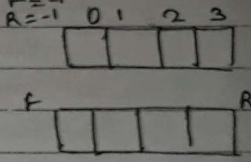
S No.	Date	Title	Pg. No.	Remarks

Unit - 1

(ii) Queue →

- Algo INQUEUE (Rear, Front, Item, Q)

1. If (REAR = MAX - 1)
   then Queue overflow & exit
   else if (Front = -1)
   then Front = 0
   Rear = 0
   Q[Rear] = Item & exit
   else Rear = Rear + 1
   Q[Rear] = Item & exit.



- Algo - DEQUEUE (Front, Item Q)
  If (Front = -1)
  then write Queue is empty & exit
  else Item = Q[Front]
  Delet Item
  Front = Front + 1
  Exit.

- Algo POP (Top, Item S)

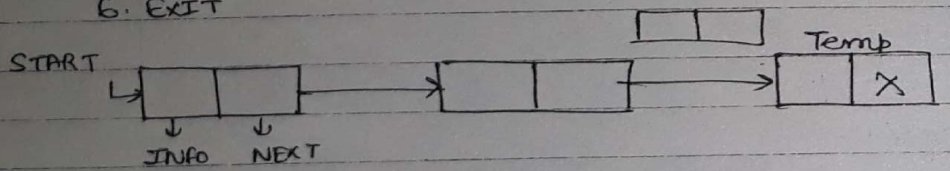
1. If (Top < 0)
   then write STACK is empty & exit
   else Item = S[Top]
   Top = Top - 1
   Delet Item
2. Exit

Scanned by CamScanner

(iii) Linked list

- Algo Ins-Beg (Item, Temp)

1. Read the item do be inserted
2. Temp = (struct NODE *) malloc (size of (NODE*))
3. Temp → INFO := Item
4. Temp → Next := START
5. START := TEMP
6. EXIT



- Algo Ins-last (Item, Temp, PTR)

1. Read the item do be inserted
2. TEMP = (struct NODE *) malloc (size of (NODE*))
3. Temp → INFO = Item
4. PTR = START
5. Repeat while PTR → NEXT != NULL
   PTR = PTR → NEXT
6. PTR → NEXT = TEMP
7. TEMP → NEXT = NULL
8. Exit

Temporary node variable (Temp variable)

- Algo Ins-middle (Item, Temp, PTR, PTR1, count, loc) → location

1. Read the item do be inserted
2. Temp = (struct NODE *) malloc (size of (NODE*))
3. Temp → INFO = Item
4. Count := 1 ; PTR = START ; PTR1 := START
5. Repeat while count < loc
   PTR1 := PTR
   PTR := PTR → NEXT
   count ++
6. PTR1 → NEXT := Temp
7. TEMP → NEXT := PTR.
8. Exit

- Algo DEL-START (start Temp, Item)

1. If (start = NULL)
   write Empty list & Exit
2. If (START → INFO = ITEM) & (START → NEXT = NULL)
   TEMP := START → INFO
   Delet TEMP
   START := NULL
   Exit
3. else TEMP := START → INFO &
   Delete Temp → INFO free (TEMP)
   START := START → NEXT
   Exit

- Algo Del-last (Ptr, Ptr1, Item)

1. PTR = START
2. Repeat while PTR → NEXT != NULL
   PTR1 := PTR
   PTR = PTR → NEXT

3. PTR1→NEXT = NULL
4. TEMP = PTR → INFO
5. DELET TEMP
6. Free PTR

• Algo DEL – Middle (Item, PTR, PTR1, Temp, count, LOC)
1. PTR = START ; count = 1
2. Repeat while count < LOC
   PTR1 := PTR
   PTR = PTR→NEXT
   count ++
3. Delet PTR→INFO
4. PTR1 → NEXT := PTR→NEXT
5. Free PTR
6. Exit.

Discription
Algo
complexity
Implementation

• Algo Search (Ptr, Item)
1. PTR = START
2. Repeat while PTR→NEXT != NULL
   If (ITEM = PTR→INFO)
   write Item is present in linked list
3. If (PTR = NULL)
   write Item is no present in linked list
4. Exit.

• Algo Traversal (Ptr)
1. PTR = start
2. Repeat while PTR→NEXT != NULL
3. write PTR→INFO
4. Exit

Imp

Analyzing Algorithm →
The performance analysis of various algo is based upon two parameters
1. Space Complexity →
The space complexity of an algorithm is the amount of memory it needs to run to till its completion.
It is denoted by S & it is based upon Sc + Sv
S = Sc + Sv
where Sc = Amount of space that is taken by the const. this parameter is fixed & is independent of the inputs & o/p.
It include space of simple variables, constant & fixed sized variables. (At compile time)
where Sv = It is the variable part that consist of the space leaded by the reference variable, component variables, whose size is dependent upon particular problem instance beinged solved.
S = Sc + Sv
= C + Sv
∴ S = Sv
Sc = 0 in the case of value is assign i.e it is const.
If we have already fixed the variable with given values in initial state then the space complexity in this case will be zero so S will be equal 0
like int a = 2, b = 4
c = a+b ∴ In this case C = 6 (Always)

Scanned by CamScanner

• Algo Calculate (a,b,c)
1. int a:=4, b:=6
2. {
3. write c:= a+b-(a/b)*b
4. }
∴ S=0

• Algo sum(a,n)
1. {
2. sum=0
3. for(i=1 to n) step1 do
4. sum = sum + a[i]
5. return sum
6. }

Total variable = 3 (Sv) i, sum, n
Amount of space = n (Sc)
∴ S = n+3

2. ★ Time Complexity →
• The time complexity of an algo is the amount of time needed by it to run to its completion.
• It is sum of compile time & runtime however compile time does not depend on the instance characteristic hence we are concerned only with runtime of the program.
T = Tc + TR

(i) Count Method →
In this method the Time comple of an algorithm is computed using following →

(i) Take a global variable count and initialize it zero.
(ii) Now you each statement that will consumes a compiler time increement the value of count by 1
(iii) The count will not be increemented for the statement that are comments, punctuations symbol etc.

• Algo sum(a,n)
1. {
2. S=00 1
count = count+1;
3. for(i:=1 to n) Step1 do n
{
S = S+a[i] n
count = count+1;
} count = count+1; // To check the loop exit condition
4. return S; 1
count = count+1
5. }

$T(n) = 2n+3$

Initially S=0 for i=4 return
1+6+1+1 = 9
all i=3

i=1 S = 0+1 = 1
i=2 S = 1+2 = 3
i=3 S = 3+3 = 6
i=4

for n = 3 2*3+3 = 9

```
• Algo Add (a,b,c,m,n)
1  for i=1 to m do                    m
2  {
3  for j=1 to n do                    m×n
4  {
5  c[i,j] = a[i,j] + b[i,j]           mn
6  }                                  m
7  }
```

$T_{Add} = 2mn + 2m + 1$

```
• Algo RSum (a,n)                     n≤0    n>0
1    if (n≤0) then                    1      1
2    {
3    return 0;                        1      0
4    }
5    else
6    {
7    return RSum (a, n-1) + a[n];     1      1+tRSum(n-1)
     }
```

where $T_{RSum}(n-1)$ = Total time taken by recursion Sum procedure to compute the sum of first n natural no

$$\therefore T_{RSum} = \begin{cases} 2 & \text{if } n \le 0 \\ 2 + t_{RSum}(n-1) & \text{if } n > 0 \end{cases}$$

(ii) Step-Table method →

This is the second method to determine the step count method in which we compute total no of steps contributed by each statement i.e S/e (Space / execution)

S/e of a statement is the amount by which the count changes as a result of execution of that statement. it value will be either zero or one, the value will be zero when the S/S is not taking the compilers time and the value will be one when sum of the compilers time will be needed to execute that S/S.

The next parameter is frequency which denotes the no of time that particular statement will be executed.

	Statement	S/e	Frequency	S/e * Frequency = Total Steps
1	Algo Sum (a,n)	0	-	-
2	{	0	-	-
3	Sum=0;	1	1	1
4	for (i=1 to n) step 1 do	1	n+1	n+1
5	{Sum = Sum + a[i];	1	n	n
6	}	0	-	0
7	return Sum;	1	1	1
8	}	0	-	-

$\therefore$ Time Complexity = 2n+3

Scanned by CamScanner

Statement	s/e	frequency	Total steps
1. Algo matrix Add (a,b,c,m,n)	0	-	-
2. {	0	-	-
3. for(i=1 to m)do	1	m+1	m+1
4. {	0		
5. for(j=1 to n)do	1	mn+m	mn+m
6 {	0	-	-
7. c[i,j]=a[i,j]+b[i,j]	1	mn	mn
8. }}}	0	-	-
			2mn+2m+1

Statement	s/e	frequency $n \le 0$	frequency $n > 0$	Total steps $n \le 0$	Total steps $n > 0$
1. Algo RSum(a,n)	0	-	-	-	-
2. {	0	-	-	-	-
3 If ($n \le 0$) then	1	1	1	1	1
4. return 0;	1	1	-	1	-
5. else	0	-	-	-	-
6. return RSum(a,n-1)+a[n]	1	-	1	-	1
7 }	0	-	-	-	-
				2	2+trsum(n-1)

$$trsum(n) = \begin{cases} 2 & \text{if } n \le 0 \\ 2 + trsum(n-1) & \text{if } n > 0 \end{cases}$$

★ Asymptotic Notation

- Different running time can be obtain from a single algo depending upon the various characteristic
- The algo efficiency / performance can be described by the order of the growth of the running of an algo.
- Example → The time complexity for algo A is $c_1n^3 + c_2n^2$ & for B it is c_3n^2

A: $c_1n^3 + c_2n^2$

B: c_3n^2

$c_1 = 2$, $c_2 = 3$, $c_3 = 100$

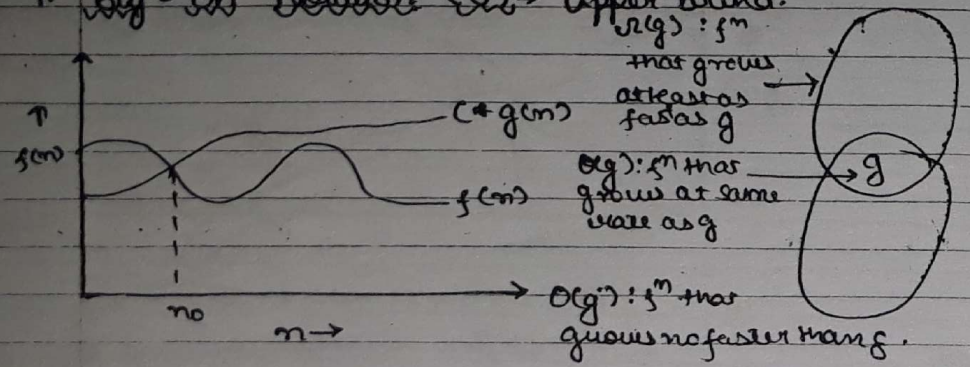
Normally it is consider as algo A will take more time than algo B bcz no of terms involved in A is more than B, however if we specify the value of constraints as given, then there would be a considerable change then algo B will take much more time than algo. This diff of growth of f^n can be denoted by asymptotic notation which are used to find out asymptotic running time of an algo.

They are of 5 types →

i. Big - oh notation (O)
ii. Big - omega notation (Ω)
iii. Big - Theta notation (Θ)
iv. Small - oh notation (o)
v. Small - omega notation (ω)

Scanned by CamScanner

1. Big - oh Notation (O) → Upper bound.



Consider g to be a f^n of non-negative integers that is mapped onto the $f^n f(n)$ with the condition $f(n) = O(g(n))$ such that there exist +ve const 'c' & 'no' which satisfy the condition $f(n) \leq c * g(n)$ for $n \geq n_0$

$f(n)$ = Running time of the algorithm.
$g(n)$ = Highest degree in and
c = value of Highest degree coefficient of 'n+1'

Eg → (i) $f(n) = 2n + 3$

$2n + 3 \leq 3n$

$n=1$	$5 \not\leq 3$
$n=2$	$7 \not\leq 6$
$n=3$	$9 \leq 9$

$f(n) = c * g(n)$ for $n \geq n_0$.

$f(n) = O(g(n))$ for $n \geq n_0$

$f(n) = O(n)$ for $n \geq 3$ $c = 3$.

(ii) $f(n) = 7n + 5$

$7n + 5 \leq 8n$

$n=1$	$12 \not\leq 8$
$n=2$	$19 \not\leq 16$
$n=3$	$26 \not\leq 24$
$n=4$	$33 \leq 32$
$n=5$	$40 \leq 40$.

$f(n) = O(n)$ for $n \geq 5$; $c = 8$

(iii) $f(n) = 27n^2 + 16n + 25$.

$27n^2 + 16n + 25 \leq 28n^2$

$n=16 = 27*16 + 16 \cdot 16 + 25 \not\leq 28 (16)^2$

$n=18 = 27 \cdot 18 + 18 \cdot 18 + 25 \leq 28 (18)^2$

$\therefore f(n) = O(n^2)$ for $n \geq 18$; $c = 28$.

(iv) $5 * 2^n + 3n + 5$

$5 * 2^n + 3n + 5 \leq 6 * 2^n$

$f(n) = O(2^n)$ for $n \geq$ $c = 6$

(v) $f(n) = 2^n + 6n^2 + 3n$

		2^n	n^2
n^2	$n=1$	2	1
	$n=2$	4	4
	$n=3$	8	9
	$n=4$	16	16
2^n	$n=5$	32	25
	⋮		

Case I If $n \geq 4$ & $n=1$ $2^{n_0} + n^2 + 3n \leq 2 * 2^n$

$f(n) = O(2^n)$ for $n \geq$, $c = 2$

Case II If $n > 4$ $2^n + n^2 + 3n \leq 2n^2$

$f(n) = O(n^2)$ for $n \geq$; $c = 2$

$$f(n) = \begin{cases} O(2^n) & \text{if } n > 4,\ 2n = 1 \\ O(n^2) & \text{otherwise} \end{cases}$$

(vi) $f(n) = 600 * n^0$

$600 * n^0 \leq 601 * n^0$

$f(n) = O(n^0)$

$\boxed{f(n) = O(1)}$

2. Big-Omega Notation (Ω) → lower bound → Best case

Consider g is a function of non-negative integer that is map onto $f(n)$ such that $f(n) = \Omega(g(n))$ for $n \geq n_0$ such that there exist +ve const c & n_0 for which

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

for $n \geq n_0$

c = (i) If the f^n contain +ve sign then the value is heighest degree coefficient of n

(ii) If the f^n contain -ve sign then the value is heighest degree coefficient of n then -1

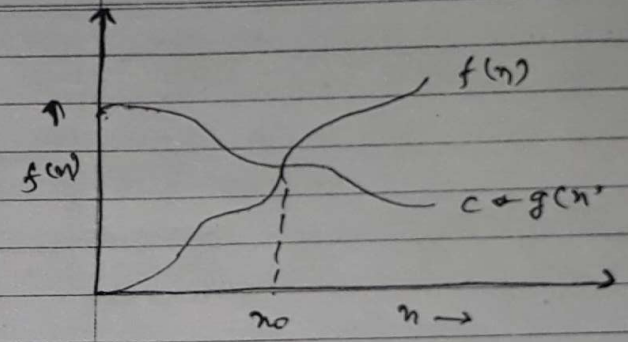
However if the f^n contain both + & - then value of c will be considered by the sign of two heighest degree power of n.

$f(n) = 2n+5 \quad \therefore c = 2$

$f(n) = 3n-5 \quad \therefore c = 3-1 = 2$

$f(n) = 2n^2+3n-5 \quad \therefore c = 2.$

$f(n) = 3n^2-2n-5 \quad \therefore c = 3-2 = 1.$



$f(n) = \Omega(g(n))$ for $n \geq n_0$

(i) $\therefore f(n) = 3n+5$

$3n+5 \geq 3n$

$f(n) = \Omega(n)$ for $n \geq 1 \quad c = 3$

(ii) $f(n) = 2n^2+6n-5$

$2n^2+6n-5 \geq 2n^2$

$f(n) = \Omega(n)$ for $n \geq 1 \quad c = 2$

(iii) $f(n) = 3 * 2^n + 4n^2 + 5n + 3$

	3×2^n	$4n^2$
$n=1$	6	4
$n=2$	12	16
$n=3$	24	36
$n=4$	48	64

Case I $n=1$

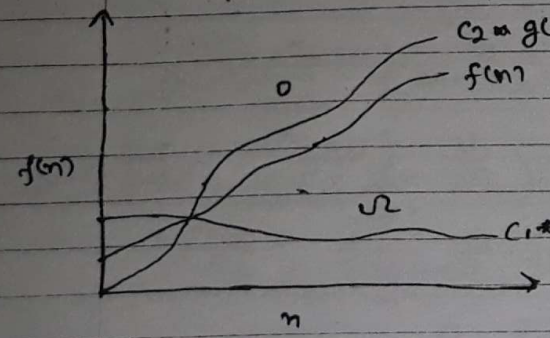
$f(n) = 3 * 2^n + 4n^2 + 5n + 3 \geq 3 * 2^n$

Case II $n \geq 1$

$f(n) = 3 * 2^n + 4n^2 + 5n + 3 \geq 4n^2$

Scanned by CamScanner

3. Big Theta Notation (Θ) → Worst case



The lower & upper bound for the f^n $f(n)$ is provided by big theta notation.
Considering $g(n)$ is the f^n of non-ve integer
i.e map onto $f(n)$ such that $f(n) = \Theta(g(n))$
where $\Theta(g(n)) = O(g(n)) \cap \Omega(g(n))$
The condition for Big Theta can be obtained by
$c_1 * g(n) \leq f(n) \leq c_2 * g(n)$
for $n \geq n_0$
Here $c_1 * g(n)$ calculated by Ω notation
$c_2 * g(n)$ calculated by O notation

(i) $f(n) = 2n + 3$
$2n \leq 2n+3 \leq 3n$
$n=1 \quad 2 \leq 5 \leq 3$ ✗
$n=2 \quad 4 \leq 7 \leq 6$ ✗
$n=3 \quad 6 \leq 9 \leq 9$ ✓
$\therefore f(n) = \Theta(g(n))$
$f(n) = \Theta(n)$ when $n \geq 3$ $c_1=2$ & $c_2=3$

(ii) $f(n) = 3*2^n + 4n^2 + 5n + 3$.

	$3*2^n$	$4n^2$	
$n=1$	6	4	$3*2^n$
$n=2$	12	16	n^2
$n=3$	24	36	n^2
$n=4$	48	64	n^2
$n=5$	96	100	n^2
$n=6$	192	144	2^n
$n=7$	384	196	2^n
$n=8$	768	256	2^n
$n=9$	1536	324	2^n
$n=10$	3072	400	2^n

Case I when $n=1$, $n>5$
$3*2^n \leq 3*2^n + 4n^2 + 5n + 3 \leq 4*2^n$
$f(n) = \Theta(2^n)$ for $n \geq \ldots$
$c_1 = 3$, $c_2 = 4$

Case II when $n = 2, 3, 4, 5$
$4n^2 \leq 3*2^n + 4n^2 + 5n + 3 \leq 5n^2$
$f(n) = \Theta(n^2)$ for $n \geq \ldots$
$c_1 = 4$, $c_2 = 5$

$$f(n) = \begin{cases} \Theta(2^n) & \text{when } n=1, n>5 \\ \Theta(n^2) & \text{otherwise} \end{cases}$$

Scanned by CamScanner

(ii) $f(n) = 4*2^n + 3n$

$4*2^n \leq 4*2^n + 3n \leq 5*2^n$

$f(n) = O(2^n) \quad n \geq 4$

$c_1 = 4 \quad c_2 = 5$

$n=1$	$8 \leq 11 \leq 10$	✗
$n=2$	$16 \leq 22 \leq 20$	✗
$n=3$	$24 \leq 41 \leq 40$	✗
$n=4$	$64 \leq 76 \leq 80$	✓

4. little Oh notation (o) →

This notation represents a loose bounded value of version of big-oh notation.

It bounds from top but does not bound the bottom.

Consider g is the f^n of non-ve integer that is mapped on f(n) such that

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = 0$$

$f(n) = og(n)$

$f(n) = o(n^2)$

(i) $f(n) = 4n^3 + 2n + 1$

$f(n) = O(n^4)$

$$\therefore \frac{f(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{4n^3 + 2n + 1}{n^4}\right)$$

$$= \lim_{n\to\infty} \left(\frac{4}{n} + \frac{2}{n^3} + \frac{1}{n^4}\right) = 0$$

(ii) $f(n) = 4n^3 + 2n \quad 4*2^n + 3n$

$$f(n) = \frac{4*2^n + 3n}{n*2^n}$$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \lim_{n\to\infty} \left(\frac{4*2^n + 3n}{n*2^n}\right)$$

$$= \lim_{n\to\infty} \left(\frac{4*2^n}{2^n * n} + \frac{3n}{n*2^n}\right)$$

$$= \lim_{n\to\infty} \left(\frac{4}{n} + \frac{3}{2^n}\right)$$

$\frac{1}{\infty} = 0$

$= 0$

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

The value you constant is undefinied in the case of little o.

5. Little Omega notation (ω)

This notation provide a loose boundary of the f^n $\Omega(g(n))$ it bounds the values from the bottom but not from the top.

Consider a f^n least g(n) in the set of non-ve integer that is mapped onto the f(n)

$f(n) = \omega(g(n))$

$$\lim_{n\to\infty} \frac{f(n)}{g(n)} = \infty$$

(i) $f(n) = \lim_{n\to\infty} (3n+2)n$

$= \lim_{n\to\infty} (3n^2 + 2n)$

$= \infty$

(ii) $f(n) = 16$

$f(n) = \lim_{n\to\infty} 16n^4$

$= \infty$

(ii) $f(n) = \lim_{n\to\infty} 6n + 3n^5$

$f(n) = \lim_{n\to\infty} \left(\frac{16}{n^4} + 3\right) n^5$ → Θ^n

$= \lim_{n\to\infty} \left(\frac{16}{n^4} + 3\right) n^5$

$= \infty$

2. Change of variable method →

In this method, we can change some of the variables of the given recurrence relation so that it can appear in a form already familiar to us.

(i) $T(n) = 2T(\sqrt{n}) + \log_2 n$ —①

Let any variable $m = \log_2 n$ —②

Taking exponential both sides $2^m = n$ —③

Taking square root on both the sides

$2^{m/2} = \sqrt{n}$ —④

Put 2, 3 & 4 in ①

$T(2^m) = 2T(2^{m/k}) + m$ —⑤

Let $T(2^m) = A(m)$ —⑥

Replace $m \to m/2$

$T(2^{m/2}) = A(m/2)$ —⑦

Put 6 & 7 in 5

$A(m) = 2A(m/2) + m$ —⑧

$a = 2, b = 2, f(m) = m$

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

$\boxed{T(m) = \Theta(m \log m)}$

(ii) $T(n) = T(\sqrt{n}) + 1$

Let $m = \log_2 n$ — ①

Taking exponential both sides,

$2^m = n$ — ②

Sq.

$2^{m/2} = n^{1/2}$ —④

Put 2, 3, 4 in ①

$T(2^m) = T(2^{m/2}) + 1$ —⑤

Scanned by CamScanner

let $T(2^m) = A(m)$ — (6)

Replace $m \to m/2$

$T(2^{m/2}) = A(m/2)$

$T(2^{m/2}) = A(m/2)$ — (7)

Put (6) & (7) in (5)

$A(m) = A(m/2) + 1$

$a=1, b=2 \quad f(m) = 1 = m^0$

$n^{\log_b a} = n^{\log_2 1}$

$n^{\log_b a} = m^0$

$T(m) = \Theta(m^0 \log m)$

$T(m) = \Theta(\log_2 m)$ is $\Theta(\log(\log_2 n))$

3. Iteration Method →

This method is applied to expand the recurrence & express the relation as summation of terms dependent only upon 'n' & the initial condition.

Example → $T(n) = T(n-1) + 1$ — (1)

let $n = n-1$ in (1)

$T(n-1) = T(n-2) + 1$ — (2)

Put (2) in (1)

$T(n) = T(n-2) + 2$ — (3)

let $n = n-2$ in eq (1)

$T(n-2) = T(n-3) + 1$ — (4)

Put (4) in (3)

$T(n) = T(n-3) + 3$ — (5)

Extending the substitution for 'k' iterations

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

As the max value of k can be extended upto $(n-1)$ terms so putting value of $k = n-1$

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

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

Ignoring the time complexity of const.

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

4. Substitution Method →

In this method, we make repeated substitution for each occurrence of 'n' in right hand side until all such occurrence disappear.

$$T_n = \begin{cases} a + (n/b) + f(n) & \text{if } n > 1 \\ 1 & \text{if } n = 1 \end{cases}$$

(i) $T(n) = 2T(n/2) + n/2$ — (1)

let $n = n/2$

$T(n/2) = 2T(n/4) + n/2$ — (2)

Put (2) in (1)

$T(n) = 2[2T(n/4) + n/2] + n/2$

$T(n) = 4T(n/4) + n + n = 4T(n/4) + 2n$ — (3)

Put $n = n/4$ in eq 1

$T(n/4) = 2T(n/8) + n/4$ — (4)

Put (4) in 3

$T(n) = 4[2T(n/8) + n/4] + 2n$

$= 8T(n/8) + n + 2n$

$= 8T(n/8) + 3n$ — (5)

Scanned by CamScanner

DELTA Pg No. Date / /

expanding the substitution upto k iterations

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

Put $n = 2^k$

$k = \log_2 n$

$T(n) = nT\left(\frac{n}{n}\right) + \log_2 n(n)$

$T(n) = nT(1) + n\log_2 n$

$T(n) = n + n\log_2 n$

(ii) $T(n) = \begin{cases} \Theta(n\log_2 n) & \text{if } n \le 10 \\ \Theta(n) & \text{if } n > 10 \end{cases}$

$T(n) = T(n/2) + 1$ — (1)

Put $n = n/2$ in (1)

$T(n/2) = T(n/4) + 1$ — (2)

Put 2 in (1)

$T(n) = T(n/4) + 2$ — (3)

Let $n = n/4$ in (1)

$T(n/4) = T(n/8) + 1$

Put 4 in 3

$T(n) = T(n/8) + 3$

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

Let $n = 2^k$

$k = \log_2 n$

$T(n) = T(n/n) + \log_2 n$

$T(n) = T(1) + \log_2 n$

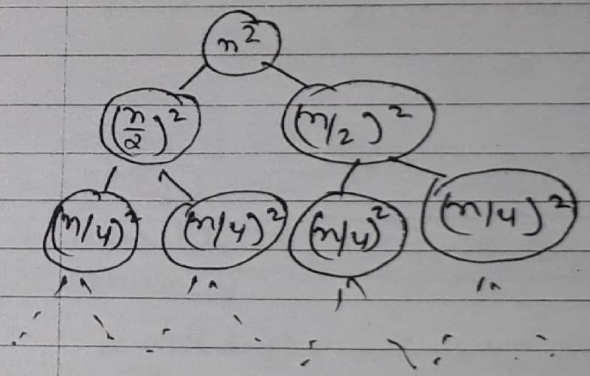
$T(n) \approx \Theta(\log_2 n)$

DELTA Pg No. Date / /

5. Recurrence Trees →

- It is the pictorial representation of Iteration method.
- It is in the form of a tree where each of every level is considered and the term of $f(n)$ is labeled as root node of recurrence Trees.
- It is useful when divide & conquer algorithm

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



level 0 $2^0 * n^2$ $f(n) = n^2$

level 1. $2^1\left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)^2\right)$

level 2. $2^2\left(\left(\frac{n}{4}\right)^2 + \left(\frac{n}{4}\right)^2 + \left(\frac{n}{4}\right)^2 + \left(\frac{n}{4}\right)^2\right)$

$f(n) = n^2 + 2 * \frac{2n^2}{4} + 4 * \frac{4n^2}{16} + \dots$

$= (n^2 + n^2 + n^2 + \dots)$ L time. [L = no of level]

$= Ln^2$

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

(ii) $T(n) = 8T(n/2) + n^2$

level 0 = $8^0 * n^2$

level 1 = $8^1 \left(\left(\frac{n}{2}\right)^2 + \left(\frac{n}{2}\right)^2\right)$

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

$f(n) = n^2 + 8 * \left(\frac{2n^2}{4}\right) + 64\left(\frac{4n^2}{16}\right)$

$= n^2 + 4n^2 + 16n^2 + \ldots$

$= n^2(1 + 4 + 16 \ldots)$

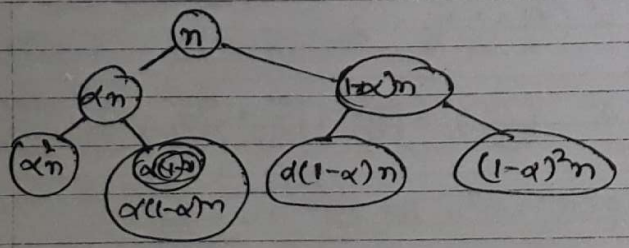
$\frac{a}{1-r}$ $r = \frac{4}{1}$ $\frac{a}{r-1}$ G.P

$= n^2\left(\frac{1}{4-1}\right) = \frac{n^2}{3}$

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

(iii) $T(n) = T(\alpha n) + T(1-\alpha)n + n$

Height of tree is $\log n$.



n

$\alpha n + (1-\alpha)n$

$\alpha(1-\alpha)n + (1-\alpha)^2 n$

$f(n) = (n + \alpha n + \alpha(1-\alpha)n + \alpha(1-\alpha)n + (1-\alpha)^2 n) \log n$

$= (n + \alpha n + 2\alpha(1-\alpha)n + (1-\alpha)^2 n) \log n$

$= n(1 + \alpha + 2\alpha(1-\alpha) + (1-\alpha)^2) \log n$

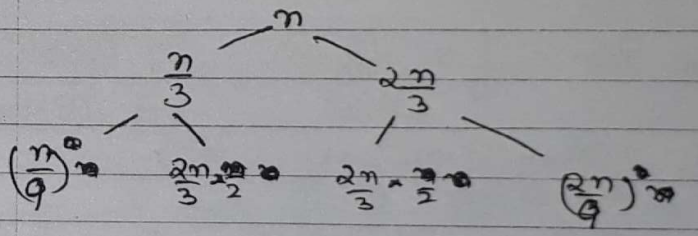
$= n(1) \log n$

$= n \log n$

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

(iv) $T(n) = T(n/3) + T(2n/3) + n$ Height = $\log n$

$f(n) = n$



$n^2 + \frac{n}{3} + \frac{2n}{3} + \frac{n^3}{9} + \frac{2n^3}{6} + \frac{2n^3}{6} + \frac{4n^3}{9}$

$n^2 + \frac{3n}{3} + \frac{5n^3}{9} + \frac{8n^3}{6} + 4n^3 + 4^3$

$\left(n + \frac{n}{3} + \frac{2n}{3} + \frac{n}{9} + \frac{2n}{6} + \frac{2n}{6} + \frac{2n}{9}\right) \log n$

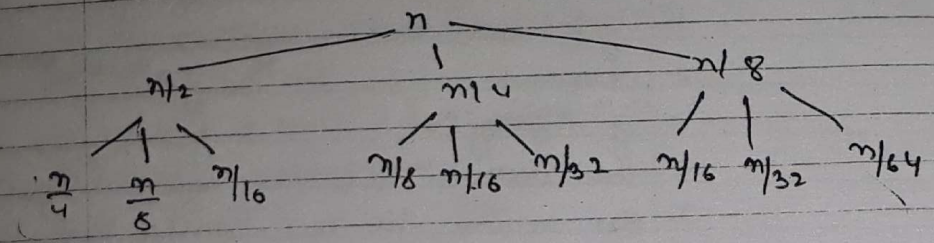
$\left(n + \frac{3n}{3} + \frac{3n}{9} + \frac{4n}{6}\right) \log n.$

$n + n + \frac{n}{3} + \frac{2n}{3}$

$n\left(2 + \frac{1}{3} + \frac{2}{3} + \ldots\right) \log n.$

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

(v) $T(n) = T\left(\frac{n}{2}\right) + T\left(\frac{n}{4}\right) + T\left(\frac{n}{8}\right) + n$ Height of the tree is $\log n$.



$$n + \frac{n}{2} + \frac{n}{4} + \frac{n}{8} + \frac{n}{4} + \frac{n}{8} + \frac{n}{16} + n$$

$$n + \frac{n}{2} + \frac{2n}{4} + \frac{3n}{8} + \frac{3n}{16} + \frac{2n}{32} + \frac{n}{64}$$

$$\left(n + n + \frac{3n}{8} + \frac{3n}{16} + \frac{n}{16} + \frac{n}{64}\right) \log n$$

$$\left(2n + \frac{3n}{8} + \frac{4n}{16} + \frac{n}{64}\right) \log n$$

$$\left(2n + \frac{3n}{8} + \frac{n}{4} + \frac{n}{64}\right) \log n$$

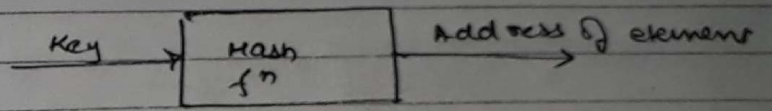
$$\left(\frac{128n + 24n + 16n + n}{64}\right) \log n$$

$$n\left(\frac{169}{64}\right) \log n$$

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

128
24
16
168

★ Hash Tables →



1. Folding method
2. Division Reminder method.
3. Multiplication method
4. Mid square method.

→ Collision Resolving Technique →

- Open Addressing

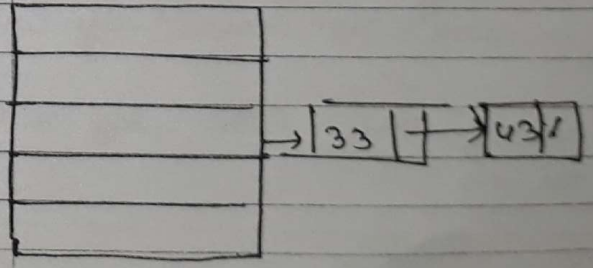
HASH-INSERT(T, K)

1. $i \leftarrow 0$
2. repeat $j \leftarrow h(K, i)$
3. if $T[j] = nil$
4. Then $T[j] \leftarrow K$
5. return j
6. else $i \leftarrow i + 1$
7. until $i = m$
8. error hash table overflow.

Here T is the Table
Table of hash slots
K is key element
& j is the address of element.

- Seperate chaining →

As in order values are found will be place in chains.



3. Linear Probing →
h, h+1, h+2, ...

b. Quadratic Probing →
$h; h+1^2, h+2^2, h+3^2, \ldots$
$h, h+1, h+4, h+9, \ldots$

- Double Hashing →
$k \pmod{m_1} \rightarrow \ldots$
$k \pmod{m_2} \rightarrow$

★ Binary search trees (B.S.T.) →
BST is organized in a Binary Tree which satisfies the following BST properties →
(i) Let 'x' be a node in BST if 'y' is a node in the left subtree of x then key of y
key[y] < key[x]
(ii) If y is node in the right subtree of x then key[y] ≥ key[x]

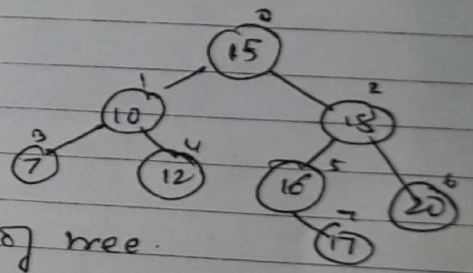
- INORDER - TREE WALK (x)
1. If x ≠ NIL
2. Then INORDER - TREE WALK (left[x])
3. Print key[x]
4. INORDER - TREE WALK (right[x])

Running time O(n) bcz n no of nodes in the tree.

- Querying a BST →

1. Searching → (root node of the tree, element to be search)
REC TREE - SEARCH (x, k)
1. If x = NIL or k = key[x] (actual element)
2. return x
3. If k < key[x]
4. Then return REC TREE - SEARCH (left[x], k)
5. else return REC TREE - SEARCH (right[x], k)

ITERATIVE - TREE - SEARCH
1. when x ≠ NULL & if k = key[x]
2. If k < key[x]
3. x = left[x]
4. else x = right[x]
5. Return x.



O(h) h = height of tree.

2. To find minimum & maximum element →
Tree - Minimum (x)
1. while left[x] ≠ NIL
2. x = left[x]
3. return x.
O(h)

Tree - Maximum (x)
1. while right[x] ≠ NIL
2. x = right[x]
3. return x
O(h)

If when left subtree is smaller then height of right & 0 when both are equal.

Scanned by CamScanner

3. To find successor of an element →
Tree - Successor (x) → node of which the successor is to be computed.

1. If right [x] ≠ NULL
2. return Tree - minimum (right [x])
3. else y := p[x]
4. while y ≠ NULL & x = right [y]
5. x := y → parent node.
6. y := p [y]
7. return y

O(h)

4. To insert an element in BST
Tree - insert (T, z)

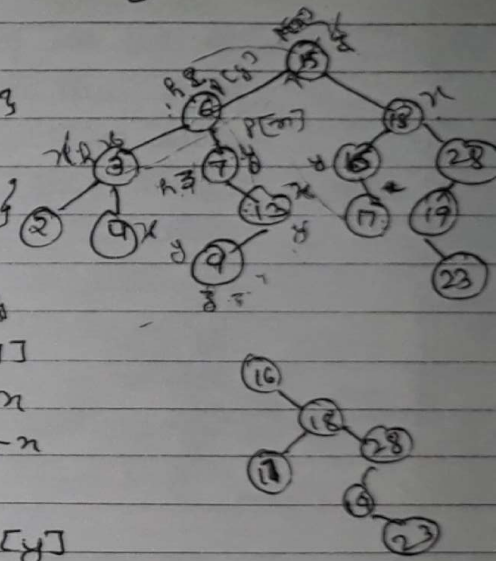
1. y := NIL
2. x := root [T]
3. while x ≠ NIL
4. {y := x
5. If key [z] < key [x]
6. x = left [x]
7. else x := right [x] }
8. p[z] := y
9. If y = NIL // Tree is empty
10. root [T] : z
11. If key[z] < key[y]
12. left [y] := z
13. else right [y] := z

O(h)

5. Deletion of an element from B.S.T →
Tree - Delete (T, z)

1. If left [z] = NIL or right [z] = NIL
2. then y ← z
3. else y ← Tree - Successor (z)
4. {If (left [y] ≠ NIL)
5. then x ← left [y]
6. else x ← right [y] }
7. {If x ≠ NIL
8. then p[x] ← p[y] }
9. { If p [y] = NIL
10. root [T] ← x }
11. else if y = left [p[y]]
12. then left [p[y]] ← x
13. else right [p[y]] ← x
14. If y ≠ z
15. Then key [z] ← key [y]
16. Return y

O(n)



Case I z is a leaf node z = 9
Case II z is having a single child z = 7
Case III z is having both the children z = 6

Scanned by CamScanner

★ Heap Sort →

- A Heap is a DS that is use to store or manage different element in the form of complete binary tree.

Heap is of two type

(i) max-heap →

It has the property that the value of parents node will always will be greater than valued child node.

Algo → max-heap

Parent(i)
return ⌊i/2⌋
left-child(i)
return 2i
Right-child(i)
Return 2i+1

(ii) min-heap →

The value of parent node ≤ child node.

- Heap sort & priority queue DS uses the following procedure →

(i) MAX-HEAPIFY →

It is use do maintain the max heap property & runs time θ(lgn)

(ii) BUILD MAX-HEAP →

This procedure produces max heap from an under-ed unordered i/p array & runs linear time θ(n)

(iii) HEAP-SORT →

This procedure actually sorts an array take O(nlgn) time.

(iv) Priority Queue →

The MAX-HEAP-INSERT, HEAP-INCREASE-KEY, HEAP-EXTRACT-MAX, HEAP-MAXIMUM allows the heap DS to be used as a priority queue and runs in O(lgn) time.

- HEAP-SORT(A)

1. BUILD-MAX-HEAP(A)
2. for i := length[A] down to 2
3. exchang A[1] with A[i]
4. heapsize[A] := heapsize[A]-1
5. MAX-HEAPIFY(A,1)

BUILD-MAX-HEAP(A)

1. HeapSize[A] := length[A]
2. for i = ⌊length[A]/2⌋ down to 1 (i=5 to 1; 10/2=5 ∴ i=5)
3. MAX-HEAPIFY(A,i)

MAX-HEAPIFY(A,i)

1. l = left(i) (2i=10 = l)
2. r = Right(i)
3. if l ≤ heapsize[A] & A[l] > A[i]
4. then largest := l
5. else largest := i
6. If r ≤ heapsize[A] & A[r] > A[largest]
7. largest := r
8. If largest ≠ i

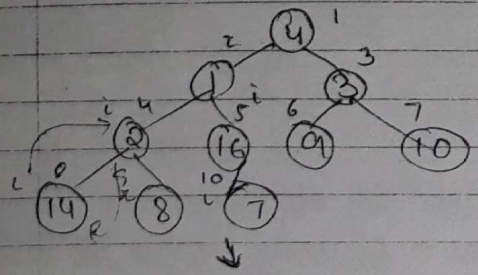
Scanned by CamScanner

DELTA Pg No. Date / /

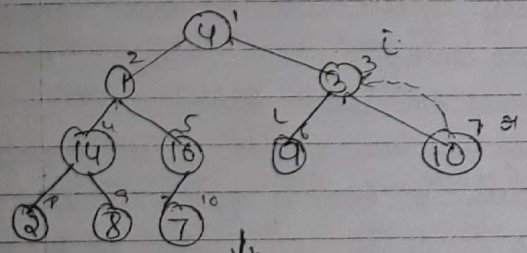
a. exchange A[i] with A[largest]

b. Max-HEAPIFY (A, largest)

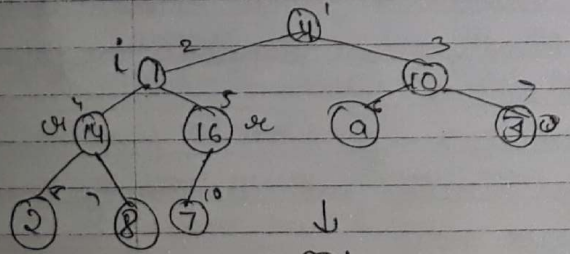
Example → 4, 1, 3, 2, 16, 9, 10, 14, 8, 7



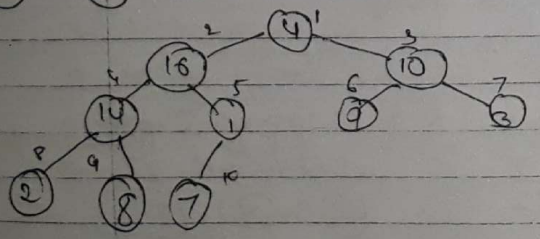
largest 8

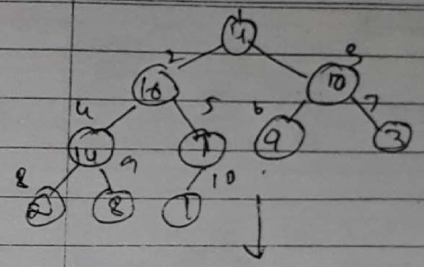


largest 6 then largest = 7

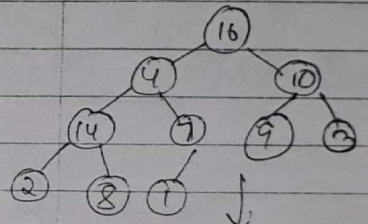


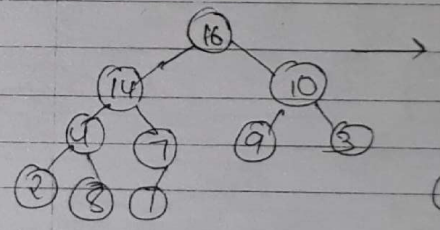
largest 4 then largest 1

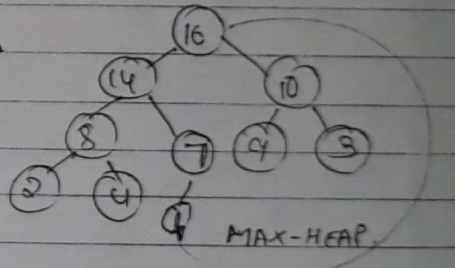




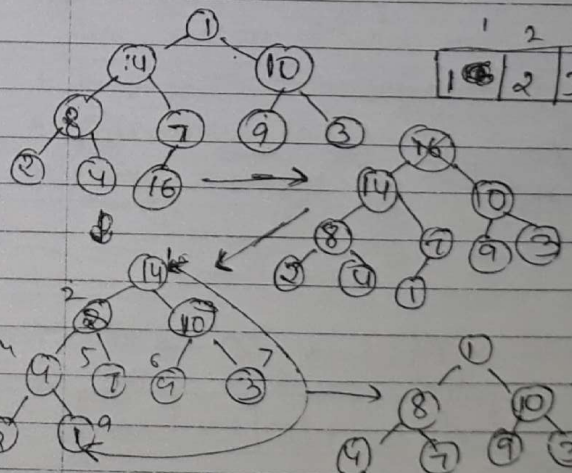
largest 16



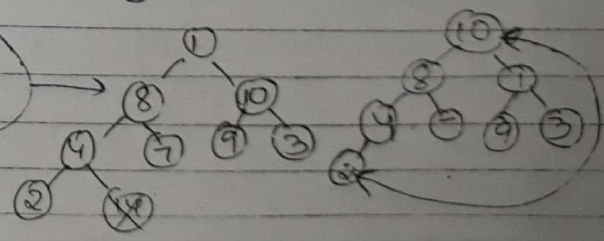


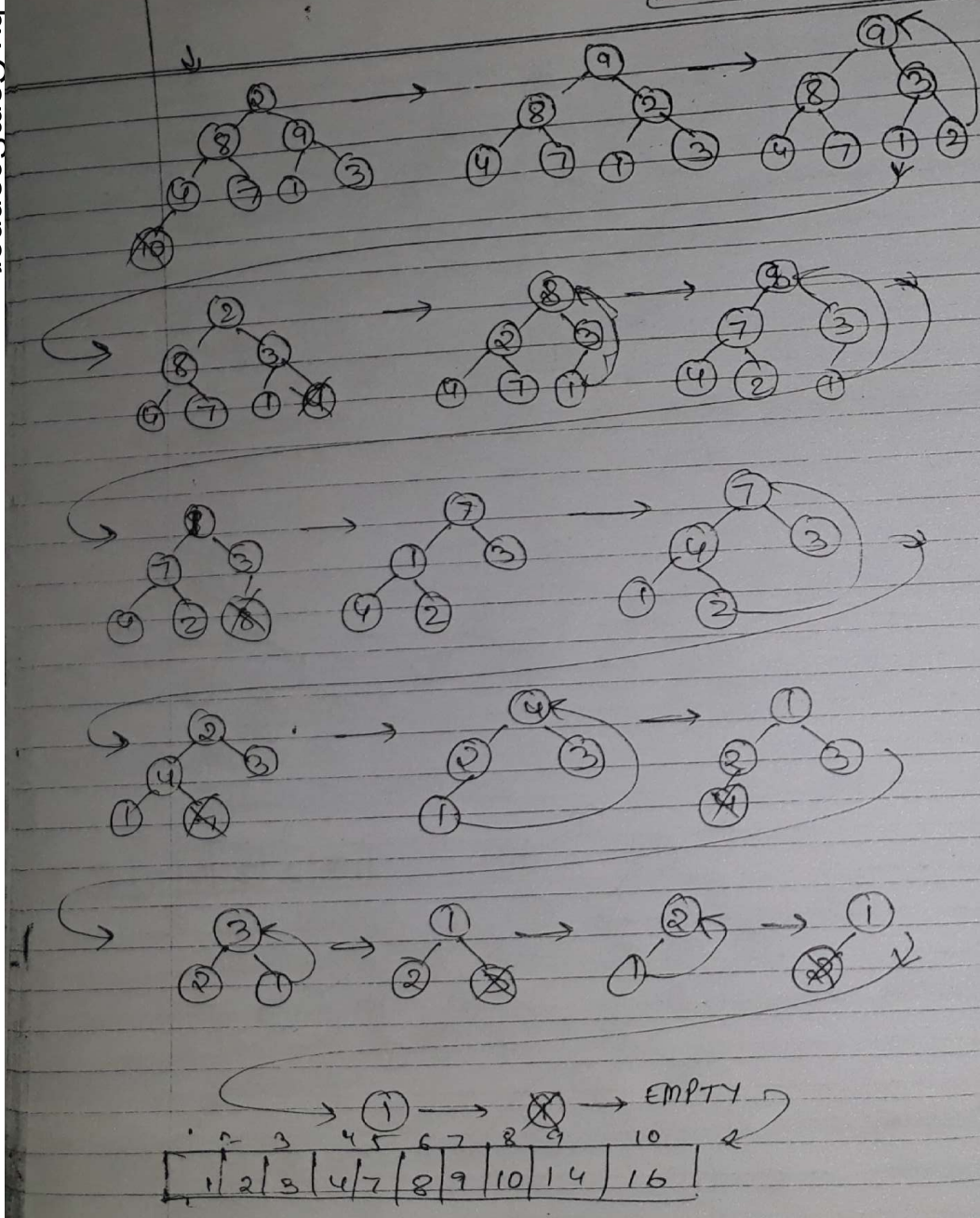


MAX-HEAP

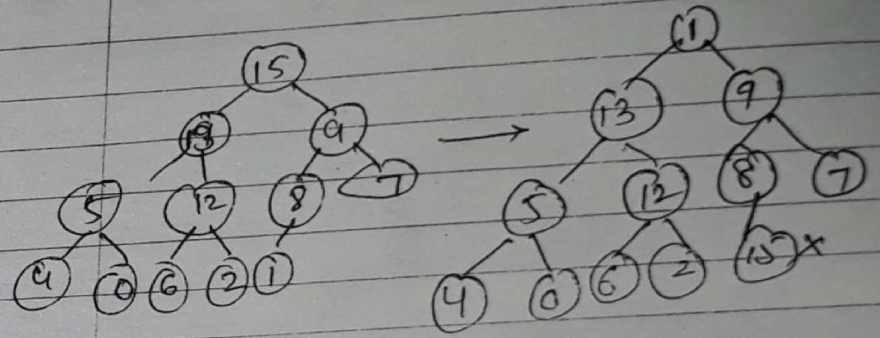


1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1



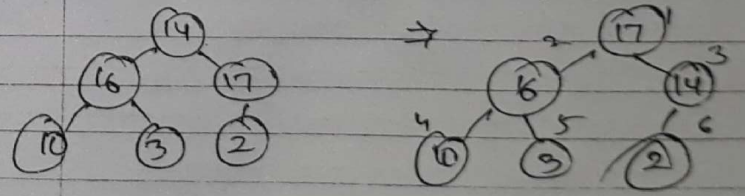


Ques 15, 13, 9, 5, 12, 8, 7, 4, 0, 6, 2, 1
using Heap sort.



→ ~~Application of Heap Sort~~ →

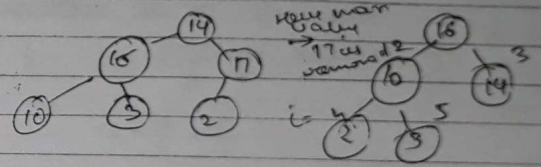
- ~~Heap sort~~ :- ~~Priority Queue DS~~
- Priority queue is the most popular appli of heap sort.
- It comes in two form
  (i) max priority queue (ii) min priority Qu
- Basically a Priority queue is DS for maint ning a set S of elements each ~~present~~ with an associated value known as heap.
- A max-Priority queue supports the following operation →
1. MAXIMUM (S)
   This algo returns the element of S with the largest heap.
   Ex→ 14, 16, 17, 10, 3, 2

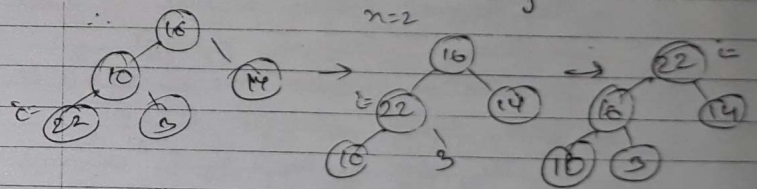


- Heap - maximum (S)
1. return A[1]
   O(1)
2. EXTRACT - MAX (S)
This algorithm removes and returns the element of S with the largest heap.
- Heap - extract max (S)
1. If heapsize [A] < 1
2. then write "Heap underflow" & exit
3. max := A[1]
4. A[1] := A [heapsize]
5. heapsize [A] := heap [A] - 1
6. MAX - Heapify (A, 1) O(log n)
7. return max.
3. INCREASE - KEY (S, x, K)
This algorithm ↑ the value of element x's key to the new value K which is assume to be atleast as large as x current key value
- Heap - increase - Key (A, i, key)
1. If A [i] > key
2. then write "Error: new key is smaller than current key" & exit
3. A[i] := key
4. while (i > 1) & & (A[i] > A (parent [i])
5. exchange A[i] & A [parent [i]]
6. i := parent [i]
   O (log n)

Here i is the node index at which replacement will be made & key is the new value which is to be replaced.

14, 16, 17, 10, 3, 2





4. MAX - Heap - Insert (A, key)
This algorithm inserts a new obj. element x into the set S such that S = S ∪ {x}
1. heapsize [A] := heapsize [A] + 1
2. A [heapsize [A]] := -∞
3. Heap - Increase - Key (A, heapSize [A], key)

O (log n)

Scanned by CamScanner

Scanned by CamScanner

★ Median - Order statistics →

- Median is the middle or average value of the list

Ex→ If the list is having n no of element then median can be calculated depending on whether n is odd or even.

$$\text{median} = \frac{n+1}{2} \quad \text{If 'n' is odd}$$

$$\frac{n}{2}, \frac{n}{2}+1 \quad \text{If 'n' is even}$$

- The ~~Neither~~ i^{th} order statistic of a set of n elements is the i^{th} smallest element

Ex→ The minimum the set of the element is the first order statistic i.e $i=1$.

The max of the n no of element will be the n^{th} order ~~Static~~ statistic

- MINIMUM - ORDER STATISTIC (A)

```
1. min := A[1]
2. for i = 2 to length[A]
3. { if A[i] < min
4.   min := A[i] }
5. return min
```

- MAXIMUM - ORDER STATISTIC (A).

```
1. max := A[1]
2. for i := 2 to length[A]
3. { if A[i] > max
4.   max := A[i] }
5. return max.
```

★ Median and order statistic →

- Algorithm to find the "Kth" smallest element

Suppose there are n elements in an array and we have to find K^{th} smallest element then the following needs to be followed.

1. If the ~~participant~~ Partitions element is at position a[j] then j-1 element are less than or equal to a[j]

   $a[j] \geq a[j-1]$

   and n-j element are $\geq j$
2. If $(K < j)$ then the Kth smallest element is in the left $A[1:j-1]$
3. If $(K = j)$ then the A[j] is the K^{th} smallest element

   If $(K > j)$ then K^{th} smallest element is the $(K-j)^{th}$ element in the list $A[j+1:n]$

Algo - Select (a, n, K)

Select the K^{th} smallest element in a[1:n] & place it in the K^{th} position of array such that the remaining element is rearrange in such a way that $a[m] \leq a[K]$ for $1 \leq m \leq K$ & $a[m] \geq a[K]$ $K+1 \leq m \leq n$

```
1. low := 1 , up = n+1
2. a[n+1] = ∞
3. repeat
4. {
5.   j = Partition (a, low, up)
6.   If K = j, return a[j]
```

7. If $K \le j$, then up = j
8. else low = j+1
   if

Algo - Partition (a, m, p)
1 v := a[m], i := m, j = p+1
2 repeat
3 { repeat
4 i := i+1
5 until (a[i] ≥ v)
6 repeat
7 j := j-1
8 until (a[j] ≤ v)
9 if (i < j)
10 then INTERCHANGE (a, i, j)
11 } until (i ≥ j)
12 a[m] := a[j], a[j] := v, return j

1. a[i] ≥ v i++
2. a[j] < v j--
3. i = j a[i] ↔ a[j]
4. i > j a[j] ↔ v

Algo INTERCHANGE (a, i, j)
1 temp := a[i]
2 a[i] := a[j]
3 a[j] := temp

Here a is the i/p array
n = total no of element
K = index of the smallest element that we wish to find
Partition ⇒ m = index of first element
p = index of last element of the array

low = 1, up = 10

Example

Index →	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	65	70	75	80	85	60	55	50	45	∞
v =										

K = 7

i	j
2	9

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	65	45	75	80	85	60	55	50	70	∞

i	j
3	8

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	65	45	50	80	85	60	55	75	70	∞

i	j
4	7

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	10
	65	45	50	55	85	60	80	75	70	∞

i	j
5	6

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	65	45	50	55	60	85	80	75	70	∞

i	j
6	5

now beg K > j, 75 > up, i ↓ up

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	60	45	50	55	65	85	80	75	70	∞

i	j
10	8

now beg K < j, 65 < up, j ↓ up

	(1)	(2)	(3)	(4)	(5)	now beg K > j, 65 > low, i ↑ low	(7)	(8)	(9)	(10)
	60	45	50	55	65	70	80	75	85	∞

i	j
7	6

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)
	60	45	50	55	65	70	80	75	85	∞

i	j
7	6

Scanned by CamScanner

Scanned by CamScanner

DELTA Pg No. Date

low pivot up

60 45 50 55 |65 70| 80 75 |85| ∞

i	j
8	8

60 45 50 55 |65 70| 75 |80 85| ∞

i	j
7	7

j=8 k=7 (7)

60 45 50 55 |65 70 75 80 85|

∴ a[7] = 75 is the 7th smallest element.

low up

(9)

	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	(10)	i j
K=5	65	70	75	80	85	60	55	50	45	∞	2 9
v=	(1)	(2)	(3)	(4)	(5)	(6)	(7)	(8)	(9)	10	i j
	65	45	75	80	85	60	55	50	45	∞	3 8
	65	45	50	80	85	60	55	75	45	∞	4 7
	65	45	50	55	85	60	80	75	45	∞	5 6
	65	45	50	55	60	85	80	75	45	∞	6 5
	60	45	50	55	[65]	85	80	75	45	∞	

i=4 j up

a[5] = 65

44

60 45 50 55 |65 70 75 80 85|

up

60 45 50 55 |65 70 75 80 85|

- Dynamic Order statistic →

It use to compute →

(i) Size →

It is defined as the no of nodes in the subtree including root.

(ii) Rank →

It is defined as the i^{th} smallest element and is computed as

$r = r + Size(left(x)) + 1$

where x is the corresponding node.

1. Retrieving an element with a given rank →

DS-Select (x, i) (x → root node, i → element we want to find)

r ← Size[left[x]] + 1

if i = r

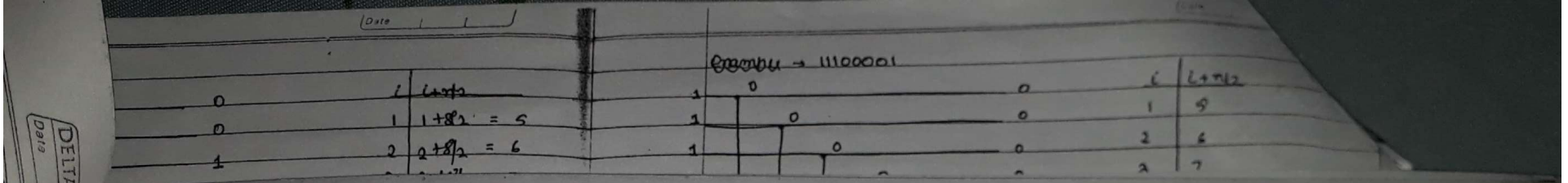
then return x

else if i < r

then return DS-Select (left[x], i)

else return DS-Select (right[x], i − r)

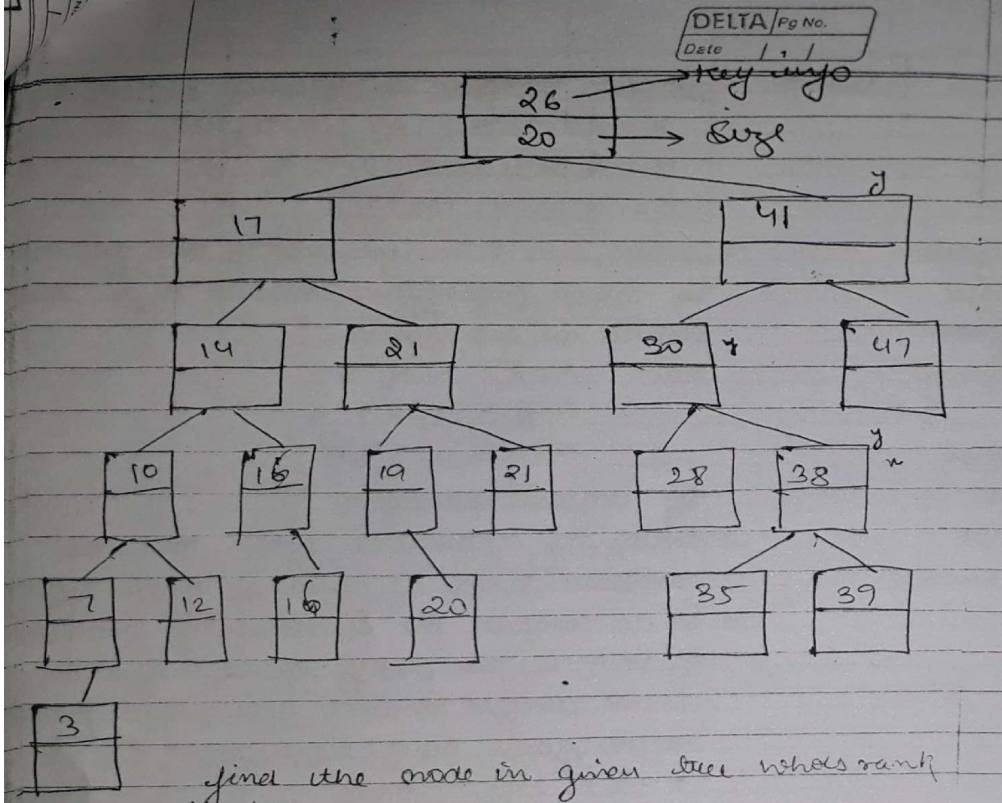
Scanned by CamScanner



i	Count
0	
1	1+82 = 5
2	2+82 = 6

Example → 11100001

i	Count
1	5
2	6
3	7



find the node in given tree whose rank is 17

1) $r = 17+1 = 18$ $i = 17$, $r = 18$
2) $n = 41$, $i := i - r = 3$
3) $r = 3+1 = 6$ $i = 4$ $i < r$
3) $n = 30$, $i = 4$
$r = 2$
$i > r$ $n = 38$, $i = 4-2 = 2$
4) $i = 2$ $r = 2$
$n = 38$

2. To find the rank of a given element
DS - Rank (T, x) → Tree, x → element whose rank is to be computed.

```
1 r ← size[left[x]]+1
2 y ← x
3 while (y ≠ root[T])
4 do if y = right[p[y]]
5 then r ← r + size[left[p[y]]+1]
6 y ← p[y]
7 return r
```

$r = 1+1 = 2$
$r = 2+1+1 = 4$
$r = 4+12+1 = 17$