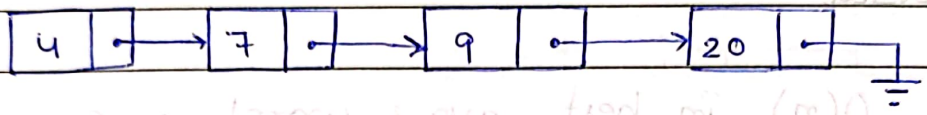
LINKED LIST

A linked list is a collection of nodes, which may or maynot be continuous. Each node can store data, along with a pointer to some other node.

Ex.




```
struct Node {
    int data;
    struct Node * next;
};
```

If m/m is allocated locally then it will be allocated on the stack, and get destroyed when respective activation record is deleted.
Hence the m/m is allocated on the heap:

```
struct Node *p = malloc(sizeof(struct Node));
```

* Common operations on linked list

(1) Accessing element
(2) Insertion : Beginning, B/w, End
(3) Deletion : Beginning, B/w, End
(4) Traversal

* Accessing element

Takes O(n) avg case, and worstcase. O(1) in best case.

* Traversal

Takes O(n) in best, avg & worst case.

* Insertion

(1) Beginning: O(1) all case.

(2) Inbetween: O(n) avg case.

(3) End: O(n) all case.

* Deletion

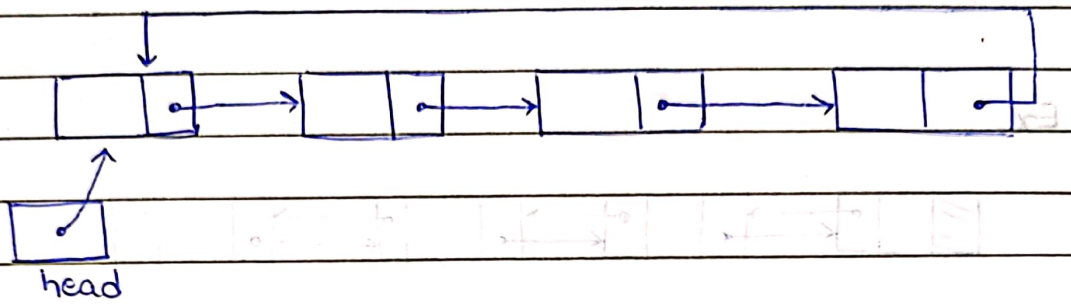
(1) Beginning: O(1) all case

(2) Inbetween: O(n) avg case

(3) End: O(n) all case.

* Circular Linked List

The last node, rather than pointing to NULL, points back to the first node.



Traversal:

```
process(head);
p = head->next;
while (p != head).
    process(p);
```

O(n) all case.

Insertion and Deletion:

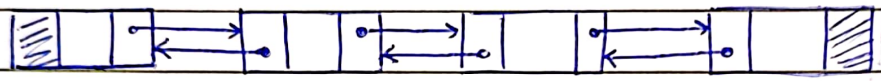
Beginning: O(1) all case
Inbetween: O(n) avg case.
End: O(n) all case

* Doubly Linked List

Each node has two pointers: Previous and Next. The previous pointer points to the previous node in addition to next pointer pointing to the next node.

```
struct DNode {
    struct DNode *prev;
    struct DNode *next;
    int data;
};
```

Ex.



Traversing: O(n)

Insertion & Deletion:

(1) Beginning: O(1)

(2) Inbetween: O(n)

(3) End: O(n)

ASYMPTOTIC ANALYSIS

* Goal of Asymptotic Analysis is to simplify analysis of running time by getting rid of details.

Ex:

$$3n^2 + 4 \simeq n^2$$

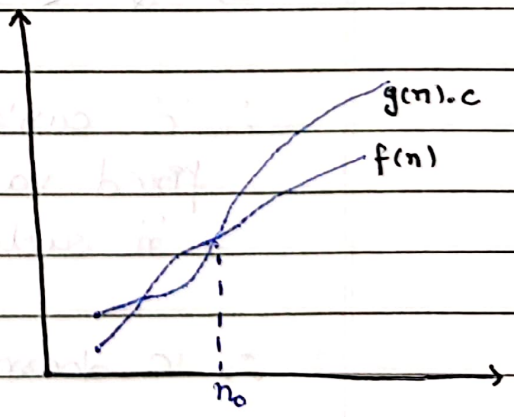
* Big-Oh Notation : (O)

$T(n)$ is $O(g(n))$, for some $g(n)$, if there exist a constant $c > 0$ and $n_0 \geq 0$, such that $\forall n > n_0$,

$$T(n) \leq cg(n)$$

Ex. $f(n) = n^2$ $\quad$ $g(n) = 2^n$

	n^2		2^n
1	1	<	2
2	4	=	4
3	9	>	8
4	16	=	16
5	25	<	32
6	36	<	64
7	49	<	128
8	64	<	256
9	81	<	512



$\therefore n_0 = 5$

Ex. Find (c, n_0) for $f(n) = n + \log n$.

$$n + \log n \leq cn$$

Taking $c = 2$.

$$n + \log n \leq 2n.$$

$$n_0 = 1.$$

$(2, 1)$. There could be infinitely many such pairs.

Ex. $2^{2n} = O(2^n)$ for which (c, n_0)?

$$2^{2n} \leq c \cdot 2^n$$

$$\Rightarrow 2^n \leq c.$$

∴ 'c' can't be a constant, since for a fixed value of 'c' we can always have a 'n' such that $2^n > c$.

∴ c doesn't exist.

∴ $2^{2n} \neq O(2^n)$.

Ex. $2^{n+1} = O(2^n)$ for which (c, n_0)?

$$2^{n+1} \leq c \cdot 2^n$$

$$\Rightarrow 2 \cdot 2^n \leq c \cdot 2^n$$

$$\Rightarrow 2 \leq c.$$

So, 'c' could be any value greater than 2.

So, $c=3$, $n_0=1$ is one such pair

* Big Omega Notation: (Ω)

$T(n)$ is $\Omega(g(n))$, for some func. $g(n)$, if there exists a constant $c>0$ & $n_0 \geq 0$, such that $\forall n \geq n_0$:

$$T(n) \geq c.g(n)$$

Ex. Is $10n^2 = \Omega(100n^2)$?

$$10n^2 \geq c.\ 100 \times n^2$$

for $c = \frac{1}{100}$, $n_0 = 1$.

Yes.

Ex. Is $6.2^n + n^2 = \Omega(2^n)$?

$$6.2^n + n^2 \geq c.2^n$$

for $c=1$, $n_0=1$.

Yes.

* Theta Notation :

(Θ)

- The problem with O & Ω notations is that they may or maynot convey the exact idea of Time complexity for an algorithm.

Ex. for $f(n) = 3n^2 + 2$.

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

but,

$f(n) = O(n^5) = O(n^7)$, etc.

and,

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

but

$f(n) = \Omega(n) = \Omega(1)$, etc.

So, we can report $f(n) = O(n^5)$ and be mathematically correct but, it won't help us make good estimate.

- Θ notation gives the following:

$$T(n) = \Omega(g(n)) \text{ and } T(n) = O(g(n))$$

$$\Updownarrow$$

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

There exists, c_1, c_2, and n_0 such that :

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

for all $n \geq n_0$.

Ex. $f(n) = 3n^2 + 2$.

$$c_1 \cdot n^2 \leq 3n^2 + 2 \leq c_2 \cdot n^2$$

$c_1 = 1$, $c_2 = 4$, $n_0 = 2$.

$\therefore 3n^2 + 2 = \Theta(n^2)$.

* $\log n < n < n \log n < n^2 < n^3 < 2^n < n!$

* • $f(n) \geq g(n) \iff f(n) = \Omega(g(n))$
  • $f(n) \leq g(n) \iff f(n) = O(g(n))$
  • $f(n) = g(n) \iff f(n) = \Theta(g(n))$

  • iff $f(n) = \Theta(g(n))$ then { (1) $f(n) = O(g(n))$ and (2) $f(n) = \Omega(g(n))$ }

* Asymptotic comparision only cares about the growth of a func.

  • We ignore constant terms.

Ex. (1) $n^2 + 5 \simeq n^2$

(2) $n^{5+n} \neq n^{2n}$

- We ignore multipliers.

Ex. (1) $2n^2 \simeq n^2$

(2) $2^{2n} \neq 2^n$

- When is it safe to ignore a constant?

Whenever a constant 'c' is such that the func. $f(n)$ can be written as

$f(n) = c.g(n)$ or $f(n) = g(n) + c$.

then it is safe to ignore the constant.

Ex. $f(n) = 2n^2 + 5$
$c_1 \quad c_2 \quad c_3$

$f(n) = 2[n^2 + 5/2]$

∴ ignore 2.

$f(n) = [n^2] + 5/2$

∴ ignore 5/2.

∴ $f(n) = n^2$

Ex. $f(n) = 2n^{5+n}$
$c_1 = c_2$

$f(n) = 2[n^{5+n}]$

∴ ignore 2.

No way to exclude 5.

∴ $f(n) = n^{5+n}$

Ex. Compare 2^n & n^n.

taking log.

$n \log 2$ $\qquad$ $n \log n$

$\Rightarrow n$ $\qquad$ $\Rightarrow n \log n$.

$\therefore 2^n \leq n^n$

Ex. Compare $n^2 \log n$ & $n (\log n)^{10}$.

cancelling the common factors

n $\qquad$ $(\log n)^9$

taking log.

$\log n$ $\qquad$ $9 \log \log n$.

K

K $\qquad$ $\log K$

$K > \log K$.

$\therefore n^2 \log n \geq n (\log)^{10}$

- For arbitrary ϵ & K,
  $n^\epsilon > (\log n)^K$

- If after taking log, the two funcs. come out to be equal then we can't comment about the original functions.

Ex. n^2 n^3

$2 \log n$ $3 \log n$

$\log n = \log n.$

But $n^2 < n^3$.

∴ Since after log both func. become equal they give no info. about original func.

For real no., a and b, if

$a < b$

then

$\log a < \log b.$

But its not always true for asymptotic analysis.

If asymptotically,

$a < b$

then,

either $\log a < \log b$

or $\log a = \log b$

asymptotically.

(log won't flip ~~order~~ asymptotic order)

* Procedure for log method :

1. Cancel out common terms.
2. Ignore constant for :
   - ① $f(n) = h(n) + c$
   - ② $f(n) = c \times h(n)$

   forms.
3. Take log on both the funcs, say $f(n)$ and $g(n)$.
   If :
   - ① $\log(f(n)) > \log(g(n)) \Rightarrow f(n) > g(n)$
   - ② $\log(f(n)) < \log(g(n)) \Rightarrow f(n) < g(n)$
   - ③ $\log(f(n)) = \log(g(n)) \Rightarrow$ Can't say

Ex. $f(n) = (\log n)^{\log\log n}$ $\quad g(n) = (\log \log n)^{\log n}$

let $n = 2^{2^k}$

$\Rightarrow (2^k)^k$ $\quad \Rightarrow (k)^{2^k}$

Taking log

$\Rightarrow k^2$ $\quad \Rightarrow 2^k \log k$

$\Rightarrow f(n) < g(n)$

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

Ex. $f(n) = n.2^n$ $\quad g(n) = 4^n$

$\Rightarrow (2^2)^n$

$\Rightarrow (2^n)^2$

Cancelling common terms.

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

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

Ex. $f(n) = n^2 \log n \qquad g(n) = n^{100}$

$\Rightarrow \log n \quad < \quad \Rightarrow n^{98}$

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

Ex. $f(n) = n^{\log n} \qquad g(n) = 2^n$

Taking log.

$\Rightarrow \log n \cdot \log n \qquad \Rightarrow n \log 2$

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

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

Ex. $f(n) = \log_2 n \qquad g(n) = \log_{10} n.$

$\Rightarrow \log_2 n \qquad \Rightarrow \frac{\log_2 n}{\log_2 10}$

ignoring const.

$\Rightarrow \log_2 n \quad = \quad \Rightarrow \log_2 n.$

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

* The base of log, doesn't affect asymptotic order.

Ex. $f(n) = 2^n$ $\quad$ $g(n) = n^{\sqrt{n}}$

Taking log.

$\Rightarrow n \log 2$ $\quad$ $\Rightarrow \sqrt{n} \log n$

$\Rightarrow n$ $\quad$ $\Rightarrow \sqrt{n} \log n$.

Cancelling common term.

$\Rightarrow \sqrt{n} = n^{1/2}$ $\quad > \quad$ $\Rightarrow \log n$.

$\therefore f(n) = \Omega(g(n))$

Ex. $f(n) = n^{\log_2 n}$ $\quad$ $g(n) = n^{\log_{10} n}$

Taking log.

$\Rightarrow \log_2 n \log_2 n \quad = \quad \Rightarrow \log_{10} n \log_2 n$

So, we can't say by log method.

$f(n) = n^{\log_2 n}$ $\quad$ $g(n) = n^{\log_{10} n}$

Base is same.

dividing by $n^{\log_{10} n}$ each side.

$\Rightarrow n^{\log_2 n - \log_{10} n}$ $\quad$ $\Rightarrow 1$

$\Rightarrow n^0$ $\quad$ $\Rightarrow 1$

$\therefore f(n) = \Omega(g(n))$

Ex. Compare 2^{2^n}, $n!$, 4^n, 2^n
A B C D

Comparing A and B.

2^{2^n}
$\Rightarrow 2^n \log 2$
$\Rightarrow 2^n$

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

$\therefore A > B$

Comparing C & D.

$4^n = (2^2)^n$
$\Rightarrow (2^n)^2$
$\Rightarrow 2^n$

2^n
$\Rightarrow 2^n$
$\Rightarrow 1$

$\therefore C > D.$

Comparing B & C.

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

$4^n = 2^{2n}$
$\Rightarrow 2n \times \log 2.$

$\therefore B > C.$

$\therefore A > B > C > D.$

* Little-Oh Notation:
(o)

$T(n)$ is $o(g(n))$, for some func. $g(n)$, for $\forall c > 0$, if there exists $n_0 > n$, such that

$$T(n) < c \cdot g(n)$$

- There are two minute diffs from Big-Oh:
  (1) Little-Oh requires $\forall c > 0$, Big-Oh needs some c.
  (2) Big-Oh had $T(n) \leq g(n)$, Little-Oh has strictly greater, $T(n) < g(n)$.

* Little Omega Notation: (ω)

$T(n)$ is $\omega(g(n))$, for some $g(n)$, if $\forall c > 0$, $\forall n_0 > n$,

$$f(n) > c.g(n)$$

* There are certain func. which aren't comparable.
  Ex. n, $n^{1+\sin x}$

* Properties:

1. $f(n) = \Theta(g(n))$ and $g(n) = \Theta(h(n)) \rightarrow f(n) = \Theta(h(n))$
2. $f(n) = O(g(n))$ & $g(n) = O(h(n)) \rightarrow f(n) = O(h(n))$
3. Similarly for Ω
4. Similarly for o
5. Similarly for ω

6. $f(n) = \Theta(f(n))$
7. $f(n) = O(f(n))$
8. $f(n) = \Omega(f(n))$

9. $f(n) = O(g(n)) \longleftrightarrow g(n) = \Omega(f(n))$

10. $f(n) = o(g(n)) \longleftrightarrow g(n) = \omega(f(n))$

11. $f(n) = o(g(n)) \rightarrow f(n) = O(g(n))$

12. $f(n) = \omega(g(n)) \rightarrow f(n) = \Omega(g(n))$.

* Comparision b/w $n!$ & n^n

- Stirling's Appx.:

$$n! \sim \sqrt{2\pi n}\left(\frac{n}{e}\right)^n$$

$n!$	n^n
$\sqrt{2\pi n}\left(\frac{n}{e}\right)^n$	$\Rightarrow n^n \times \left(\frac{e}{n}\right)^n$
$\Rightarrow \sqrt{2\pi n}$	$\Rightarrow e^n$
$\Rightarrow \sqrt{2\pi} \times \sqrt{n}$	$\Rightarrow e^n$
$\Rightarrow \sqrt{n}$	$\Rightarrow e^n$

$\therefore n! = O(n^n)$

& $n! = o(n^n)$

Ex. Compare: $n^{\sqrt{n}}$, 2^n, $n^{\log n}$, $n!$

A B C D

Comparing A & B

$\Rightarrow \sqrt{n}\log n \Rightarrow n$

$\Rightarrow \log n \Rightarrow \sqrt{n}$.

$A < B$

Comparing C & D

$\Rightarrow \log n \times \log n \Rightarrow n \log n$

$C < D$.

Comparing B & C.

$\Rightarrow n$ $\Rightarrow \log n \times \log n$.

$\Rightarrow \log^2 n$.

$B > C$.

Comparing A & C

$\Rightarrow \sqrt{n} \log n$ $\Rightarrow \log^2 n$

$\Rightarrow \sqrt{n}$ $\Rightarrow \log n$.

$A > C$.

Comparing B & D.

$\Rightarrow n$ $\Rightarrow n \log n$.

$B < D$.

$C < A < B < D$.

* $O(g(n))$, $\Omega(g(n))$, etc are not just single funcs, rather they are set of funcs.

$O(g(n))$ = Set of all funcs, $f(n)$, such that

$0 \leq f(n) \leq c \cdot g(n)$, $\exists c > 0$, $\forall n \geq n_0$.

And when we write

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

we actually mean,

$f(n) \in O(g(n))$.

We use $=$ instead of $\in$ for convinience's sake.

Ex. Consider,

$$g_1(n) = \begin{cases} n^3, & 0 \le n \le 10000 \\ n^2, & n > 10^4 \end{cases}$$

$$g_2(n) = \begin{cases} n, & 0 \le n \le 100 \\ n^3, & n > 100 \end{cases}$$

Always consider a large value which can be used for comparision, like, $10^4 + 1$

So, $g_1(n) = n^2$ $\quad$ $g_2(n) = n^3$

$\therefore g_1(n) = O(g_2(n))$

Ex. Which is false?

(A) $100\, n \log n = O\left(\frac{n \log n}{100}\right)$

(B) $\sqrt{\log n} = O(\log \log n)$ ✓

(C) If $0 < x < y$, $n^x = O(n^y)$

(D) $2^n \neq O(nK)$

B, since $\sqrt{K} > \log K$.

$K^\epsilon > (\log K)^m$

Ex. Compare: $3n^{\sqrt{n}}$, $2^{\sqrt{n}\log_2 n}$, $n!$
A B C.

A vs B.

$\Rightarrow n^{\sqrt{n}}$ $\Rightarrow 2^{\sqrt{n}\log_2 n}$

$\Rightarrow \sqrt{n}\log n$ $\Rightarrow \sqrt{n}\log n$.

Can't say.

B vs C.

$\Rightarrow \sqrt{n}\log\sqrt{n}$ $\Rightarrow n\log n$

$\Rightarrow 1$ $\Rightarrow \sqrt{n}$.

$\therefore$ B < C.

Similarly, A < C.

$\Rightarrow n^{\sqrt{n}}$ $\Rightarrow 2^{\sqrt{n}\log_2 n}$

$\Rightarrow 2^{\log_2 n^{\sqrt{n}}}$

$\Rightarrow n^{\sqrt{n}}$ $\Rightarrow n^{\sqrt{n}}$

$\therefore$ A = B.

$\therefore$ A = B < C.

Ex. Which is correct?

(A) $(n+K)^m = \Theta(n^m)$, K, m are const. ✓

(B) $2^{n+1} = O(2^n)$ ✓

(C) $2^{2n+1} = O(2^n)$

$(n+k)^m = n^m + m \times n^{m-1} \times K + \ldots = \Theta(n^m)$

$\therefore$ A is correct.

$\Rightarrow 2^{n+1} = 2 \cdot 2^n = 2$ $\Rightarrow 2^n = 1$.

$\therefore$ B is correct.

$\Rightarrow 2 \times 2^{2n}$ div by 2^n $\Rightarrow 2^n$

$\Rightarrow 2 \times 2^n$ $\Rightarrow 1$

$\therefore$ Incorrect.

Ex. Let $f(n), g(n), h(n)$ be funcs. defined on +ve Pnts such that,

$$\left.\begin{array}{l} f(n) = O(g(n)) \\ g(n) \neq O(f(n)) \end{array}\right\} \; g(n) > f(n)$$

$$\left.\begin{array}{l} g(n) = O(h(n)) \\ h(n) = O(g(n)) \end{array}\right\} \; g(n) = h(n)$$

Which is false.

(A) $f(n) + g(n) = O(h(n) + h(n))$
(B) $f(n) = O(h(n))$
(C) $h(n) \neq O(f(n))$
(D) $f(n)h(n) \neq O(g(n)h(n))$

B & C are trivially true.

A. $f(n) + g(n) = O(2 \cdot h(n))$
$g(n) = O(h(n))$
$\therefore f(n) < g(n)$
True.

D. $f(n)\,h(n) > g(n)\,h(n)$

False.

Ex. Compare: $\underset{A}{n^{1/3}}, \underset{B}{e^n}, \underset{C}{n^{7/4}}, \underset{D}{n \log^2 n}, \underset{E}{1.000000^n}$

$B > E > \ldots$

C vs D

$n^{7/4}$ $\quad$ $n \log^9 n$.

$\Rightarrow n^{3/4}$ $\quad$ $\Rightarrow \log^9 n$

$C > D$.

$D > A$.

$\therefore B > E > C > D > A$.

Ex. Compare: 2^n, $n^{3/2}$, $n\log_2 n$, $n^{\log_2 n}$

A B C D.

A vs D

$\Rightarrow 2^n$ $\quad$ $\Rightarrow n^{\log_2 n}$

$\Rightarrow n$ $\quad$ $\Rightarrow \log_2^2 n$

$A > D$.

B vs C.

$\Rightarrow n^{3/2}$ $\quad$ $\Rightarrow n\log_2 n$

$\Rightarrow n^{1/2}$ $\quad$ $\Rightarrow \log_2 n$

$B > C$.

B vs D

$\Rightarrow n^{\log_2 n}$ $\quad$ $\Rightarrow n^{3/2}$

$\Rightarrow \log^2 n$ $\quad$ $\Rightarrow \frac{3}{2}\log n$

$\Rightarrow \log n$ $\quad$ $\Rightarrow 3/2$

$D > B$.

$\therefore A > D > B > C$.

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

* $1^2+2^2+3^2+\cdots+n^2 = \frac{n(n+1)(2n+1)}{6}$

* $\sum_{k=1}^{n} \frac{1}{k} = 1+\frac{1}{2}+\frac{1}{3}+\cdots+\frac{1}{n} = \log n$

* $\sum_{k=1}^{n} \log k = n\log n$

* Analysing Loop Complexity

Ex. for (i: 0 to N-1)
    for (j: i+1 to N-1)
        stmt;

$$\sum_{i=0}^{N-1} \sum_{j=i+1}^{N-1} 1 = \sum_{i=0}^{N-1} (N-i-1) = N\times(N-0) - \frac{N\times(N-1)}{2} - N$$

$$= N(N-1) - \frac{N(N-1)}{2}$$

$$\Rightarrow \frac{N(N-1)}{2}$$

$\therefore O(N^2)$

Ex. for (i=1; i<=N; i=i*2)
stmt;

i = 1 = 2^0
i = 2 = 2^1
i = 4 = 2^2
i = 8 = 2^3
:
i = 2^K = N

$2^K = N$
$\Rightarrow K = \log_2 N$
$\therefore O(\log N)$

Ex. for (i=1; $i^2 \leq N$; i++)
stmt;

i = 1
i = 2
:
i = K $\Rightarrow K^2 = N \Rightarrow K = \sqrt{N}$.

$\therefore O(\sqrt{N})$

Ex. for (i=1; $i \leq 2^N$; i=i*2)
stmt;

i = 1
i = 2
i = 2^2
:
i = 2^K = 2^N

$\therefore K = N$
$\therefore O(N)$

Ex. for (i=2; i≤N; i = i^2)
stmt;

$i = 2 = 2^{2^0}$
$i = 2^2 = 2^{2^1}$
$i = (2^2)^2 = 2^4 = 2^{2^2}$
$i = (2^4)^2 = 2^8 = 2^{2^3}$
⋮
$i = 2^{2^K} = N$

$\Rightarrow 2^K = \log N$
$\Rightarrow K = \log \log N$.
$\therefore O(\log \log N)$

Ex. for(i=n^2; i>=2; i = √i)
stmt;

$i = n^2 = (n^2)^{\frac{1}{2^0}}$
$i = \sqrt{n^2} = n = (n^2)^{\frac{1}{2}}$
$i = \sqrt{n} = n^{1/2} = (n^2)^{\frac{1}{2^2}}$
$i = \sqrt{n^{1/2}} = n^{1/4} = (n^2)^{\frac{1}{2^3}}$
⋮
$i = (n^2)^{\frac{1}{2^K}} = 2$.

$\Rightarrow \frac{1}{2^K} = \log_{n^2} 2 = \frac{1}{\log_2 n^2}$
$\Rightarrow 2^K = \log_2 n^2$
$\Rightarrow K = \log_2 \log_2 n^2$

$\Rightarrow \log_2 (2 \times \log_2 n)$
$\Rightarrow \log_2 2 + \log_2 \log_2 n$

$\therefore O(\log_2 \log_2 n)$.

Ex. for (i=1; i<=N; i++)
    for (j=1; j<=i²; j++)
        Stmt;

$$\sum_{i=1}^{N} \sum_{j=1}^{i^2} 1 = \sum_{i=1}^{N} i^2 = \frac{n(n+1)(2n+1)}{6} \quad \therefore O(N^3)$$

Ex. for (i=1; i<=N; i++)
    for (j=1; j<=i; j+=i)
        Stmt;

j: 1, i, 2i, 3i ... Ki = i

$\therefore K = 1$

$$\sum_{i=1}^{N} \sum_{j=1}^{i} 1 = \sum_{i=1}^{N} 1 = N$$

$\therefore O(N)$.

Ex. for (i=1; i<=N; i++)
    for (j=1; j<=N; j+=i)
        Stmt;

j: 1, i, 2i, ... Ki = N

$\therefore K = \frac{N}{i}$

$$\sum_{i=1}^{N} \frac{N}{i} = N \sum_{i=1}^{N} \frac{1}{i} = N \log N$$

$\therefore O(N \log N)$

Ex. for (i=1; i<=N; i+=1)
    for (j=1; j<=i; j=j*2)
        Stmt;

j: 1, 2, 2², 2³, ... $2^K = i$

$K = \log i$

$$\Rightarrow \sum_{i=1}^{N} \log i = \log \prod_{i=1}^{N} i$$

$$\Rightarrow \log N! = N \log N$$

$$\therefore O(N \log N)$$

Ex. for (int i = 1; i <= N; i = i * 2)
for (j = 1; j <= i; j++) } j : 1, 2, ... K = i
stmt; $\therefore K = i$

$i = 1$	$K = 1$
$= 2$	$= 2$
$= 2^2$	$= 2^2$
$= 2^3$	$= 2^3$
⋮	⋮
$2^K = N$	$= 2^K$

$$\Rightarrow 1 + 2 + 2^2 + \dots + 2^K$$
$$\Rightarrow 2^{K+1} - 1$$
$$\Rightarrow 2 \cdot 2^K - 1$$
$$\Rightarrow 2 \cdot 2^{\log_2 N} - 1$$
$$\Rightarrow 2 \cdot N - 1$$

Ex. for (i = 1; i <= N; i = i * 2)
for (j = 1; j <= i^2; j++) } j : 1, 2, ... $K = i^2$
stmt;

$i = 1$	$K = 1$
$= 2$	$= 2^2$
$= 2^2$	$= (2^2)^2$
$= 2^3$	$= (2^3)^2$
$= 2^4$	⋮
⋮	
$\Rightarrow 2^K = N.$	$\Rightarrow (2^K)^2$

$\Rightarrow 1 + 2^2 + 2^4 + 2^6 + \dots + 2^{2K}$

$\Rightarrow 1 + 4 + 4^2 + 4^3 + \dots + 4^K$

$\Rightarrow \frac{4^{K+1} - 1}{3}$

$\Rightarrow \frac{1}{3}(4 \times 4^{\log_2 N} - 1)$

$\Rightarrow \frac{1}{3}(4 \times N^2 - 1)$

$\therefore O(N^2)$

- Ex.
```
for (i=2; i<=n; i = i^2)
    for (j=1; j<=n; j++)
        for (K=1; K<=n; K=K+j)
            Stmt;
```
$K = 1, j+2j, \dots, Kj = n$

$K = \frac{n}{j}$

for j:	1	$K = \frac{n}{1}$
	2	$\frac{n}{2}$
	⋮	
	n	$\frac{n}{n}$

$\sum_{j=1}^{n} \frac{n}{j} = n \log n.$

for i:

$i = 2^{2^0}, 2^{2^1}, 2^{2^2}, 2^{2^3}, \ldots 2^{2^K} = n$.

$K = \log_2 \log_2 n$.

$\therefore \log_2 \log_2 n \times (n \log_2 n)$

$\Rightarrow (n \log n)(\log \log n)$

* Stack

Stack is a First-in-Last-out data structure in which data can be inserted and extracted from only one side.
At any time only a single element is accessible: the top element.

* Stack Operations:

(1) Push: Puts a single element on the stack top.
(2) Pop: Removes a single element (top) from stack.
(3) Top: Access top element of the stack.

* Stack Permutation:

Given the sequence of push operations, a valid stack permutation is the order in which elements are popped off of the stack.

Ex. 1,2,3,4. Which isn't valid stack perm.?

(A)	1, 2, 3, 4	P(1), P(), P(2), P(), P(3), P(), P(4), P()
(B)	2, 1, 3, 4	P(1), P(2), P(), P(), P(3), P(4), P(), P()
(C)	4, 2, 3, 1	✓ No sequence possible.
(D)	3, 4, 2, 1	P(1), P(2), P(3), P(), P(4), P(), P(), P()

- Total no. of valid stack permutations for n elements is given by:

  - $\frac{^{2n}C_n}{n+1}$ { Catalan No. }

No. of invalid permutations = $n! - \frac{^{2n}C_n}{n+1}$

* Implementation

(i) Array

```
int stack[capacity];
int top = -1;

push (x) {
    top += 1;
    if (top == capacity):
        top -= 1;
        raise StackFullError

    stack[top] = x;
}

pop (x) {
    if (top == -1):
        raise StackEmptyError
    x = stack[top];
    top -= 1;
    return x;
}
```

(2) Linked List

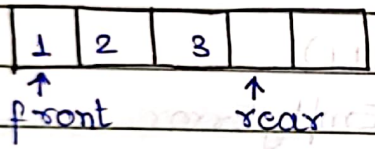
```
LLNode *stack;

push (x) {
    LLNode *new = malloc(...);
    new→data = x;
    new→ next = stack;
    stack = new;
}

pop () {
    if (stack == NULL)
        raise StackEmptyError

    x = stack→data;
    Stack = stack→ next;
    return x;
}
```

- Push and Pop both are O(1) operations.
- For array implementation, size should be fixed in advance.

* Queue

A Queue is a First-in-First-out data structure, in which data is inserted from one end and extracted from other end.

A queue has two pointers : front and end.
"front" points to the next element to be extracted.
"rear" points to the next place where element will get inserted.



* Queue Operations

1. Enque : Insert element into the queue.
2. Deque : Remove element from the queue.

* Implementation

1. Array.

```
int Queue [capacity];
int f = -1, r = -1;
```

Linear

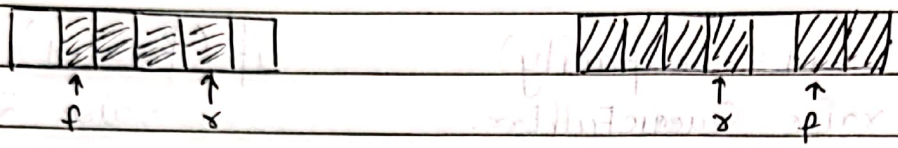
```
enqueue(x) {
    if (r+1 == capacity)
        raise QueueFullErr.
    r = r+1;
    Queue[r] = x;
    if (r == 0)
        f = 0;
}
```

```
dequeue() {
    if (f == -1)
        raise QEmptyErr.
    x = Queue[f];
    if (f == r)
        f = r = -1;
    else f = f+1;
    return x;
}
```

Circular

```
enqueue(x) {
    if (r != -1 && (r+1)%capacity == f)
        raise QueueFullError.
    r = (r+1)%capacity;
    Q[r] = x;
    if (f == -1)
        f = 0;
}
dequeue() {
    if (f == -1)
        raise QEE;
    x = Queue[f];
    if(f == r)
        f = r = -1;
    else
        f = (f+1)%capacity;
    return x;
}
```

- Elements are always between front & rear.



(2) Linked List

```
LL Node *f, *r;

enqueue(x) {
    LLNode *n = malloc(..);
    n->data = x;
    if (r != NULL)
        r->next = n;
    r = r->next;
}
```

```
dequeue() {
    if (f == NULL)
        raise QEE;
    x = f->data;
    f = f->next;
    if (f == NULL) r = NULL;
    return x;
}
```

- Both Enqueue and Dequeue operations run in O(1).

* Queue using 2 Stacks.

```
Stack S1, S2;

enqueue(x) {
    push(S1, x);
}
```

```
dequeue() {
    if (isEmpty(S2)) {
        if (isEmpty(S1)) {
            raise QEE;
        while (!isEmpty(S1))
            push(S2, pop(S1));
    }
    x = pop(S2);
    return x;
}
```

- Enqueue operation takes O(1) in all cases.
  Dequeue takes O(n) in worst case.

* Stack using 2 Queues:

```
queue q1, q2;
int curr = 1;
push(x) {
    if curr == 1
        enqueue(q1, x);
    else
        enqueue(q2, x);
}
```

```
pop() {
    if (isEmpty(q1) && isEmp(q2))
        raise SEE;
    if (curr == 1) {
        while (1) {
            x = dequeue(q1);
            if (!isEmpty(q1))
                enqueue(q2, x)
            else
                break;
        curr = 2;
        return x;
    }
```

```
        else {
            while(1) {
                ;
            }
            curr=1;
            return X;
        }
    }
```

- push takes O(1).
  pop takes O(n) in all cases.

* Balancing Bracket using Stack:

```
Bal (s) {
    stack s;

    for (char c in s) {
        if ( c == '(' or '[' or '{' )
            push (s, c);
        if ( c == ')' or ']' or '}') {
            if ( isEmpty(s) or ! isMatching(c, top(s)))
                return "Unbalanced";
            pop (s);
        }
    }
    if (! isEmpty (s)) return "Unbalanced";
    return "balanced";
}
```

* 2 Stacks 1 Array

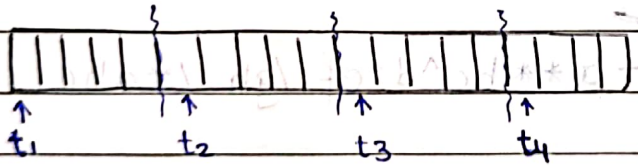
Implemented by starting the stack from two ends and growing in opposite direction.

Initial: $t_1 = -1$ $t_2 = n$.

1 element: $t_1 = 0$ $t_2 = n-1$

Stack full: $t_1 + 1 == t_2$ or $t_1 == t_2 - 1$

- For more than 2 stacks using single array, we'll have to divide ~~stacks~~ array in parts.



* Infix, Prefix, Postfix Notation.

Prefix → Polish, Postfix → Reverse Polish

~~Name~~

Name is based on the position of the operator.

Ex. $x + y - z / e * a \wedge b / c + d$

Prefix: $x+y- (/ze)*(\wedge ab) /c +d$

$+[-(+xy)[/(*(/ze)(\wedge ab))c]]d.$

Postfix: {(xy+)[[(ze/) (ab^*) *] c /] - }d +

Ex. (a+b * c - d) / (e * f)

Postfix: a bc * + d - ef * /

Prefix: / - + a * bc d * ef

Ex. a + b * c * d ^ e ^ f - g / h + a * b / c

Postfix: a + (bc * def ^ ^ *) - (gh /) + (ab * c /)
⇒ abc * def ^^ * a + gh / - ab * c / +

Prefix: a + (** bc ^d ^ef) - (/gh) + (/* abc)
~~⇒ + / * abc~~
⇒ + - + a ** bc ^d ^ef /gh /* abc.

Ex. fn (a, b, c) {, is L to R assoc.}

Postfix: ab, c, fn
Prefix: fn ,, abc

* Infix to Postfix Conversion

Algo: *
Inp: &

if ('(')
    push on stack
if (')')
    pop until '(' is popped.

if (operator)
    if (precedence(top) > precedence(curr))
        pop. perform top operation. push.
    else if (same prec.)
        ~~push~~ pop. perform {except for $ ^ }
    else
        push.

* Postfix Evaluation

Algo:
Inp: &
    if (operator)
        pop operand. perform operation. push result.
    else
        push operand.

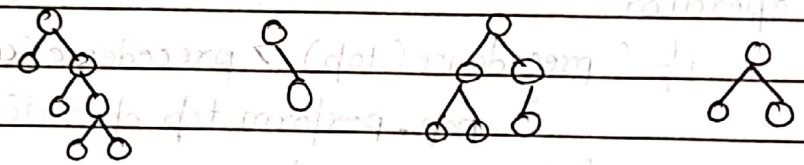
Return final result.

BINARY TREE

Binary Tree is a data structure made up of nodes, where each node can have 0, or 1, or 2 child nodes.

- Full Binary Tree: Every node has 0 or 2 children.
- Complete Binary Tree: All levels except the last are completely filled. Last level is filled from left to right.

Ex.



Full	✓	x	x	✓
Complete	x	x	✓	✓

- Perfect Binary Tree: A CBT, where even the last level is fully filled.

* Depth (or level) of a Node: The depth of a node is the no. of edges b/w the root node to the particular node.

Height of a Node: The height of a node is the no. of edges in the longest path from the node to a leaf.

Height of the Tree: Height of the root node is the height of the tree.

* There is a rel. b/w nodes with 2 children and leaf nodes in Binary Tree.

$$\#\text{Leaf} = \#\text{2-children} + 1$$
$$L = I_2 + 1$$

So total no. of nodes in a BT:

$$\text{Total} = \#\text{2-children} + \#\text{1-child} + \#\text{Leaf}$$
$$\Rightarrow I_2 + I_1 + L$$
$$\Rightarrow I_2 + I_1 + (I_2 + 1)$$
$$\Rightarrow 2I_2 + I_1 + 1$$

- Special Case: Full Binary Tree a.k.a. 2-Tree, there are only 2-children nodes or leaf nodes.

$$\text{So, Total} = 2I_2 + 0 + 1$$
$$\Rightarrow 2I_2 + 1.$$

- Special Case: For a K-tree, where K can be 2, 3, ...

$$\#\text{Leaf} = \#\text{2-children} \times (K-1) + 1$$
$$L = I_2 \times (K-1) + 1$$

$$\therefore \text{Total} = KI_2 + 1$$

{ K-Tree ≡ Full K-ary Tree }

Ex. Consider a perfect B.T with n nodes.

(1) No. of leaf nodes?

$$I_2 + L = n$$
$$\Rightarrow 2I_2 + 1 = n$$
$$\Rightarrow I_2 = \frac{n-1}{2}$$

$$L = \frac{n-1}{2} + 1 = \frac{n+1}{2}$$

(2) Height of tree?

$$n = 2^{h+1} - 1$$
$$\Rightarrow (n+1) = 2^{h+1}$$
$$\Rightarrow h = \log_2 (n+1) - 1$$

Ex. Consider a CBT of height 'h'.

(1) Min. no. of nodes?

$$\Rightarrow (2^h - 1) + 1 = 2^h$$

(2) Max. no. of nodes?

$$\Rightarrow 2^{h+1} - 1$$

Ex. Consider a BT with n nodes.

(1) Min height

$$n = 2^{h+1} - 1$$
$$\Rightarrow n+1 = 2^{h+1}$$
$$\Rightarrow h = \log(n+1) - 1$$

(2) Max height

$\Rightarrow n-1$

* Representation of BT

* Array Based

Only suitable for CBT

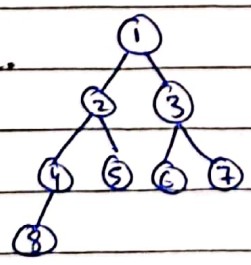
- For an index $i \geq 0$,
  left child index $= 2i+1$
  right child index $= 2i+2$.

- For an index $i > 0$,
  parent index $= \left\lceil \frac{i}{2} \right\rceil - 1$

- For index $i \geq 1$.
  left child index $= 2i$
  right child index $= 2i+1$

- For an index $i > 1$
  parent index $= \left\lfloor \frac{i}{2} \right\rfloor$

Ex.

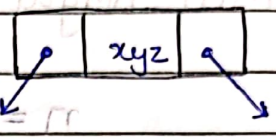


$\Rightarrow$

1	2	3	4	5	6	7	8

* Node Based.

```
struct Node {
    struct Node* lptr;
    int data;
    struct Node* rptr;
}
```



* . No. of binary trees structures possible with n unlabelled nodes

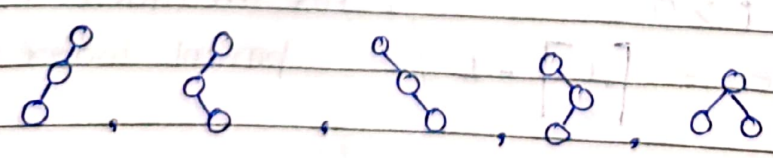
$$\frac{^{2n}C_n}{n+1}$$

. No. of binary trees possible with n distinct nodes

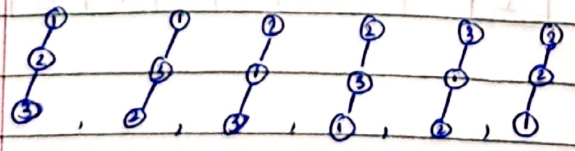
$$\frac{^{2n}C_n}{n+1} \times n!$$

Ex. $n = 3$.

structures = $\frac{^{6}C_3}{4} = 5$.



BTs = $5 \times 3! = 30$

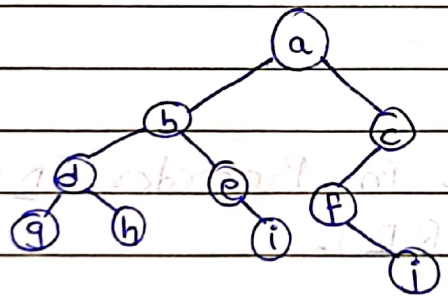


* Traversals

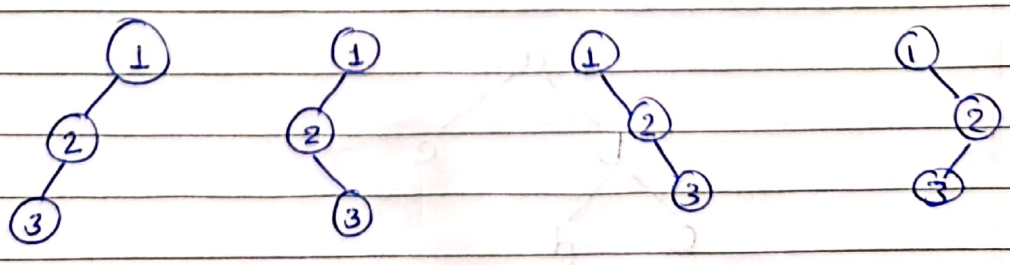
* Preorder: Data Left Right
* Inorder: Left Data Right
* Postorder: Left Right Data.

- Tree can be constructed from Inorder + Preorder/Post-order traversal.

Ex. Pre: a b d g h e i c f j
In: g d h b e i a f j c.



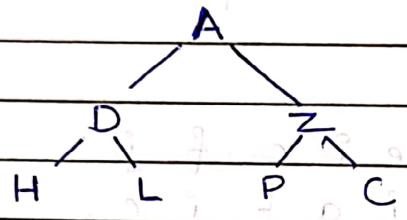
Ex. Pre: 1 2 3
Post: 3 2 1



Ex. Construct Full BT.

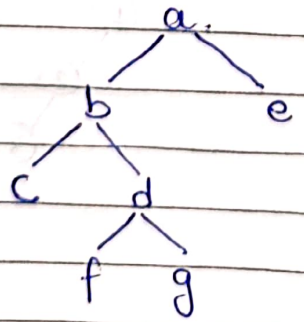
Pre: A, D, H, L, Z, P, C
Post: H, L, D, P, C, Z, A

Every node either has 2 or 0 children.



whatever comes first in Preorder (DLR) comes last in Postorder (LRD).

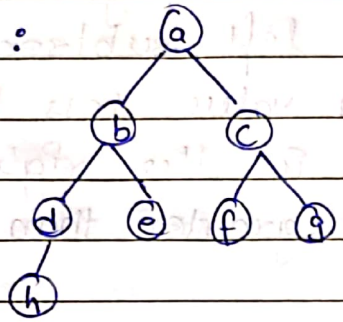
Ex. Full BT using:
Pre: a b c d f g e.
Post: c f g d b e a.



Ex. Construct Complete BT.

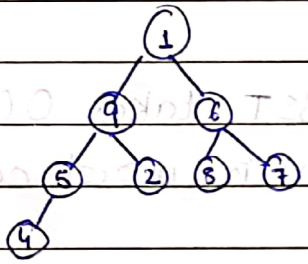
Pre: 1 9 5 4 2 6 8 7

CBT with 8 nodes:



Preorder: a b d h e c f g

∴ Req. tree is:



*. For general BT:

(1) Pre + In

or (2) Post + In.

- For Full BT even Pre + Post can be used, as well.
- For CBT any single traversal is enough.

* Binary Search Tree

A Binary Tree satisfying the property: all the nodes in the left subtree of a node K must contain value less than K's value and all the nodes in the right subtree must contain value greater than K's value.

- Inorder traversal of a BST gives ascending-order sorting of the elements.
- Searching in a BST takes O(logn) in avg and avg case. O(n) in worst case. O(1) in best case.
- Insertion in a BST takes O(logn) in avg case and O(n) in worst case. O(1) in best case.

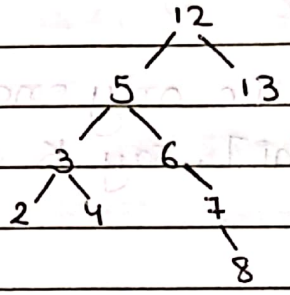
* Deletion in BST

There can be three cases for deletion depending on the no. of children the node to be deleted has.

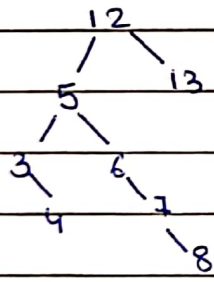
1. 0 children: Delete the node directly.
2. 1 child: Replace the node with it's child, then delete it.
3. 2 children: Find either the inorder predecesor

or successor of the node, X. Remove X from its position and replace current node with X, then delete curr. node.

Ex.

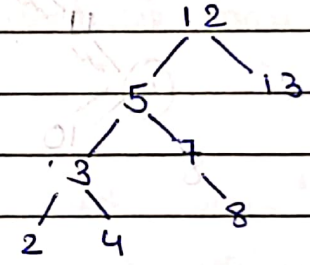


(1) Delete (2)



simply deleted.

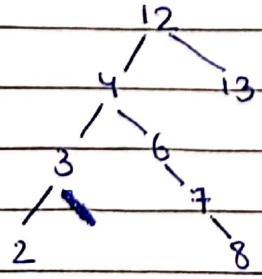
(2) Delete (6)



Replaced with single child then deleted.

(3) Delete (5)

Using inorder predecesor.



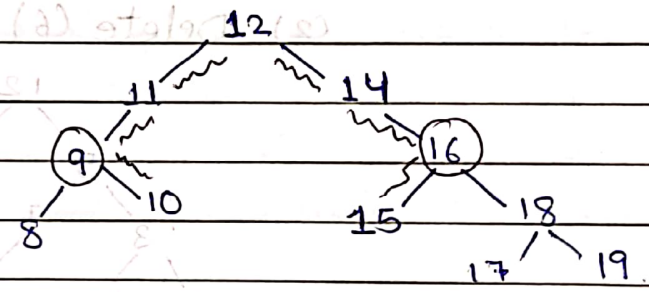
Removed 4 from its position. Replaced 5 with 4. Deleted 5.

* BST search in range

Given range [lo, hi] search all the values in the BST in that range.

In the best case, we only encounter nodes in the range [lo, hi], say K such nodes.

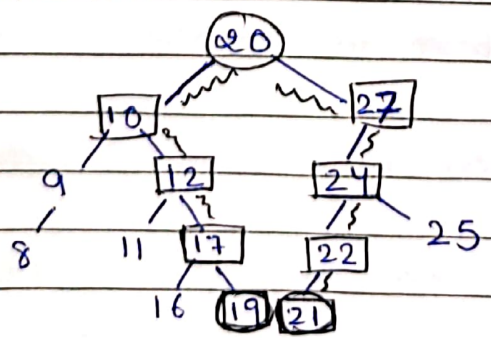
Ex. [10, 15]



So we only encountered 2 nodes out of range.

In the worst case, we may encounter 2 nodes at each level at most, which are out of range.

Ex. [19, 21]



We encountered 10, 12, 17, 27, 24, 22, i.e. at most 2 nodes at each level, along with the K

nodes in range.

Eg. for height h, T.C. is $O(2h + K)$

$\because h = O(\log n)$ for n nodes.

$$T.C = O(2\log n + K)$$
$$= O(\log n + K)$$

- Questions requiring such analysis may come.

* When searching for a node with val. 'K' in a BST we may encounter nodes with value less than or more than K, but all the nodes with value less than K encountered, must be in increasing order, and values greater than K in decreasing order.

Ex. Say while searching for 45, we encounter 12, then why would we go to left of 12? We would only go in right. Next we see 18, again going left is wrong, and so on. Hence if val < 45, it must appear in increasing order.
Vice-versa for val > 45.

Ex. Which seq. of search is correct while looking for 55?

(A) 90, 12, 68, 34, 62, 45, 55 ✓

(B) 9, 85, 47, 68, 43, 57, 55

A: >45: 90, 68, 62, Decreasing ✓

<45: 12, 34, 45 Increasing ✓

B: <45: 9, 47, 43 Increasing X

Ex. While searching for 60 in BST the following were encountered (not in that order): 10, 20, 40, 50, 70, 80, 90. How many diff. search orders are possible?

For a search order vals <60 must be increasing & >60 must be decreasing.

<60: 10, 20, 40, 50

>60: 70, 80, 90.

Total no. of orders = 7!

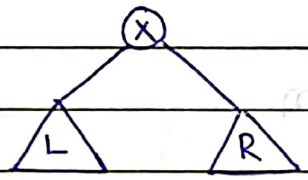
Orders with <60 increasing & >60 decreasing = $\frac{7!}{4!\,3!}$

$\Rightarrow 35$

* AVL Tree

A BST can be skewed, i.e. left or right skewed. Since all three operations: Insertion, Deletion and Search all depend on the height, i.e, O(h), when BST is skewed, the operations take O(n) {∵ h=n}.

To overcome this, AVL tree came into picture. AVL tree has the property, for every node, the diff. b/w the height of its left and right subtree is b/w -1 and 1.



height (L) - height (R) ∈ {-1, 0, 1}

This diff. b/w height of left and right subtree's of X, is called as Balance Factor (BF) of X.

∴ Balance factor of node X = height (X_L) - height (X_R).

So, AVL tree is a BST in which, every node has a balance factor is b/w -1 and 1.

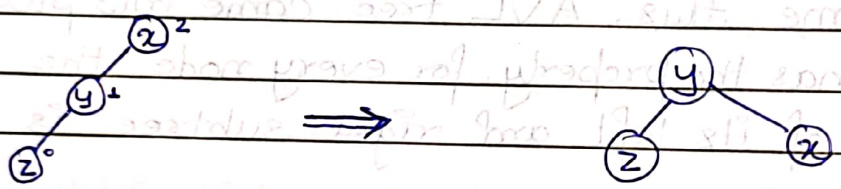
* Searching

AVL search is same algo. as BST search. Due to

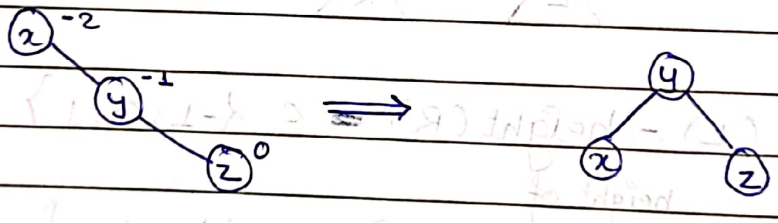
AVL tree's balance property, the worst case complexity becomes O(log n)

* Insertion

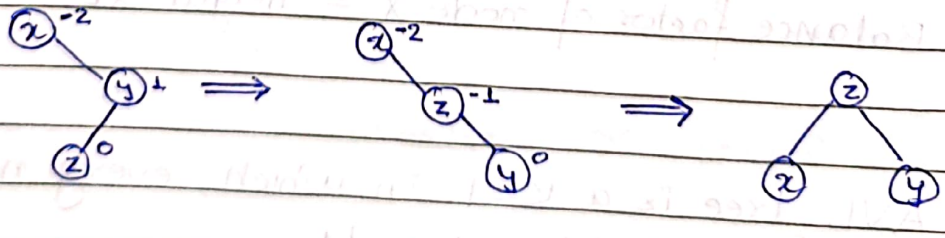
(1) Left - Left Rotation



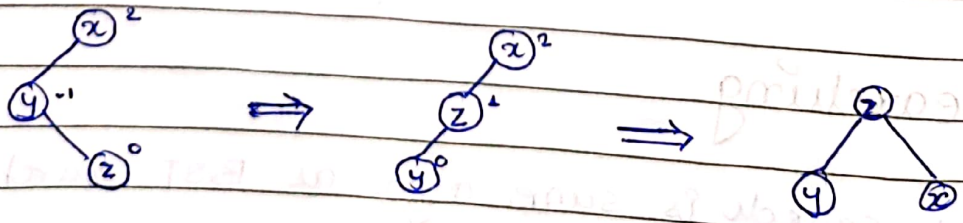
(2) Right - Right Rotation



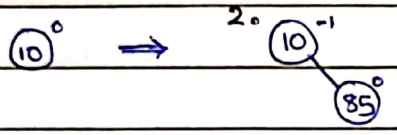
(3) Left - Right Rotation

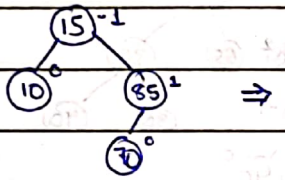


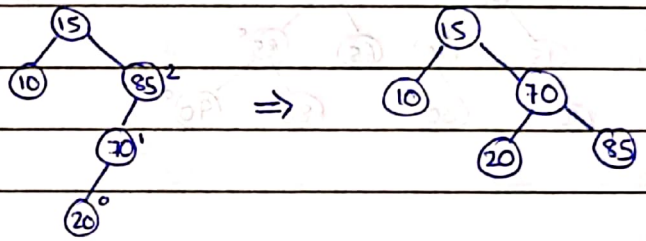
(4) Right - Left Rotation

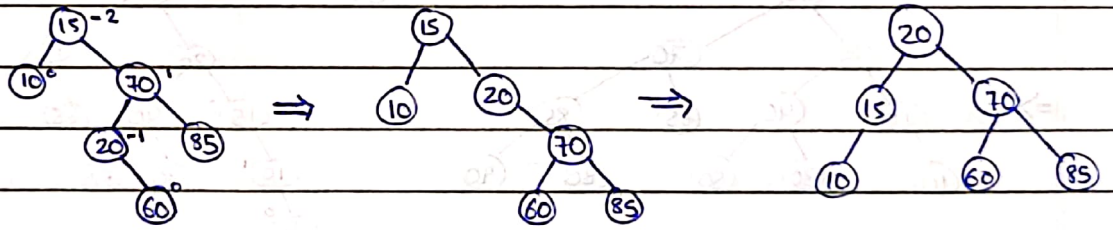


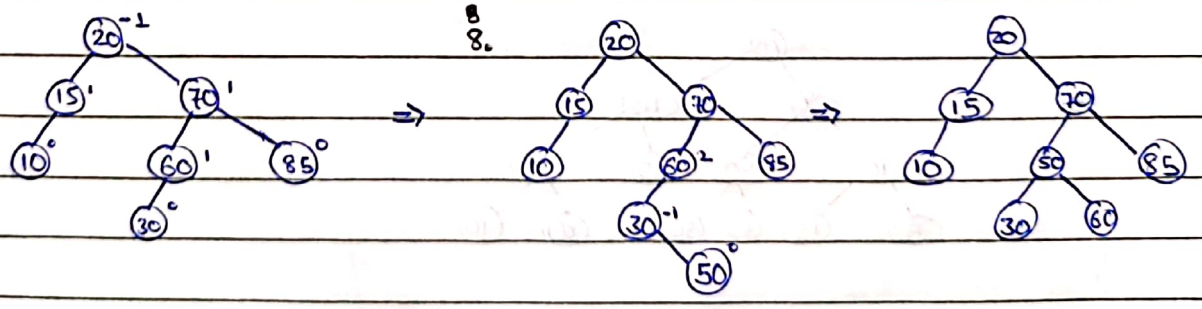
Ex. Insert the following in an AVL Tree: 10, 85, 15, 70, 20, 60, 30, 50, 65, 80, 90, 40, 5, 55

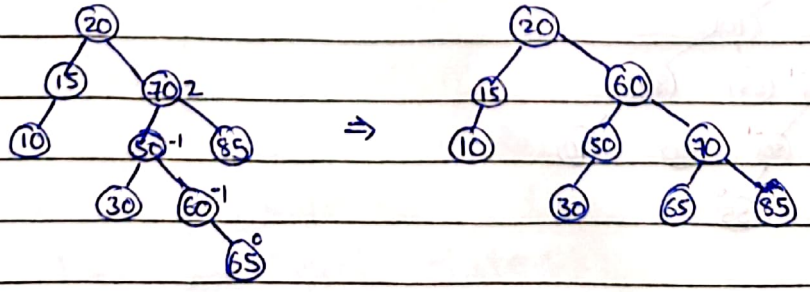
1. 

4. 

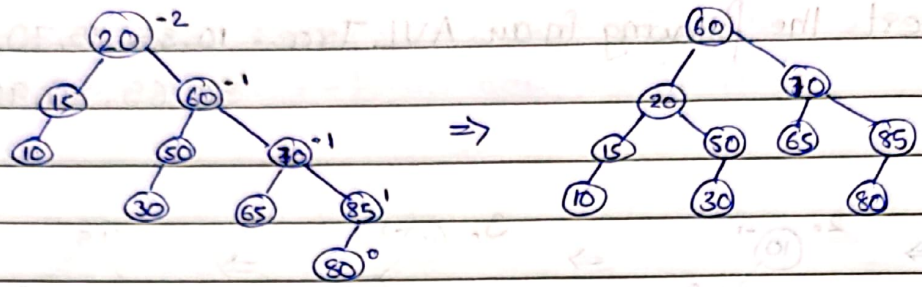
5. 

6. 

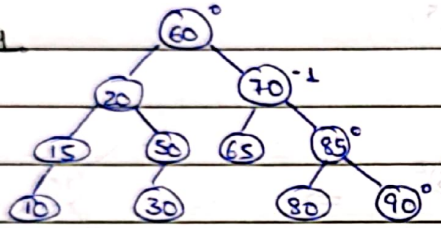
7. 

9. 

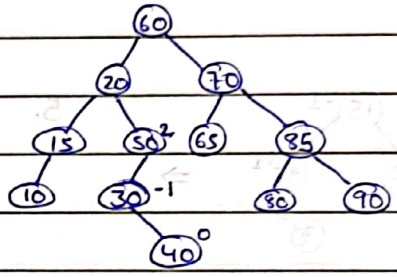
10.

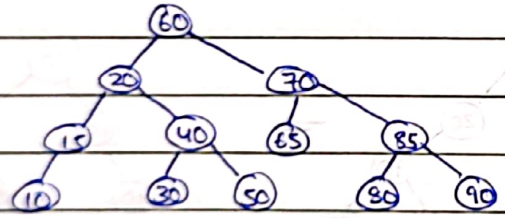

20 -2
15
60 -1
10
50
30 -1
30
65
85 1
80 0
60
20
70
15
50
65
85
10
30
80

11.

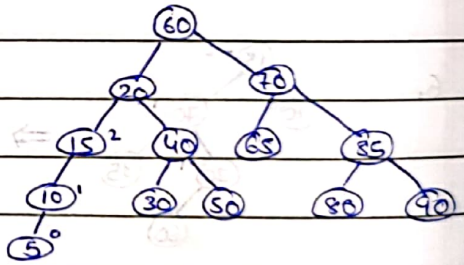

60 0
20
70 -1
15
50
65
85 0
10
30
80
90

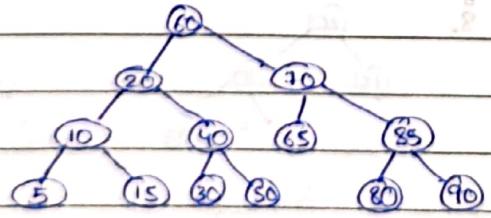
12.


60
20
70
15
50 2
65
85
10
30 -1
80
90
40 0

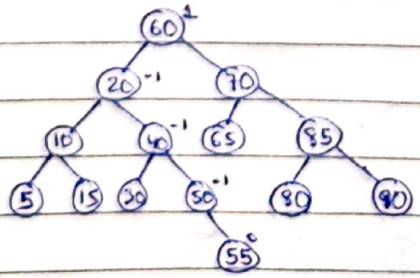

60
20
70
15
40
65
85
10
30
50
80
90

13.


60
20
70
15 2
40
65
85
10 1
30
50
80
90
5 0


60
20
70
10
40
65
85
5
15
30
50
80
90

14.


60 2
20 -1
70
10
40 -1
65
85
5
15
30
50 -1
80
90
55 0

- In case of a single insertion only one rotation is needed for first encountered imbalance, i.e. O(1) rotations.

- The time complexity of insertion in AVL tree is a culmination of:

⇒ Searching pos. for new node +	O(logn)
insertion +	O(1)
single rotation {at max}	O(1)

∴ Overall, insertion requires O(logn) operations.

* Deletion

AVL delelion is: BST delete + Balancing.

Traverse back from the deleted node, checking balance factor of each node on the way. Stop at the first imbalance if found.

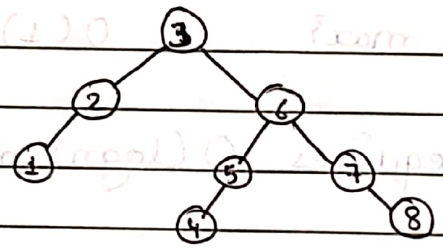
Say node X. If the deleted node was on X's left then check balance factor of X→right.

bf(x→right)	Operation
1	Left - Right Rotation(x) {leftrot(x→right); rightrot(x);}
0	Right Rotate (x)
-1	Right Rotate (x).

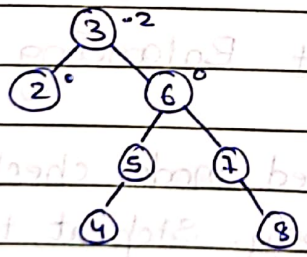
If deleted node was on right of X.

bf(X→left)	Operation
1	left rotate (X)
0	left rotate (X)
-1	right rotate (X→left); left rotate (X);

Ex.

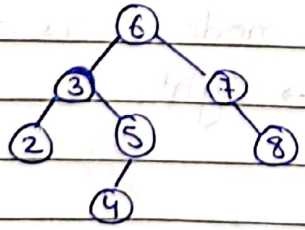


(1) Delete (1)

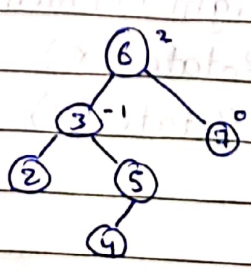


Right 0

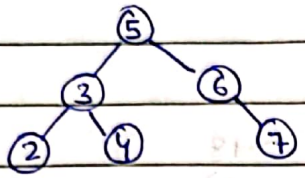
=>



(2) Delete (8)

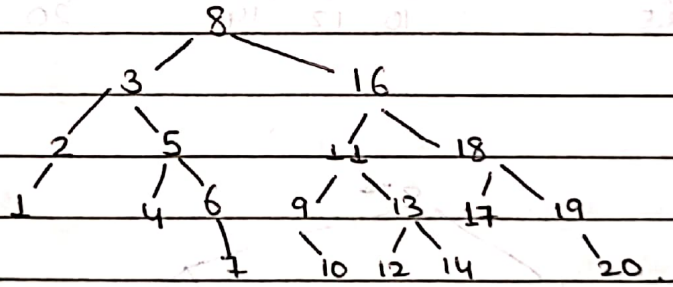


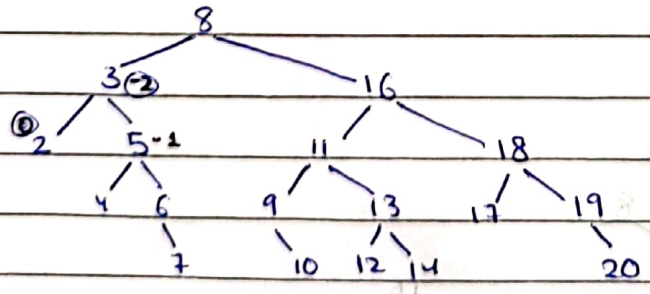
Left -1



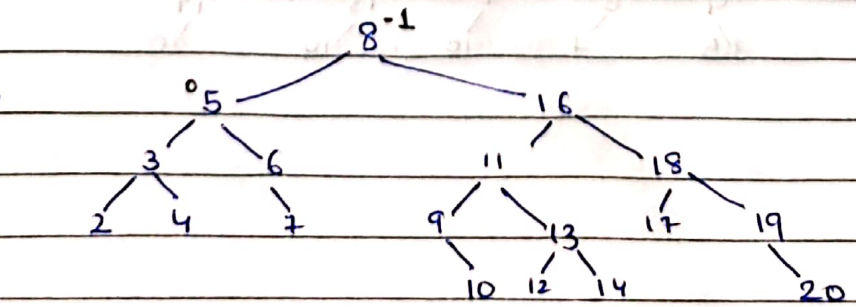
- In case of deletion a single rotation may not be enough. In the worst case every level might need rotation, i.e. O(log n) rotations.

Ex. Delete 1 from



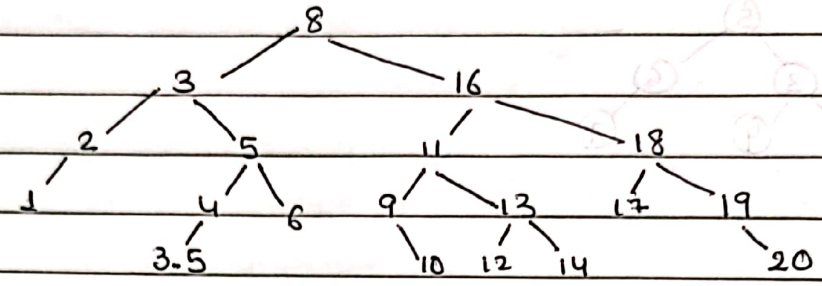


Right, -1

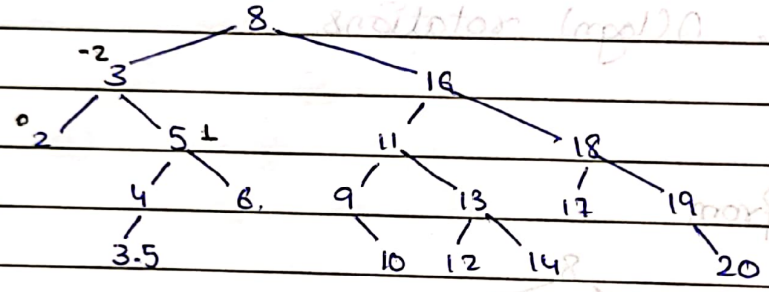


8 is balanced.

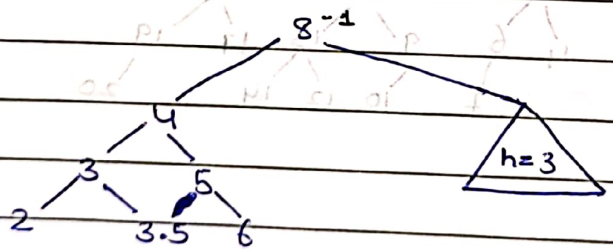
Ex.



Delete (1)



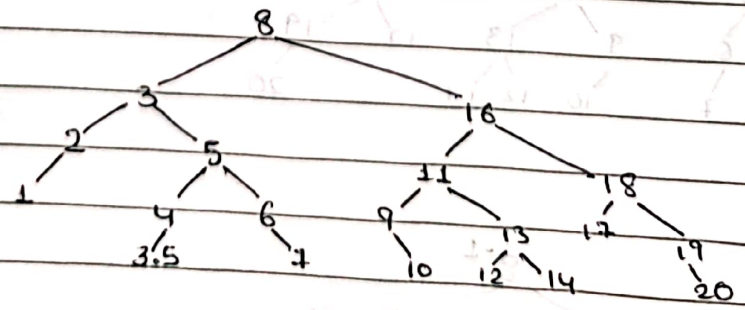
Right, 1



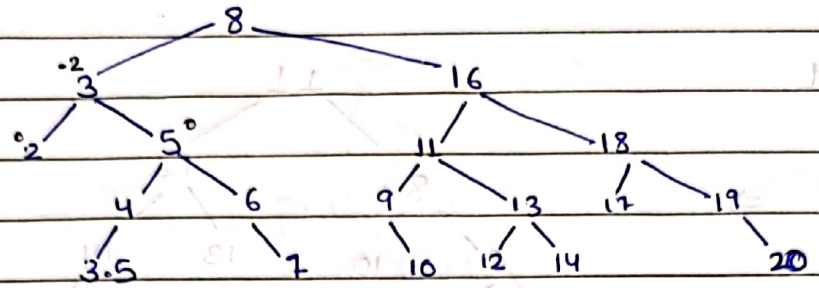
⇒

8 is balanced

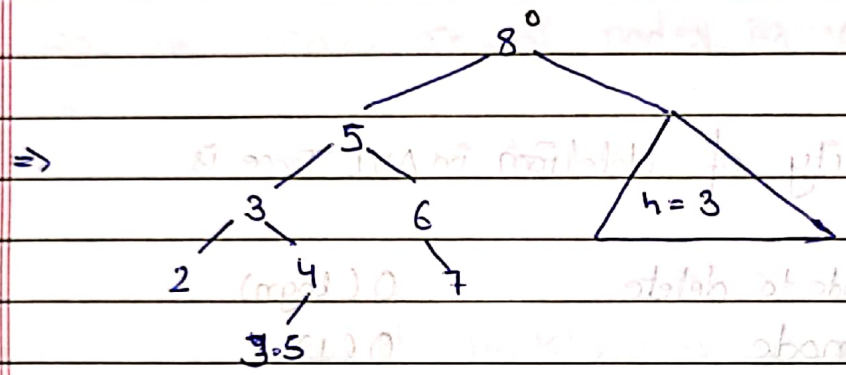
Ex.



Delete 1.

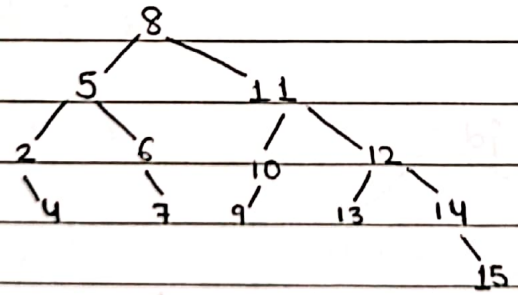


Right, 0

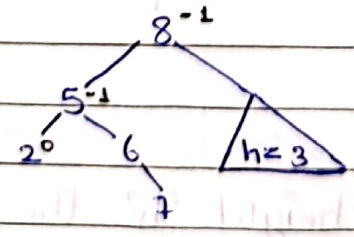


8 is balanced.

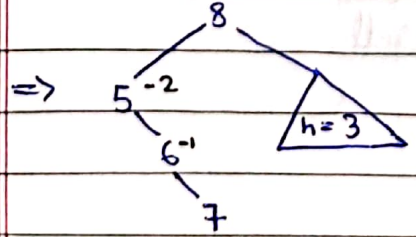
Ex.



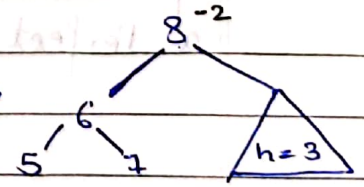
Delete 4, then 2.



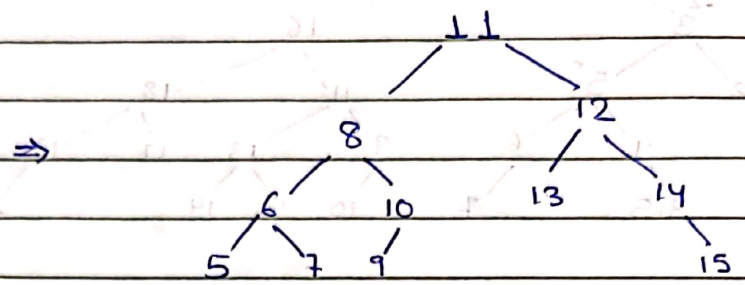
No imbalance.



Right, -1



Right, -1



- Time complexity of deletion in AVL Tree is

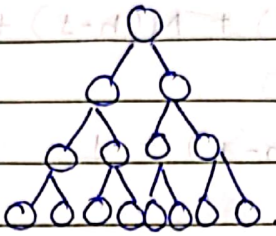
⇒ Search node to delete	O(logn)
+ delete node	O(1)
+ h x rotate	h x O(1) = O(logn)

∴ O(logn)

* Minimum and Maximum Nos. of Nodes in AVL Tree of Height 'h':

- Maximum: Given the height 'h' the max. no. of nodes in AVL tree will be same as no. of nodes in a Perfect tree of height 'h', since a Perfect BT is AVL tree as well.

Ex. $h = 3$

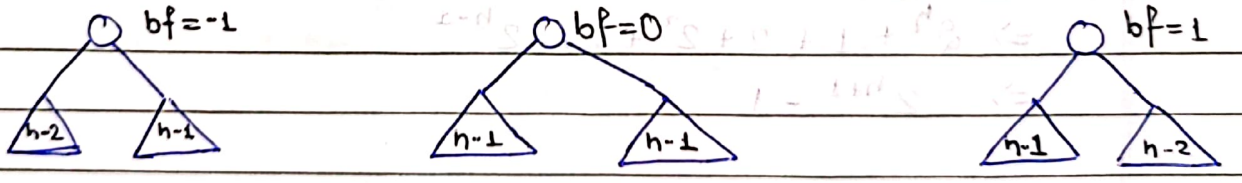


So, the max. no. of nodes in AVL Tree of height 'h' is

$$(2^{h+1} - 1)$$

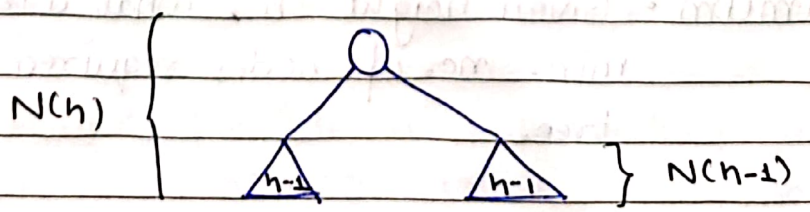
- Another way of thinking about max. no. of nodes is using balance factor.

Say height is h, then for the root node, there are three options:



Option with bf = 0 gives max. no. of nodes out of the three, since each side is of height h-1.

N(h): max. no. of nodes in AVL tree with height h.



So,

$$N(h) = N(h-1) + N(h-1) + 1$$

$$\Rightarrow N(h) = 2N(h-1) + 1$$

base case: $N(0) = 1$

So, solving for $N(h)$

$$\Rightarrow N(h) = 2N(h-1) + 1$$
$$\Rightarrow 2[2N(h-2) + 1] + 1$$
$$\Rightarrow 2^2 N(h-2) + 1 + 2$$
$$\Rightarrow 2^2 [2N(h-3) + 1] + 1 + 2$$
$$\Rightarrow 2^3 N(h-3) + 1 + 2 + 2^2$$
$$\vdots$$
$$\Rightarrow 2^k N(h-k) + 1 + 2 + 2^2 + \ldots 2^{k-1}$$

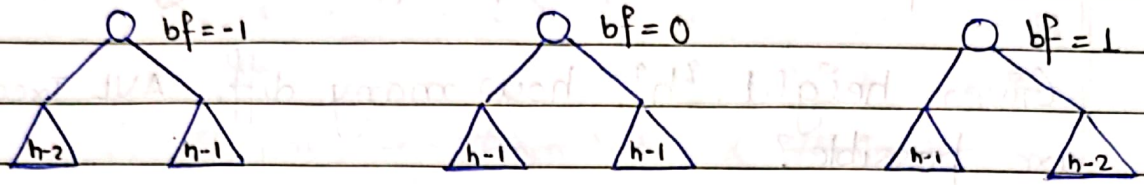
$k = h$

$$\Rightarrow 2^h + 1 + 2 + 2^2 + \ldots 2^{h-1}$$
$$\Rightarrow 2^{h+1} - 1$$

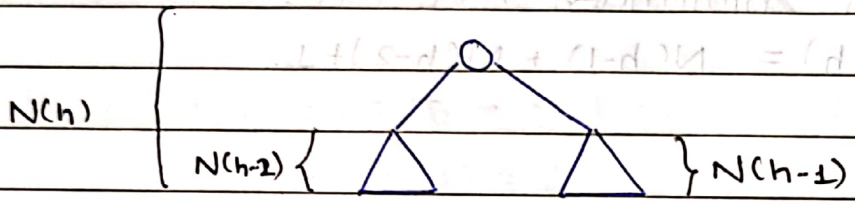
Note: Another way of framing the question is: given a fixed no. of nodes, what will be the min. height.

- Minimum: Given height 'h', what will be the min. no. of nodes required in AVL tree.

So, again for height 'h', there are three options:



So, option 1 or 3 would give min. no. of possible nodes.



N(h): Min. no. of nodes in AVL tree of height h.

$$N(h) = N(h-2) + N(h-1) + 1$$

(The closed form formula is not easy to solve)

Base Cases: $N(0) = 1$

$N(1) = 2$

$$\therefore N(h) = \begin{cases} 1, & h = 0 \\ 2, & h = 1 \\ N(h-1) + N(h-2) + 1, & h \geq 2 \end{cases}$$

Note: Another framing of the question: Given no. of nodes, max. possible height of AVL trees.

Ex. Max. height possible with 14 nodes?

$N(4) = 12$ & $N(5) = 20$. 14 is < 20, so height won't get to 5. $\therefore$ Max height = 4.

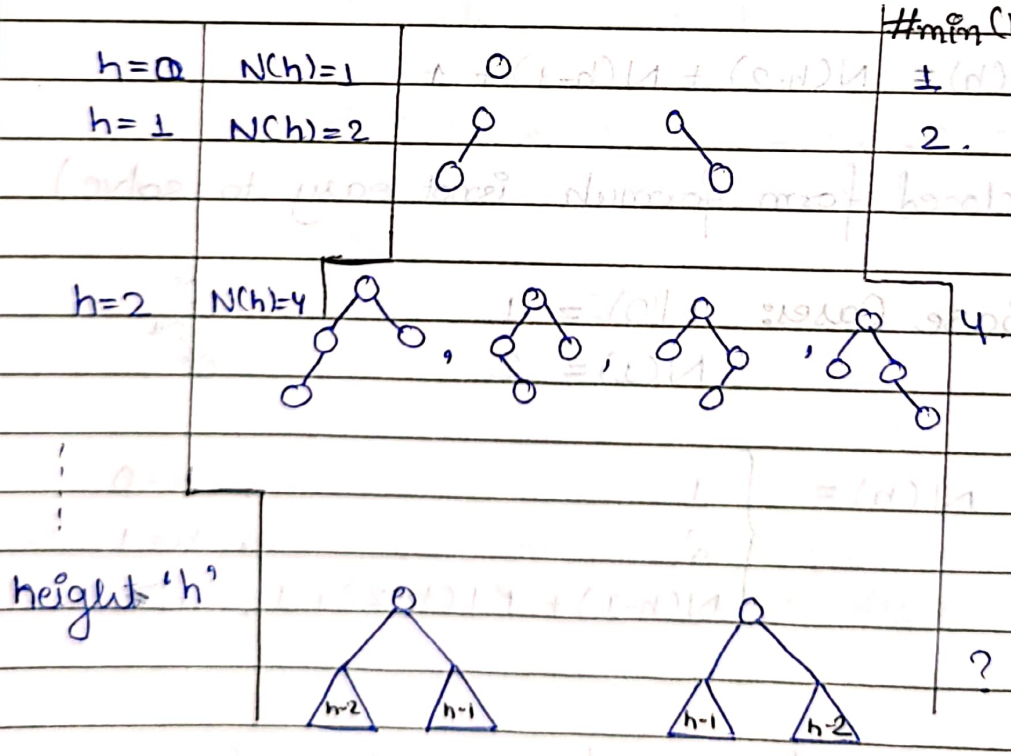
* No. of AVL Trees for given Height

Given height 'h', how many diff. AVL Trees are possible?

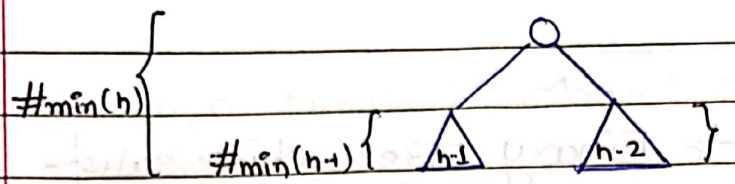
We already know, given height 'h', the no. of nodes in minimal AVL tree.

$$N(h) = N(h-1) + N(h-2) + 1.$$

#min(h) : No. of minimal AVL tree for height h.



So, for h, the no. of minimal AVL trees are count of above structures $S_1 + S_2$. Since S_1 is mirror image of S_2, we can just say $2 \times S_1$.



$\therefore$ $\#min(h) = \#min(h-1) \times \#min(h-2)$

This is just for S_1.

$\therefore$ $\#min(h) = 2 \times \#min(h-1) \times \#min(h-2)$

Base Case: $\#min(0) = 1$

$\#min(1) = 2$.

$\therefore$

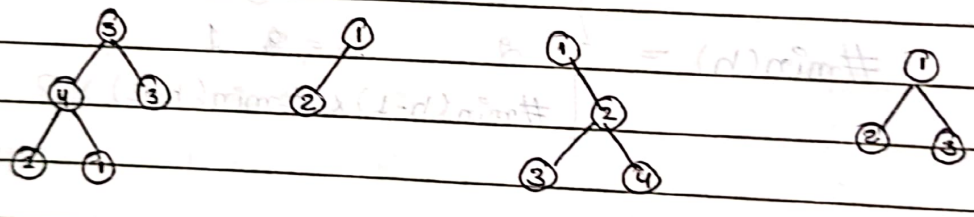
$$\#min(h) = \begin{cases} 1, & h = 0 \\ 2, & h = 1 \\ \#min(h-1) \times \#min(h-2) \times 2, & h \geq 2 \end{cases}$$

HEAPS

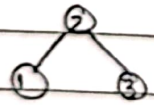
Heap is a Complete Binary Tree which satisfies the Heap Property.

- Min-Heap property: Every node in the heap should have value less than their children's.
- Max Heap Property: Every node in the heap should have value & larger than their children's.

Eg.



Max Heap Min Heap Not CBT Min Heap.



No. Doesn't satisfy either heap property.

* Since Heap is a CBT, it can be stored in an array easily.

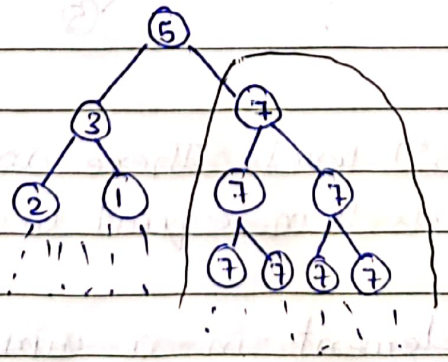
* Given a Min Heap where would the min & max element exist ?

Since we have a min-heap, intuitively the min. element would be in the root.

The max element de facto can't be ~~higher~~ lower in value than any other node, and min-heap requires every node to have value lower than it's children.

Therefore max. element would be present in some leaf node.

If there are duplicate elements in heap and max element is present in non-leaf node, then all the nodes in its subtree must be containing max value only since there is no element greater than it to be its children.



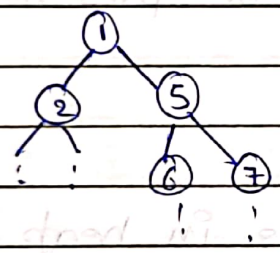
Subtree contains all nodes with max value.

* K^{th} Minimum in MinHeap

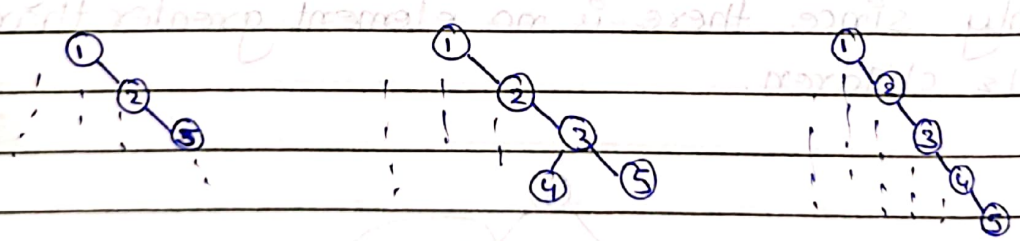
- K^{th} min. is the value such that there are K elements less than or equal to K.

Ex. Say elements in Heap are: 1, 2, 3, 4, 5, ...100

5 can't be on 0th level, since it's not min.
5 can be on 1st level.



Similarly it can be on 2nd level, 3rd level and 4th level.

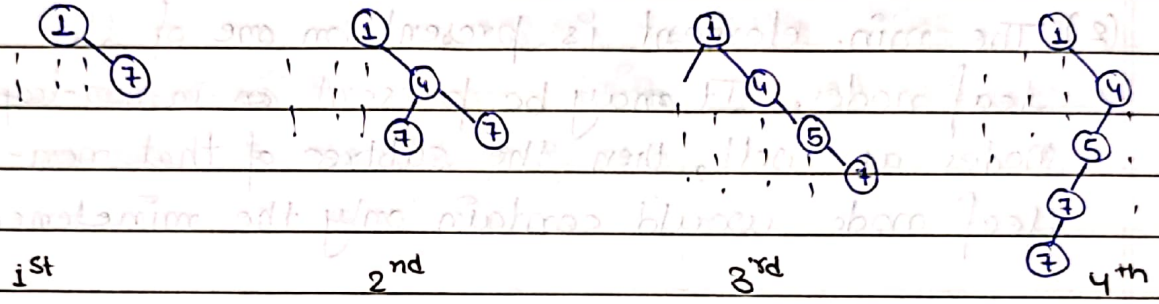


5 can't be on 5th level. There aren't sufficient no. of values less than equal to 5.

So, K^{th} min element in a min-heap can't be on higher than $(K-1)^{th}$ level.

Ex. Min Heap with: 1, 4, 5, 7, 7, 9, 10, ...
On which levels can 7 exist?

7 is both 4th and 5th minimum value, So, 7 can be on ~~2nd, 3rd or 4th or 5th~~ 1st, 2nd, 3rd or 4th level.



- In a min-heap K^{th} minimum element is present on level l, such that

$$1 \leq l \leq K-1$$

or

$$0 < l < K.$$

* • In min-heap:

① Least element is always on the root. Incase of duplicate, it can be on other nodes along with root.

② One of the leaf nodes contains max element. Max element can be in internal nodes as well in case of duplicate

③ K^{th} min. value is on level, l, s.t., $0 < l < K$. $(K > 1)$

• In max heap :

(1) The root node contains max element. Max element maybe present in other nodes as well along with root.

(2) The min. element is present on one of the leaf nodes. It may be present in non-leaf nodes as well, then the subtree of that non-leaf node would contain only the min element.

(3) k^{th} largest element is present on level l, s.t. $0 \leq l < k$. In case of duplicates, other levels are also possible along with l. $(k > 1)$

Ex. In a min-heap with n elements with the smallest element at root, the 7^{th} smallest element can be found in :

(A) $\Theta(n \log n)$
(B) $\Theta(n)$
(C) $\Theta(\log n)$
✓(D) $\Theta(1)$

Since 7^{th} min. will be on level l b/w 1 & 6, there are a total of ~~$2^{6+1} = 128$~~ nodes to search.

$$\underbrace{(2^{6+1} - 1)}_{\text{(0 to 6th level)}} - \underbrace{1}_{\text{(0th level)}} = 126.$$

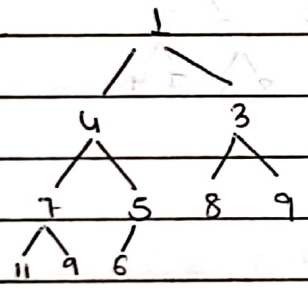
$\therefore \Theta(1)$.

* Insertion in Heap

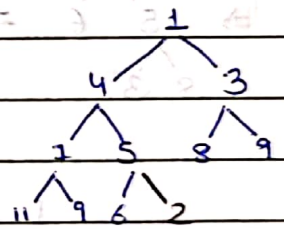
Insertion in Heap takes part in two stages:

(1) Add a new node according to CBT property & set its ~~data~~ value.

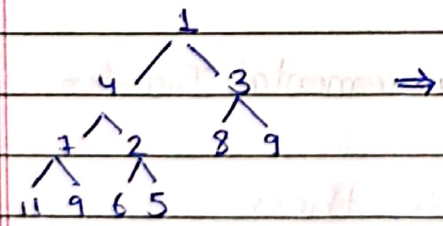
(2) Compare the node with ancestors recursively to ensure Heap property.

Ex 

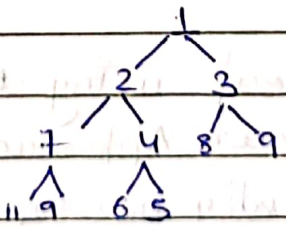
Insert 2.

(1) 

(2) $2 < 5$



$\Rightarrow$ $2 < 4$

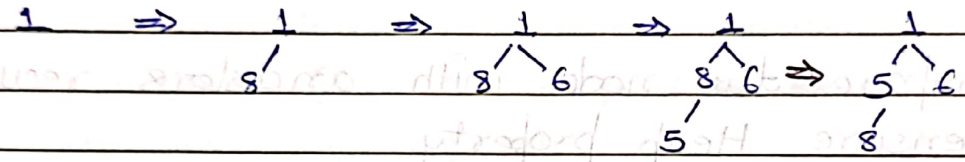


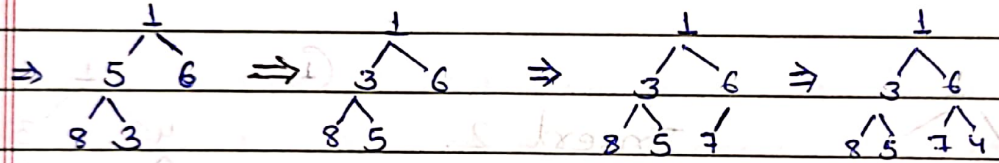
* T.C. of Insertion

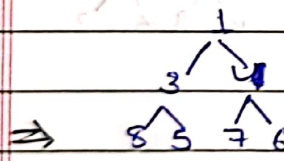
In worst case, the newly inserted leaf node might get compared up till the root node. That's

why. T.C: O(logn)

Ex. Build MinHeap from : 1,8,6,5,3,7,4







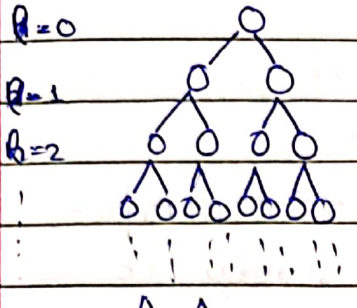
* T.C. of Build Heap using Incremental Insertion
(Top Down Approach)

If Heap currently has n nodes then,
no. of leaf nodes $\simeq \frac{n}{2}$

Each leaf insertion can take O(logn).

$\therefore \frac{n}{2} \times \log n = O(n \log n)$

max.



for level l, ↑ no. of nodes in that level
$\Rightarrow 2^l$

single insertion at level l takes = l max. comparisions.

	no. of nodes	no. of comparisions	
∴ l=0 :	$2^0=1$	0	= 0.
l=1 :	$2^1=2$	1	= 2×1
l=2 :	$2^2=4$	2	= 4×2.
⋮	⋮	⋮	⋮
l=h :	2^h	h	= $2^h \times h$.

$\therefore$ T.C = $0 + 2^1\times1 + 2^2\times2 + 2^3\times3 + \dots + 2^h\times h = S$.

$2S = 0 + 2^2\times1 + 2^3\times2 + 2^4\times3 + \dots + 2^h\times(h-1) + 2^{h+1}\times h$

$S-2S = 2^1 + 2^2 + 2^3 + 2^4 + \dots 2^h - 2^{h+1}\times h$

$\Rightarrow -S = (2^{h+1}-1) - 1 - 2^{h+1}\times h$.

$\Rightarrow S = 2^{h+1}\times(h-1) + 2$.

$= 2\times 2^h\times(h-1) + 2$.

$h = \log n$

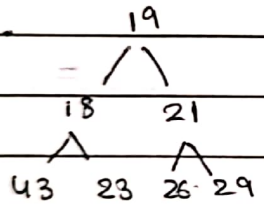
$\Rightarrow S = 2\times n\times(\log n - 1) + 2 = 2n\log n$.

$\therefore$ T.C. = $O(n\log n)$

* Heapify

Heapify method takes a node, and compares it with its descendents recursively to satisfy Heap property.

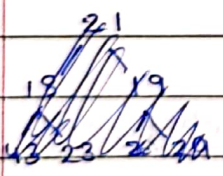
- Heapify method can be applied on a node X, only if both its right and left subtrees are heap.

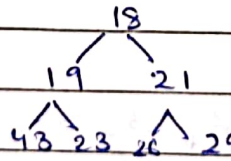
Ex. 

MinHeap.

Heapify (19)

18 < 19 < 21.





19 < 23 < 43.

∴ No need to apply again.

- Heapify in worst case will take root node and go all the way till leaf.

∴ TC = O(logn)

* Build Heap. (Bottom Up)

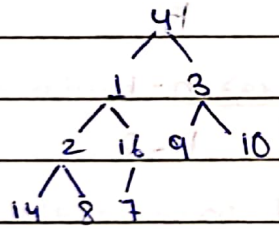
Build Heap takes bottom up approach. Every leaf node is already Heap (∵ single node).

(1) Create CBT structure from given elements blindly.

(2) Starting from last non-leaf node upto the root run Heapify method on the node.

Ex. Create min-heap from: 4, 1, 3, 2, 16, 9, 10, 14, 8, 7.

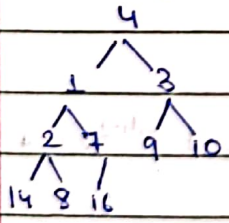
(1)



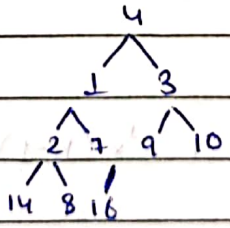
last non leaf : 16.

(2)

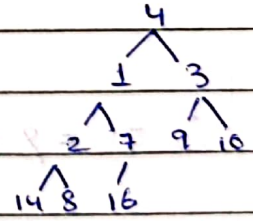
heapify (16)



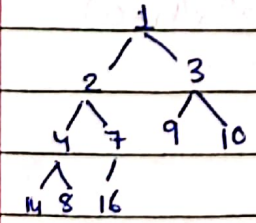
heapify (2)



heapify (3). heapify(1)

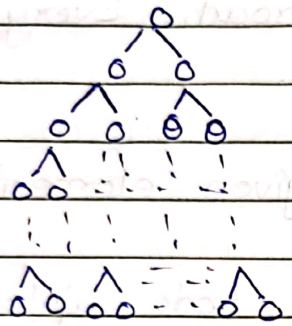


heapify (4)



This is a Heap now.
Every node satisfies min Heap property.

* Time Complexity



l	h
0	h
1	h-1
2	h-2
3	h-3
⋮	⋮
h	0

Lets say we apply Heapify to every single node.

	no. of nodes	no. of operations
l=0, h=h :	2^0	h
l=1, h=h-1 :	2^1	h-1
l=2, h=h-2 :	2^2	h-2
⋮		
l=h, h=0 :	2^h	0.

$\therefore$ TC = $S = 2^0 \times (h-0) + 2^1 \times (h-1) + 2^2 \times (h-2) + \dots + 2^{h-1} \times 1 + 2^h \times 0$

$2S = 2^1 \times h + 2^2 \times (h-1) + \dots + 2^{h-1} \times 2 + 2^h \times 1 + 2^{h+1} \times 0$

So, $2S - S = (2^1 + 2^2 + 2^3 + \dots 2^{h-1} + 2^h) - 2^0 \times (h-0)$

$\Rightarrow S = (2^{h+1} - 1) - 1 - h$

$= 2 \cdot 2^h - 2 - h.$

$\{ h = \log n \}$

$\Rightarrow \quad \$ = 2 \times 2^{\log n} - 2 - \log n$

$\Rightarrow 2 \times n - \log n - 2.$

$\therefore \quad T.C = O(n)$

* MinHeap : FindMin , DeleteMin

In a MinHeap the min. element is always at the root.

- Therefore, FindMin will just give value of root node.
- DeleteMin takes 3 steps :

(1) Delete Root Node.
(2) Delete last CBT node and put at root position.
(3) Heapify root.

- FindMin , $T.C = O(1)$
  DeleteMin , $T.C = O(\log n)$

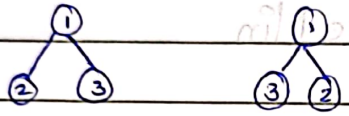
* No. of MinHeaps

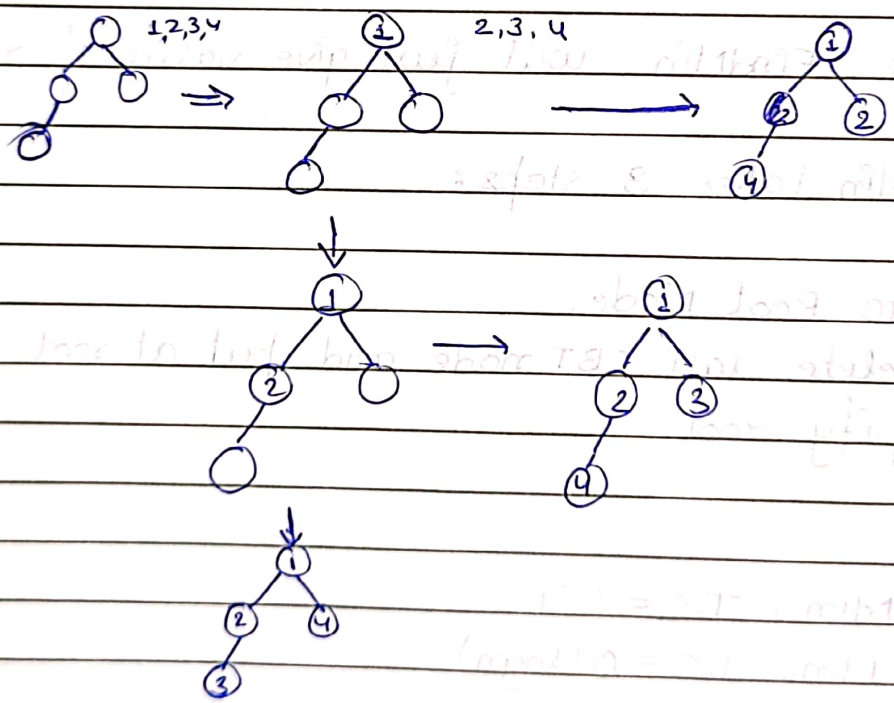
- Given 'n' no. of nodes, how many MinHeaps are possible.

- Say n is

1: 

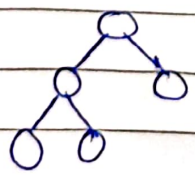
2: 

3: 

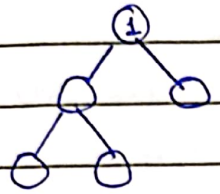
4: 

∴ 3

Ex. n = 5, 1, 2, 3, 4, 5

Structure: 

Only one of the five no. can be at root, the min. of them.



Both right and left subtrees have to be heap themselves.

Out of the remaining 4, four values, once has to go to right subtree, rest three will go to left.

No. of such divisions: ${}^4C_3$.

Say,

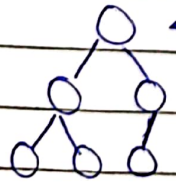
Using the selected 3 nodes left subtree creates L heaps, and right subtree creates R nodes.

$\therefore$ No. of heaps $= {}^4C_3 \times L \times R$.

Gen.

~~$= {}^{n-1}C_{n-2} \times$~~

Ex. $n=6$. 1, 2, 3, 4, 5, 6.



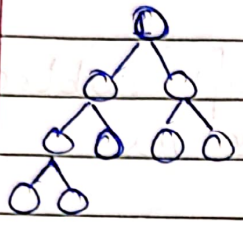
${}^5C_3 \times L \times R$

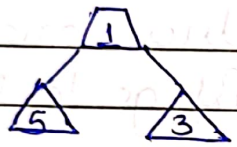
${}^2C_1 \times L \times R$ ${}^1C_1 \times L \times R$

1 1 1 1

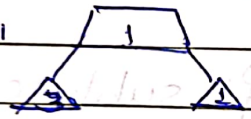
$\Rightarrow {}^5C_3 \times {}^2C_1 \times 1 = 10 \times 2 = 20$.

Ex. $n = 9$.

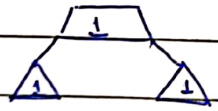


(1) 

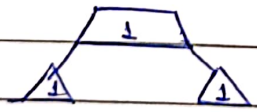
$1 \times {}^{8}C_3 \times L_1 \times R_1$

(2) L_1 

$1 \times {}^{4}C_3 \times L_2 \times R_2$

(3) R_1 

$1 \times {}^{2}C_1 \times 1 \times 1$.

(4) L_2 

$1 \times {}^{2}C_1 \times 1 \times 1$.

(5) $R_2 = 1$.

$\therefore 1 \times {}^{8}C_3 \times ({}^{4}C_3 \times 2 \times 1) \times (1 \times 2)$

$\Rightarrow \frac{6 \times 7 \times 8}{3! \, 3!} \times \frac{4!}{3!} \times 4$

$\Rightarrow 896.$

$\therefore \quad H(n) = {}^{n-1}C_K \times H(K) \times H(n-1-K)$

where K is the no. of nodes in left subtree.

Overall,

$$H(n) = \begin{cases} 1, & n = 1 \\ 1, & n = 2. \\ {}^{n-1}C_K \times H(K) \times H(n-1-K) \end{cases}$$

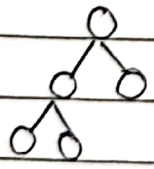
* Probability that given structure is Heap:

if not CBT structure : 0.

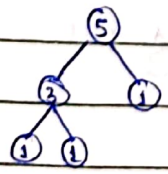
else

$$\frac{\#\text{of heaps possible with } n \text{ nodes}}{n!}$$

Ex.



$n = 5$ $\qquad \frac{H(5)}{5!}$

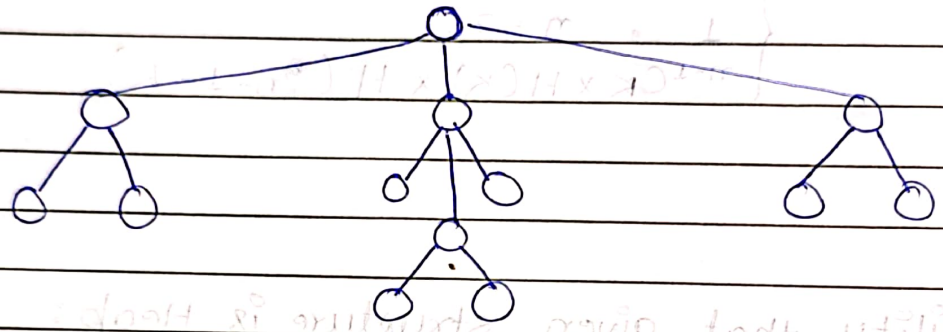


$$\Rightarrow H(5) = \frac{5!}{5 \times 3 \times 1 \times 1 \times 1}$$

Shortcut
node value = # nodes in its subtree.

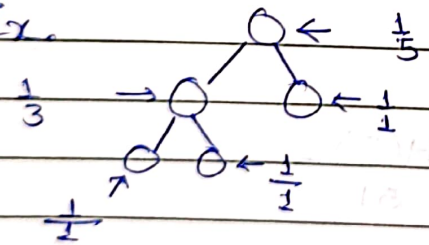
$$\therefore \quad P(\text{Heap}) = \frac{\frac{5!}{5 \times 3}}{5!} = \frac{1}{15}$$

ex. Consider the following tree. What is the prob. that every node receives the min. value in its subtree from $\{1, 2, \ldots, 13\}$.



Prob. of a node rec. min value in its subtree with n nodes $= \frac{1}{n}$.

Ex.



$\therefore$ Prob for given tree $= \frac{1}{13} \times \frac{1}{3} \times \frac{1}{6} \times \frac{1}{3} \times \frac{1}{3}$

$\Rightarrow \frac{1}{13} \times \frac{1}{6} \times \left(\frac{1}{3}\right)^3$

* Heap Complexities

(1) Insertion
- Best Case : $O(1)$
- Avg case : $O(1)$
- Worst case: $O(\log n)$

(2) Deletion
- Best Case: $O(1)$
- Avg Case: $O(\log n)$
- Worst Case: $O(\log n)$

(3) FindMin
- $O(1)$ in all cases.

* Heap Sort

```
new_arr = {};

make_heap(arr);                                   } O(n)
while ( arr is not empty):                        } O(n)
    new_arr.insert( find_min(arr));               } O(logn)
    deletemin(arr);
```

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

$\Rightarrow O(n \log n)$

* Priority Queue

Every element popped off the Queue should have highest priority of all current element.

Suppose we implement using a linear structure like array or linked list.

- insertion : O(1)
- deletion : O(n)

Suppose we use sorted linear structure.

- insertion : O(n)
- deletion : O(1)

Suppose we use Balanced BST

- insertion : O(logn)
- deletion : O(logn)

Say we use Binary Heap

- insertion : O(logn)
- deletion : O(logn)

- On average Heap has less no. of operations compared to Balanced BST.

HASHING

Hashing is a method, which uses Hash Table data structure, to performs searching in O(1). Hashing is suitable when the order between data is not important.

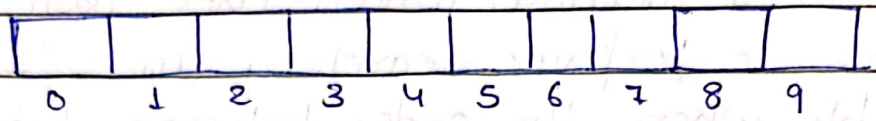
Hashing in average case can perform insertion, deletion & searching in O(1).

- Hash Table: The actual structure in which data is stored.
- Hash Function: A func. which maps a key to location in the hash table.
- Universe of Keys: All the possible keys.
  Ex. All 10 digit phone numbers.
  Ex. All the strings of all possible sizes.
- Actual Keys: The subset of keys which actually map to the Hash Table.
  May or may not be known in advance.

* Division Method.

$$hash(K) = K \% m$$

Ex. $m = 10$.



(1) $K = 5025$

$h(K) = 5025 \% 10$

$= 5$

(2) $K = 100289$

$h(K) = 100289 \% 10$

$= 9$.

* How to pick m

- Say we pick $m = 2^K$ for some K.

  Ex. $K = 2$. $m = 4$.

  $Key = (10101100101)_2$

  $h(key) = 01$

  This is not good. This depends on last K bits only.

- Similarly using $m = 10^K$ will focus on last K-digits in base-10 & ignores rest.

- Generally a prime number is used as m, since it doesn't have any cofactor with keys.

* A Good Hash Function

(1) Should consider every portion of the key.

(2) Should spread out the values in the hashtable efficiently.

(3) Hash function should be efficient to compute.

* Simple Uniform Hashing

It is when every element is equally likely to hash into any of the m slots, independently of where any other element has hashed to.

$$\text{Probability}(H(k) = i) = \frac{1}{m}, \forall k \forall i$$

* Collision

When more than one key maps to the same location in the hash table, it is known as Collision.

Closed Addressing/
Open Hashing
- Open Chaining.

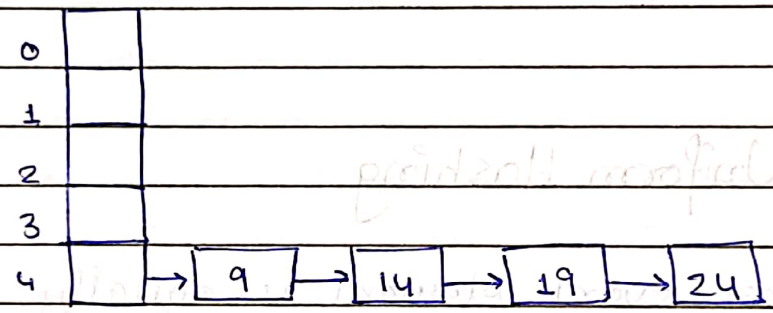
Open Addressing/
Closed Hashing
- Linear Probing
- Quadratic Probing
- Double Hashing.
- Random Hashing

* Open Chaining

If multiple keys map to the same loc. then append a chain to the location & add new element.

Ex. $h(K) = K \% 5$

Keys. 9, 14, 19, 24 map to same loc. 4.



- Searching in worst case can take $O(n)$, but average case performance is still $O(1)$.

- Load factor: no. of elements in the hash table by size of the table.

$$\alpha = \frac{n}{m}$$

- In case of chaining:
  - α is the avg. no. of elements stored in a chain.
  - α can be less than, equal to, greater than 1.

Average case analysis of searching in Open Chaining

(Assumption: Simple uniform hashing)

Theorem: In a hash table with chaining, an unsuccessful search takes average-case time $\Theta(1+\alpha)$, under the assumption of simple uniform hashing.

(α is load factor. $\alpha = \frac{n}{m}$)

It is intuitive, in case of chaining, α is the average length of the chain.

So, in case of unsuccessful search:

- Calculating $h(k)$: $\Theta(1)$
- Checking every element in chain: $\Theta(\alpha)$.

$\therefore \Theta(1+\alpha)$

- Say, $n >> m$, $\alpha \simeq n$, then $\simeq \Theta(1+n)$, so not constant.
- Say, $n \simeq m$, $\alpha \simeq 1$, then $\Theta(1+\alpha) \simeq \Theta(1)$, so constant. Therefore we prefer $m = \Theta(n)$

Theorem : In case of successful search, with chaining, it takes $\Theta(1+\alpha)$ under the assumption of simple uniform hashing.

(α is load factor. $\alpha = \frac{n}{m}$)

n: # of elements already inserted.
m: # of slots.

- inserting the i^{th} item:
  - unsuccessful search of i^{th} item.
  - insert the item

- We want to prove:

(x) time to successfully search i^{th} item

=

(y) time to ~~search~~ insert i^{th} item when there were just $(i-1)$ items.

- y takes $1 + \frac{i-1}{m}$ time.
- So, on avg, x should take:

$$\frac{1 + \sum_{i=1}^{n} \frac{i-1}{m}}{n} = \Theta(\alpha + 1)$$

* Probing

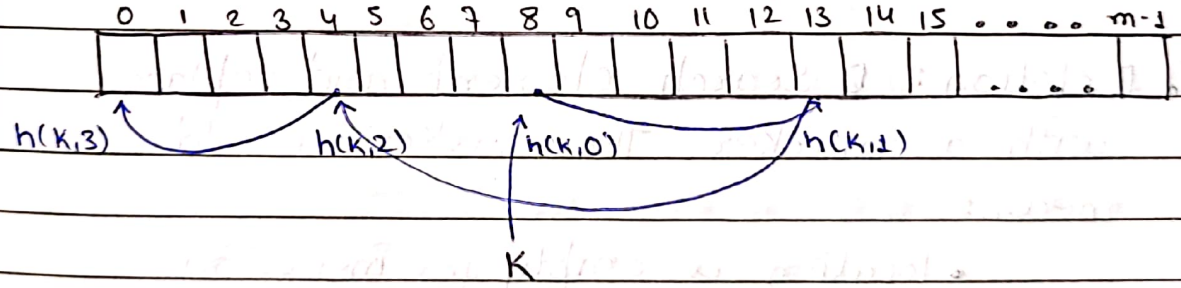
- There is no chain.
- All elements are stored in the hash table.

$h: U \times \{0,1,2,\ldots,m-1\} \longrightarrow \{0,1,2,\ldots,m-1\}$

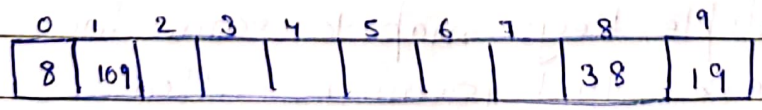
U is the universe of keys.

Ex. Given a key K, the probe sequence will be,

$h(K,0), h(K,1), h(K,2), \ldots, h(K,m-1)$



Ex. $h(K) = K \% 10$ $\quad H(K,i) = (h(K)+i) \% 10$



Say, K is 28. $\quad h(K) = 28 \% 10 = 8.$

$H(K,0) = 8$ $\quad H(K,4) = 2.$

$H(K,1) = 9$

$H(K,2) = 0$ $\quad$ loc. 2 is empty. So, K goes to loc. 2.

$H(K,3) = 1$

- In general,

$$H(K,i) = [h(K) + f(i)] \% m.$$

- Linear probing: $f(i) = i$
  Quadratic probing: $f(i) = i^2$
  Double hashing: $f(i)$ % second hashing fn, $f(x) \times i$

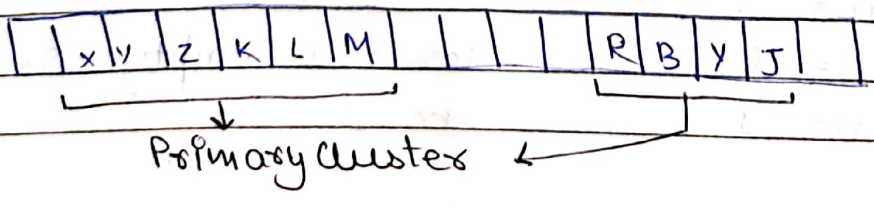
* Linear Probing

- Searching: Continue searching at successive locations, either key is found or empty location is found.

- Deletion: Search element and replace with a marker. The marker would mean:
  - location is empty for insertion.
  - while searching for a key, this location is not an empty location.

- When multiple keys map to same loc. small clusters start forming, called as primary clusters.

Linear probing suffers from primary clusters.



* Quadratic Probing

$$H(K,i) = h(K) + c_1 i + c_2 i^2$$

Generally we take $c_1 = 0$, $c_2 = 1$.

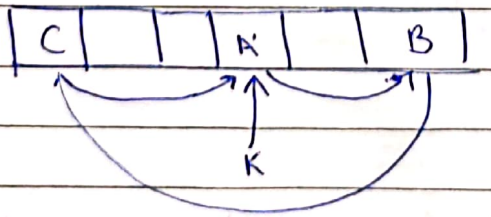
So,

$$H(K,i) = h(K) + i^2$$

- Quadratic Probing doesn't suffer from Primary Cluster problem.

It suffers from Secondary Clusters. When two keys map to the same location, then their probe sequence is same.

- In case of Quadratic Probing, even if they are empty locations they might not be visible because of cycle formation.



Above there are empty locations, K can't find them.

* Double Hashing

Uses two hash functions:

$$H(K,i) = h_1(K) + i \times h_2(K)$$

Now, since the f(i) depends on K itself, there won't form secondary clusters, since for two diff keys probe sequence would be different even if they map to same location initially.

* Number of possible probes

- Given a key K, what is the no. of linear probes possible?

  ~~Ex~~ ~~h(K), h(K)+1, h(K)+2, ...~~

  h(K), h(K)+1, h(K)+2, ...

  Ex. 2, 3, 4, ...
  5, 6, 7, 8, 9, 0, 1, ...
  etc.

  ${}^{m}C_1 = m$.
  {we can only choose starting idx}.

- Similarly for Quadratic probing, only starting index is of choice.
  m.

- For double hashing:

$h_1(K)$, $h_1(K) + h_2(K)$, $h_1 + 2h_2(K)$, $h_1 + 3h_2(K)$, ...

↓ m, ↓ m, ↓ 1, ↓ 1.

The first index is value of $h_1(K)$. Second depends on value of $h_2(K)$ as well. Once the value of $h_1(K)$ & $h_2(K)$ are fixed, rest of the probe becomes constant.

$\therefore m \times m = m^2$

* Analysis of Open Addressing (assuming Uniform Hashing)

- Since we assume Uniform Hashing, it means if we choose a location, and it is already filled, the next location is chosen with equal probability for all locations.

  So, no. of possible permutations = $m!$

- In Open addressing, $n \leq m$. So,

  $0 \leq \alpha \leq 1$. {α is load factor}

Theorem: Given an open-addressing hash table, with load factor $\alpha = n/m < 1$, the expected number of probes in an unsuccessful search is at most $1/(1-\alpha)$, assuming uniform hashing.

- Currently filled slot = n.
- Prob. of hitting filled slot = $\frac{n}{m} = \alpha$
- So, Prob. of hitting empty slot = $1 - \frac{n}{m} = 1 - \alpha$
- Expected no. of hits to get empty slot:

# of attempts	Probability
1	$1-\alpha$
2	$\alpha(1-\alpha)$
3	$\alpha^2(1-\alpha)$
⋮	⋮
m	$\alpha^{m-1}(1-\alpha)$

$\therefore$ Expected no. of hits $= 1\times(1-\alpha) + 2\alpha(1-\alpha) + 3\alpha^2(1-\alpha) + \dots m\alpha^{m-1}(1-\alpha)$

$\Rightarrow (1-\alpha)[1 + 2\alpha + 3\alpha^2 + \dots + m\alpha^{m-1}]$

$S = 1 + 2\alpha + 3\alpha^2 + \dots + m\alpha^{m-1}$

$\alpha S = 1\alpha + 2\alpha^2 + \dots + (m-1)\alpha^{m-1} + m\alpha^m$

$(1-\alpha)S = 1 + \alpha + \alpha^2 + \dots \alpha^{m-1} - m\alpha^m$

$= \frac{\alpha^m - 1}{(\alpha - 1)} - m\alpha^m$

$\{n \simeq \infty\}$

$$+S = \frac{+1}{(\alpha+1)^2}$$

$$\therefore E = (1-\alpha) * \frac{1}{(1-\alpha)^2} = \frac{1}{(1-\alpha)}$$

- The cost of successful search for open-address hash table is same as unsuccessful search.
  i.e. $1/(1-\alpha)$

ARRAYS

* Row Order & Column Order

In Row Major Order, each row is stored subsequently in linear memory.

Ex.

a	b	c
d	e	f
g	h	i

...|a|b|c|d|e|f|g|h|i|...

Row major order.

...|a|d|g|b|e|h|c|f|i|...

Column Major Order

In Column Major Order, each column is stored subsequently in linear memory.

* Address of element a[i][j]

{Base addr. = b, # of row = r, # of cols = c, element size = sz}

A[L_1...U_1][L_2...U_2]

Row Major Order:

$$addr(a[i][j]) = b + [(i - L_1) \times c + (j - L_2)] \times size$$

Column Major Order

$addr(a[i][j]) = b + [(j - L_2) \times \gamma + (i - L_1)] \times sz$

* Indexing n-dimensional Array

N_i = Size of i^{th} dimension.
d = No. of dimensions.
n_i = index of an element in dimension i

00	01	02	03
10	11	12	13
20	21	22	23

RMO ..|00|01|02|...

CMO ..|00|10|20|...

In RMO, last index is changing faster.
In CMO, first index is changing faster.

Ex. a[2][4][3]

a[0]

00	01	02
10	11	12
:	:	:

...|000|001|002|010|011|....

a[1]

The last index is changing the fastest. This is Row Major Order.

∴ a is a collection of 2 arrays of 4x3.

Now,

...	000	010	020	001	011	021	...

This isn't CMO since middle index changes fastest.

For the CMO, the array would be represented as:

00	01	02	03
10	11	12	13

a[ ][ ][0]

a[ ][ ][1]

a[ ][ ][2]

...	000	100	010	110	020	120	...

So there are 3 arrays of 2 x 4.

a[N_1][N_2][N_3]

RMO: N_1 arrays of $N_2 \times N_3$ dims.
CMO: N_3 arrays of $N_1 \times N_2$ dims.

* Indexing

a [N1][N2][N3].

RMO:

addr(a[n1][n2][n3]) = b + (n1 x N2 x N3 + n2 x N3 + n3) x Sz

CMO:

addr(a[n1][n2][n3]) = b + (n3 x N1 x N2 + n2 x N1 + n1) x Sz