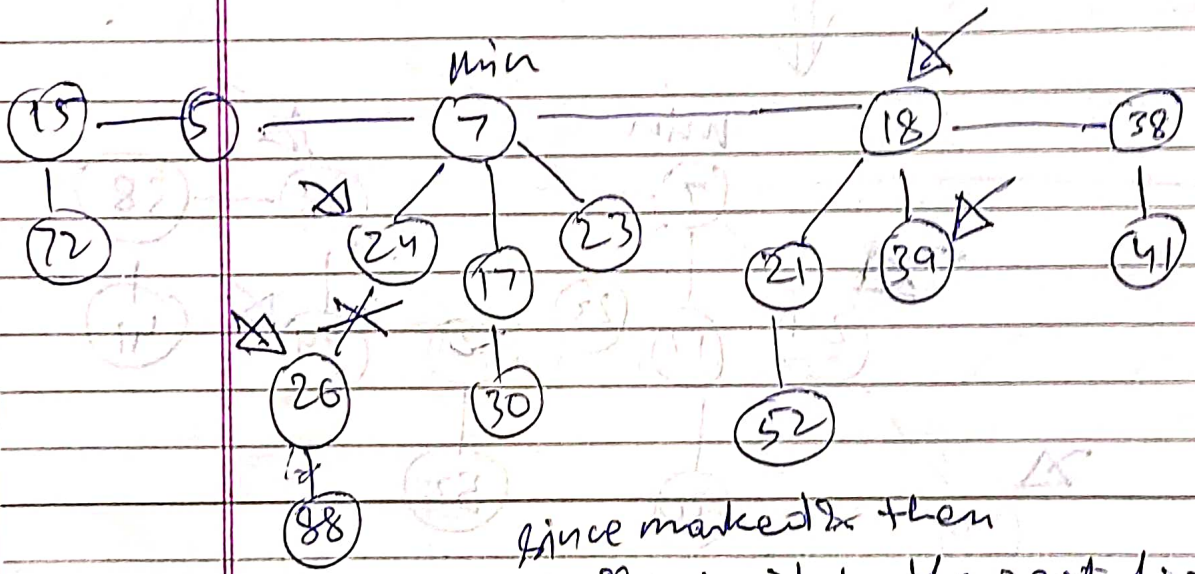
If you want to change the same node to 15.



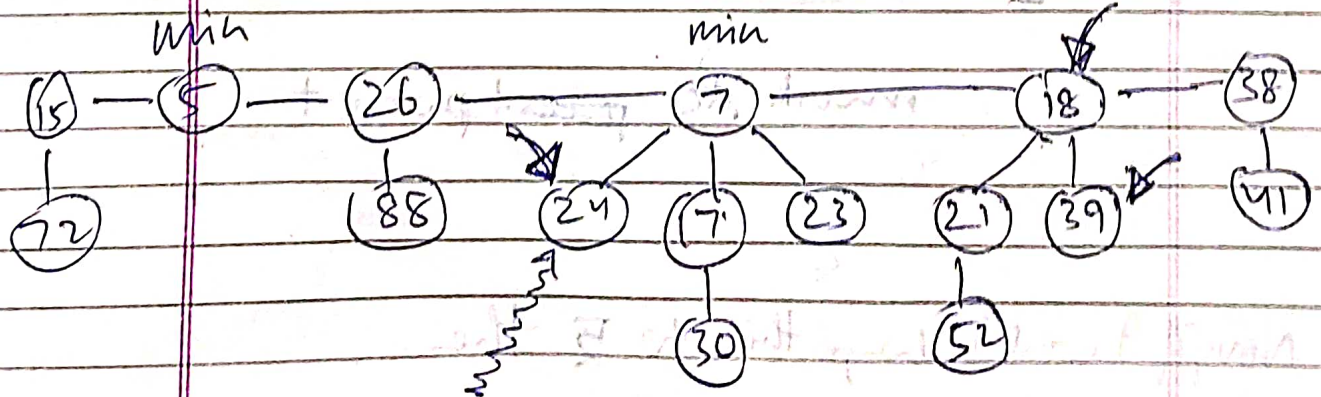
Now, if I want to change this to 5 then.



cut it & join to root list

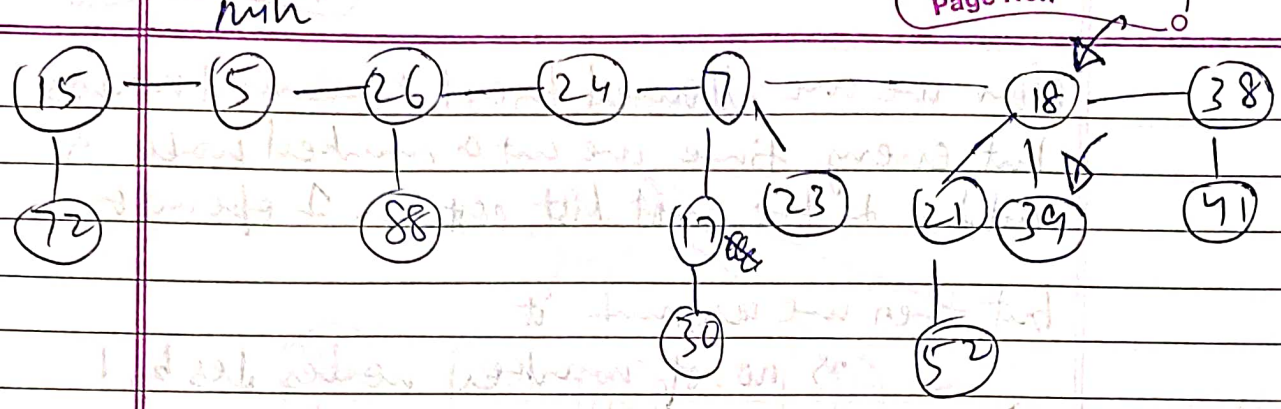


since marked ✗ then
cut its parent & join it to the root list
& unmark it



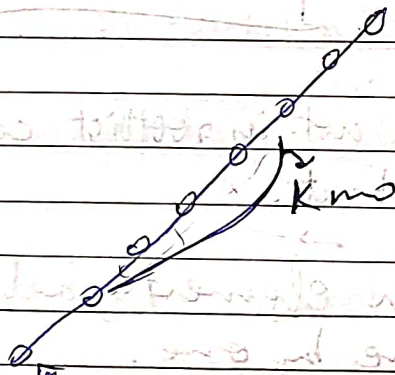
do the same with this since marked

Date: / /
Page No.:



when you reach the root or unmarked parent then you stop.

whether marked or unmarked (doesn't matter)



suppose we have to change its value to less than the value of its parent

the Amortized cost of (cutting & adding to the root list) of the required node

$= 1 + (1) = 2$

↓ Potential diff

$1 + ($ $1 + (6 + 4(5) - 5 = 24))$

$1 + (1 + 2)$

4

Date: / /
Page No.:

Now we are allowed to cut k marked nodes but every time we cut a marked node & add it to the root list cost us 1 operation

but then we unmark it

⇒ no. of marked nodes ↓ by 1

∴ Amortized cost ~~of~~ ~~car~~ for each of k node

is $1 + [1 + 2(-1)]$

$= 0$

$\Rightarrow \sum \bar{c}_i \leq 1 \quad \Rightarrow$ **order is $O(1)$**

an node which is not in rootlist can have atmost one childcut.

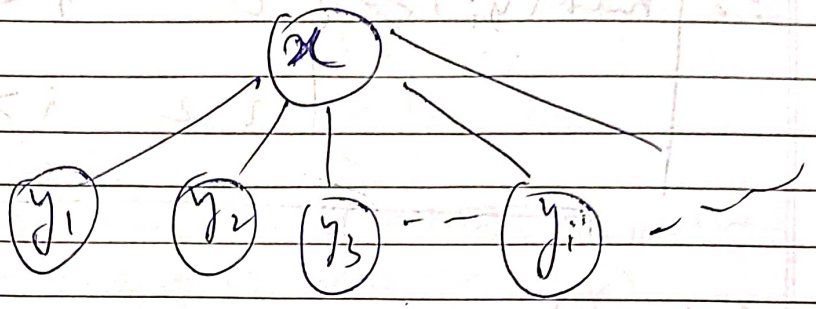
Build Heap with n elements, adding each of them one by one.

$$n \times O(1) = O(n)$$

you can't do increase key.
For that delete the node & add new one which will take $O(\log n)$.

Date: / /
Page No.:

→ Lets look at a node



Suppose this is the order in which x got its children i.e $y_1, y_2, y_3, \ldots, y_i, \ldots, y_n$

when y_i was going to become its child it must have to merge

$\Rightarrow$ Rank(x) = Rank(y_i) = i−1

It might be possible that y_i loses a child later & becomes a mark node

Now suppose x loses y_2, y_4, y_5

then index of y_i reduces but it still has the same rank

$\Rightarrow$ rank of a child of x with ith index will have rank $\geq$ i−1

It might happen that y_i loses one child & becomes marked

then $\boxed{\text{rank}(y_i) \geq i-2}$

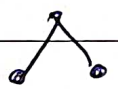
Date: / /
Page No.:

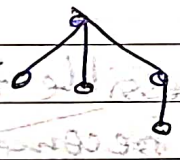
$$\therefore \text{Rank}(y_i) \geq \begin{cases} 0 & i < 2 \\ i-2 & i \geq 2 \end{cases}$$

Let S_k be the smallest possible tree of rank k satisfying the above property.

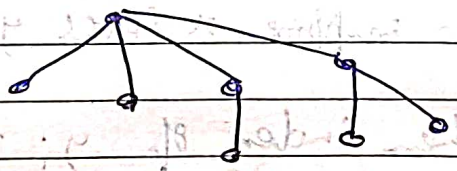
then $S_0 \rightarrow$ •

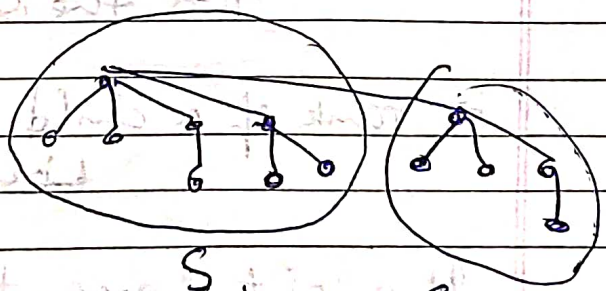
$S_1 \rightarrow$

$S_2 \rightarrow$ 

$S_3 \rightarrow$ 

$S_4 \rightarrow$

$S_4 \rightarrow$ 

$S_5 \rightarrow$ 

$S_4 \quad S_3$

$$\Rightarrow \boxed{S_k = S_{k-1} + S_{k-2}}$$

Date: / /
Page No.:

Fibonacci seq $F(0) = 0$

$F(1) = 1$

$F(2) = 1$

$F(3) = 2$

Similarly

Let the min no. of nodes required to for a heap of rank $r = S(r)$

then $S(r) = S(r-1) + S(r-2)$

$\Rightarrow n \geq S(r) = S(r-1) + S(r-2)$

$> S(r-2) + S(r-2) = 2S(r-2)$

$> 2 \times \{2S(r-4)\} = 2^2 S(r-4)$

$> 2^k S(r-2k) > 2^{r/2}$

$r \geq 2k$

$\Rightarrow k \leq \frac{r}{2}$

$\Rightarrow n > 2^{r/2} \Rightarrow \boxed{r < 2 \log n}$ ★

Date: / /
Page No.:

We are given n trees

o o —— —— n

By using a sequence of n

Fibonacci operations { 1) Add
2) delete
3) decrease key

we can make a heap which has height of order O(n).

Find this sequence

—x—

(*) histogram_problem_bruteforce (A, n)

```
{
    for (i → 0 to n-1)
    { min = A[i]
      for (j = i+1 to n)
      {
        if (A[j] < min)
            min = A[j];
        if (min*(j-1) > max)
            max = (j-1)* min
      }
    }
    print (max);
}
```

$O(n^2)$

Date: / /
Page No.:

```
index_of_a_no_smaller_than_me_right (A, n, B)
{   top = -1;
    for (i → 0 to n)  → (n-1 → -1) ★
    {
      if (top < 0)
          S[++top] = i;
      else if (A[S[top]] < A[i])
          S[++top] = i;
      else
      {
          while(top >= 0 && A[S[top]] > A[i])
              B[S[top--]] = i
          S[++top] = i
      }
    }
    while(top >= 0)
        B[S[top--]] = n;
}                          ★
                    → (-1)
```

Similarly for index_of_a_no_smaller_than_me_left just make 2 changes (★)

```
histogram_problem (A, n, L, R)
{
    for (i → 0 to n)
    { if (max < (R[i] - L[i] - 2) × A[i])   O(n)
          max = (R[i] - L[i] - 2) × A[i];
      return max
    }
}
```

Date: / /
Page No.:

```
int main()
{   input is array A
    index_of_a_no_smaller_than_me_right(A,n,R)
    index_of_a_no_smaller_than_me_left(A,n,L)
    print(histogram_problem(A,n,L,R));
}
```

	Stable	Unstable
Bubble	✓	
Selection		✓
Insertion	✓	
Merge	✓	
Quick		✓
Count	✓	
Radix	✓	
Bucket	✓	
Heap		✓

```
Binary_search(arr, left, right, a)
{
    m = (left + right)/2
    if (left > right)
        return 0;
    else if (arr[m] == a)
        return 1;
    else if (arr[m] < a)
    { left = m+1; return binary_search(arr, left, right, a); }
    else if (arr[m] > a)
    { right = m-1; return binary_search(arr, left, right, a); }
}
```

Date: / /
Page No.:

// binary-search(arr, arr_size, a)
{
~~int left, right~~ left = 0; right = arr_size - 1;

while(left ≤ right)
{
m = (left + right) / 2
if (arr[m] == a)
return 1
else if (arr[m] < a)
left = m+1;
else if (arr[m] > a)
right = m - 1;
}
return 0;
}

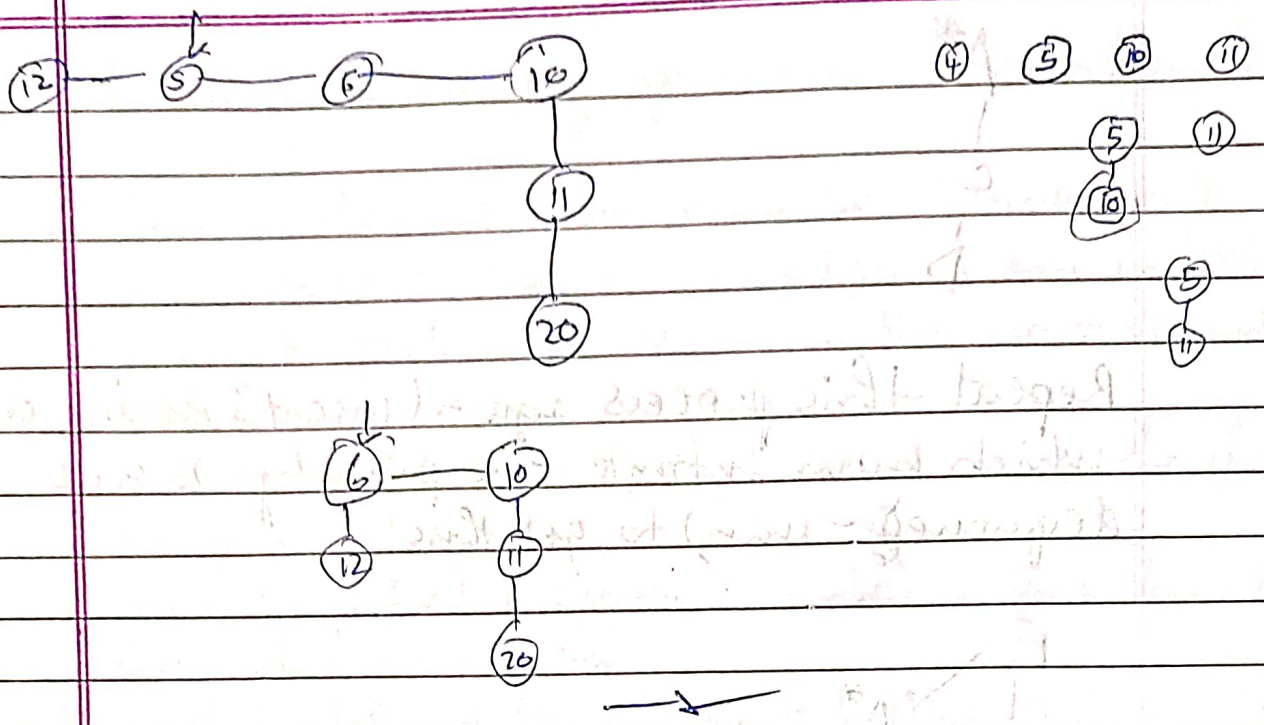
→ radix_sort(A, n)
{ count is a temporary array of size 10 with each digit initialized to zero
for(i → 1 to n)
{ if (A[i] > max) max = A[i] }

while((max/10) > 0 || (max%10 > 0))
{ for(i = 0 → n)
count[(A[i]/x)%10] += 1;
for (i = 1 to 10)
count[i] += count[i-1]
for(i = n-1; i >= 0; i--)
count[(A[i]/x)%10] -= 1
B[count[(A[i]/x)%10]] = A[i];

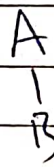
for(i = 0 to n)
A[i] = B[i]; outside for loop but inside while loop → x *= 10; max = max/10 }

Date: / /
Page No.:

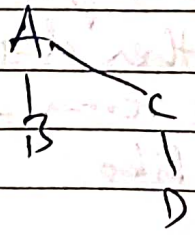
```
void count_sort(A, n, B)
{
  for(i=1 to n)
  { if(A[i] < min)
        min = A[i];
    if(A[i] > max)
        max = A[i];
  }
  ind_det_for_a_no = abs(0 - min);
  count array -> [0 0 0 0 ...]
                  ↓
                 size (max - min + 1)

  if(min <= 0)
  { for(i=0 to n)
      count[A[i] + ind_det_for_a_no] += 1 }
  else
  { for(i=0 to n)
      count[A[i] - ind_det_for_a_no] += 1 }

  for(i=1 to max - min + 1)
      count[i] += count[i-1]

  if(min <= 0)
  { for(i=n+1 to -1)
    { count[A[i] + ind_det_for_a_no] -= 1
      B[count[ _ _ _ _ _ ]] = A[i]
    }
  }
  else
  { for(i=n+1 to -1)
    { count[A - ind_det _ ] -= 1
      B[count _ _ _ _ ] = A[i]
  }}
```



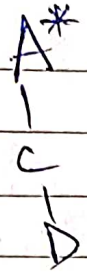
Getting to height one is easy: add in three nodes, then do a dequeue-min. This removes one node and combines the other 2 nodes, which have height 0, into this structure of height 1:



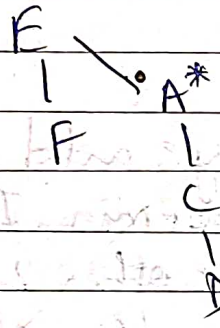
If you repeat this process again & ensure one of the new nodes has the lowest priority, you get 2 of these trees, which are then merged together like this:



Now, do a delete operation on B. This leaves A with order 1 & a mark.



Repeat this process again (insert 3 nodes, all of which have infinite -ve priority, & call dequeue(s) - min) to get this:



Delete F to get

If you repeatedly execute this process of adding 3 nodes, deleting one, then deleting the singleton child of the single remaining tree of height n-1 for any n you'd like.

Date: / /
Page No.:

Finding tree using pre-order & inorder → {O(n²)}

1.) Pick an element from preorder. Increment a preorde Inden. variable (pre inden in below code) to pick next elemet in the next recursive call.
2.) Create a new tree node tNODE with the data as picked element.
3.) Find the picked element's inden in Inorder. Let the inden be inInden.
4.) Call buildTree for elements before inInden & make the built tree as left subtree of tNode
5.) Call buildTree for element after inInden & make the built tree as right subtree of tNode
6.) return tNode.

code→

```
struct node {
    char data;
    struct node * left;
    struct node* right;
}

search(arr, strt, end, value);
struct node* newNode(char data);
```

Date: / /
Page No.:

```
struct node* buildTree(in[], pre[], inStrt, inEnd)
{
    static int preIndex = 0;
    if (inStrt > inEnd)
        return NULL;

    struct node *tNode = newNode(pre[preIndex++]);

    if (inStrt == inEnd)
        return tNode;

    int inIndex = search(in, inStrt, inEnd, tNode->data);

    tNode->left = buildTree(in, pre, inStrt, inIndex-1);
    tNode->right = buildTree(in, pre, inIndex+1,
                             inEnd);

    return tNode;
}

int search(char arr[], int strt, int end, char value)
{
    int i
                    end+1
    for (i=strt; i<=end; i++)
        if(arr[i] == value)
            return i;
}

struct node *newNode(char data)
{ struct node *node = (struct node*) malloc
                      (sizeof(struct node));
  node->data = data; node->left = NULL;
  node->right = NULL; return(node); }
```

Date: / /
Page No.:

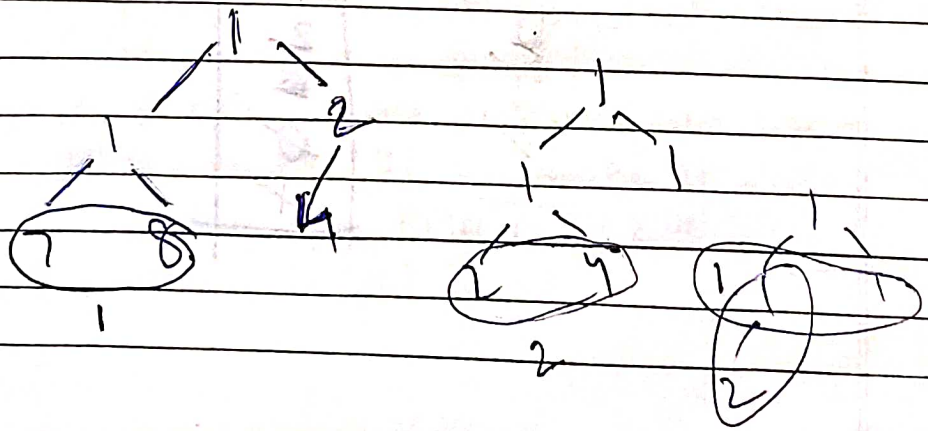
```
printInorder(struct node * node)
{
    if(node == NULL)
        return;
    printInorder(node->left)
    printf("%c", node->data);
    printInorder(node->right);
}
```

2.34 8 4.23 20 18 16 32 26
5.68 12.1 15 23 19 18.5 35 33

4.23 18.5
16 18 19
2.34 5.68 8 12.1 15 20 23 26 32 33 35

2.34 4.23 5.68 8 12.1 15 16 18 18.5 19 20
23 26 32 33 35

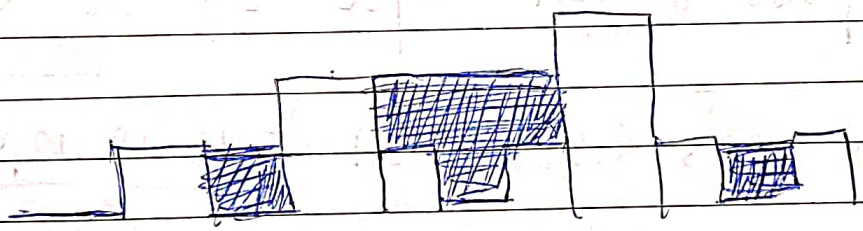
2 4 6 8 12 20
13 16 19 22 24
7 15 23 35 42
5 9 21 28 32

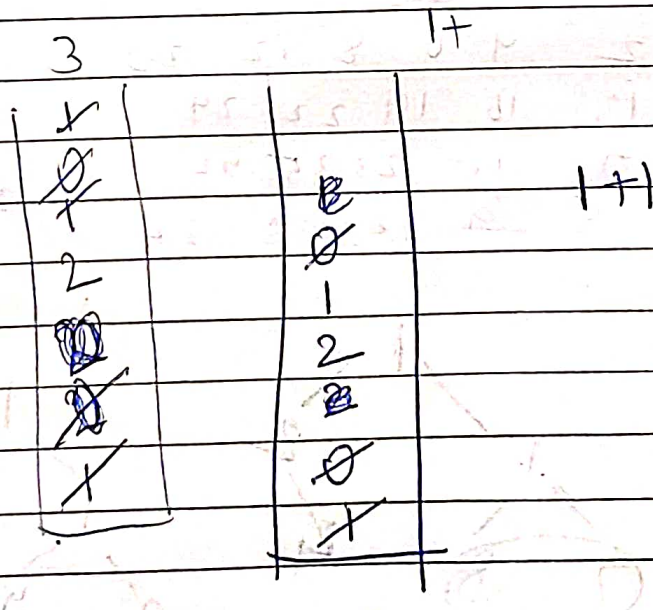


Date: / /
Page No.:

Let $a_1, a_2 \ldots a_n$ be the stock prices of a company in the last n days. Design a linear time algo to compute the span of the stock price for all n days.

→
```
calculateSpan(int price[], int n, int S[])
{
    // Span value of first day is always 1
    S[0] = 1

    // Calc. Span value of remaining days by linear
    checking
    // previous days
```





1.) struct node {
int data; → can be of any type of data structure
struct node *next;
}

2.) bool search (struct node *head, int x) { → Searching a node with data x in it.
while (head != NULL) {
if (head → data == x) return true;
head = head → next;
}
return false; }

main {
search (head, 1); → will just search for 1 in LL & return True or False. It will not change 'head' outside its scope.
}

i.e. we are not modifying the linked list then "*head" → simple star

adding the node at the beginning of LL

3.) addatbeg (struct node **head, int key) {
struct node *temp;
temp = malloc (sizeof (struct node));
temp → data = key;
temp → next = *head
*head = temp;
}

when calling → addatbeg (&head ..)

b/c if head = NULL (i.e. empty LL) then address of head has to change

addatend (struct node **head) { if (head != NULL) while (head != NULL) { ---- } }

deleteatbeg (struct node **head) {

```
struct node *temp;
if (*head != NULL) {
    temp = *head;
    *head = temp->next;
    free(temp); }
}
```

String matching

text

T → n (length of Text)

0 1 0 1 1 1 1 0 0 1 1 1 1

P → m

1001 ↓ Pattern → Searching for this in this

```
for i → 0 to n-m
    for j → 0 to m
        if (T[i+j] != P[j]) break;
    if (j == m) Print (there is a match at i)
```

$A = [a_n, a_{n-1}, a_{n-2} \cdots]$

$$P(X) \sum_{i=0}^{n} a_i x^i = a_0 + a_1 x + a_2 x^2 + \cdots + a_n x^n$$

Q.) Find $P(x_0)$

No need of temp variable (t)

```
S = a_n
for j = n-1 to 0
    S = S * x_0 + A[i]
```

Soln

```
t = x_0   S = a_0
for i → 1 to n+1
    S = S + n[i] * t
    t = t * x_0
```

11011

Find the value of 101011

0 1 2 3 4 5

$2^5 2^4 2^3 2^2 2^1 2^0$

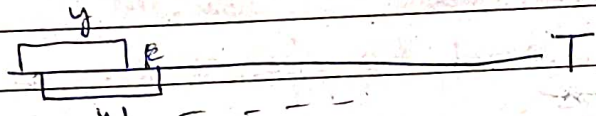
N = P[0]

for j → 1 to m

N = N × 2 + P[j]

j < m

j is strictly less than m



Ans

101011011011

101010

N

P is a prime no.

N = P[0] ; y = T[0] x = 2

for j → 1 to m → m is excluded i.e. j < m

j strictly less than m

N = (N × 2 + P[j]) % P

y = (y × 2 + T[j]) % P

x = x × 2

if (N == y) print ("Matched")

for (i → 1 to n-m)

if (N == y) print (there is a match at i)

$y = (y \times 2 + T[i+m-1] - 2^m \times T[i-1]) \% P$

x

understand for

Algo is correct but there is a limit to the largeness of x so for this,

if y < 0

y += P;

if (N == y) continue

else {

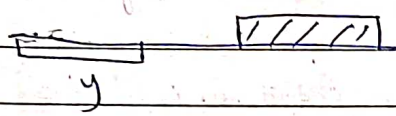
for j → 0 to m

if T[i+j] != P[j] break

if j == m → print match at i

DSA

(#) Robin-Karp Algorithm



$n = 10^5$

$m = 10^4$

P matches T at i

$P[j] = T[i+j]$

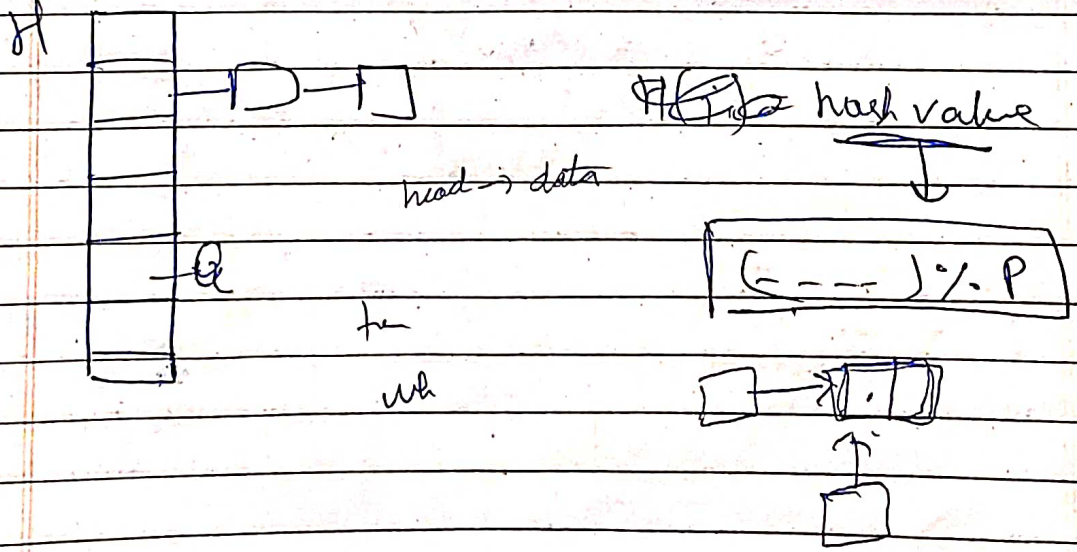
$j = 0, 1, \ldots, m-1$

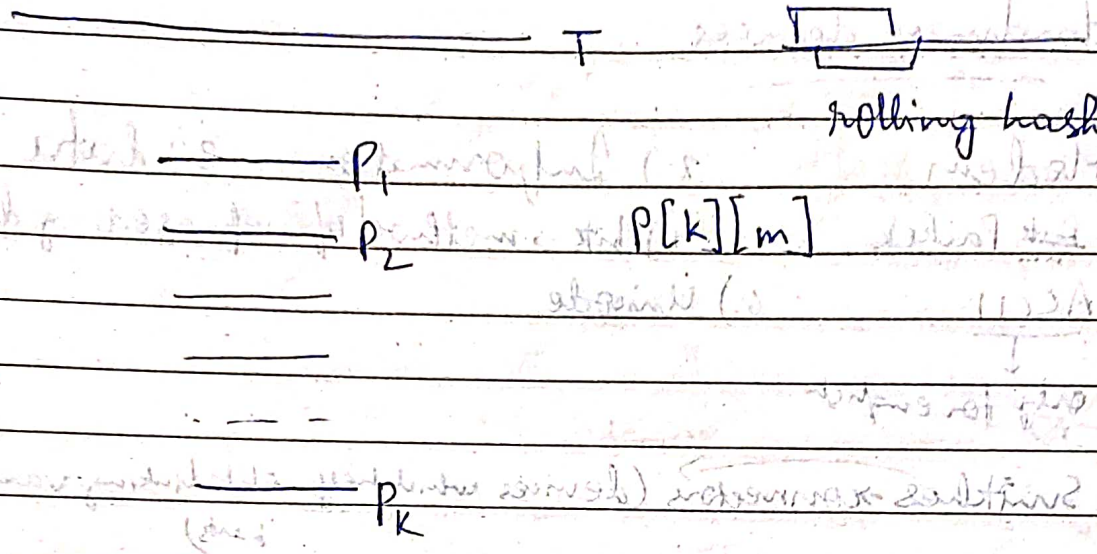
$h : D \rightarrow \{0, \ldots, m-1\}$

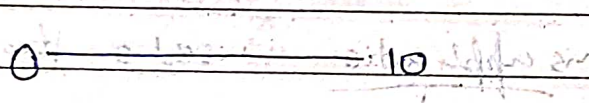
$h(x) = x \bmod P$

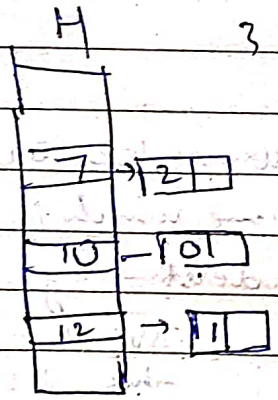
$N = \sum_{i=0}^{m-1} 2^i P[m-1-i]$

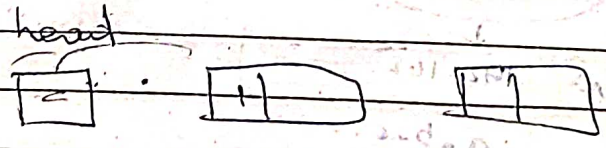
Hash table (array of linked list)




T
rolling hash
P1
P2
Pk
P[k][m]


0
10
1
12
2
3
15


H
7
12
10
101
12
11


head
2
1

DSA

Evaluation

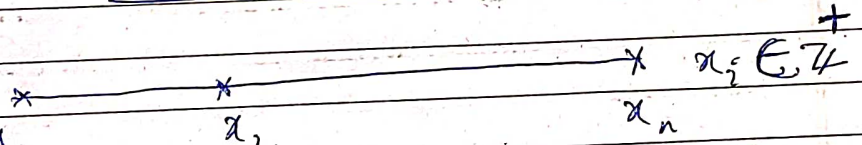
1.) Midterm - 20%
2.) End Term - 20%
3.) 2 Quizzes 30%
4.) 2 Tests 30%

All the exams will be open book/notes.

→ one just before mid-term & one just after mid-term

→ 1st week of Feb & 1st week of April.

Turn Pike reconstruction problem




```
k=0
for i → 1 to n+1
    j → i+1 to n+1
        D[k++] = xj - xi
```

$x_2 - x_1$
$x_3 - x_1$
$x_n - x_1$

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

m has to be equal to $\frac{n(n-1)}{2}$
otherwise no soln.

If it has one soln
then " " infinite soln
b/c we can add const. c
to each term $x_1+c, x_2+c, \dots$

If given m we have to find n

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

$\Rightarrow n^2 - n - 2m = 0$

↓

should have a true integer soln!

$D = \{1, 2, 2, 2, 3, 3, 3, 4, 5, 5, 15, 6, 7, 8, 10\}$

$m = 15$ $n = 6$ If we set $x_1 = 0$, x_2, x_3, x_5, $x_6 = 10$

then the largest x_i should be equal to largest value of D

Then after this we choose (from right to left) priority

$x_5 = 8$

$8 - 0 = 8$ & $10 - 8 = 2$ which is there in the set