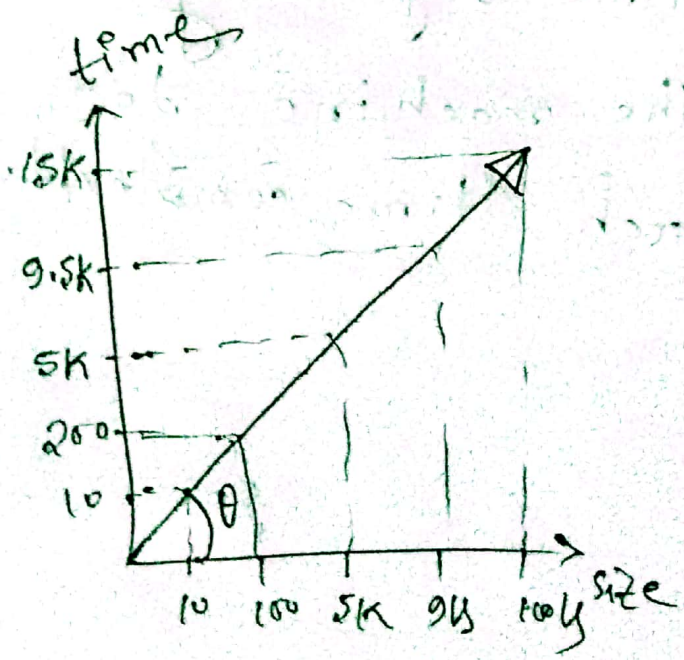
Time Complexity

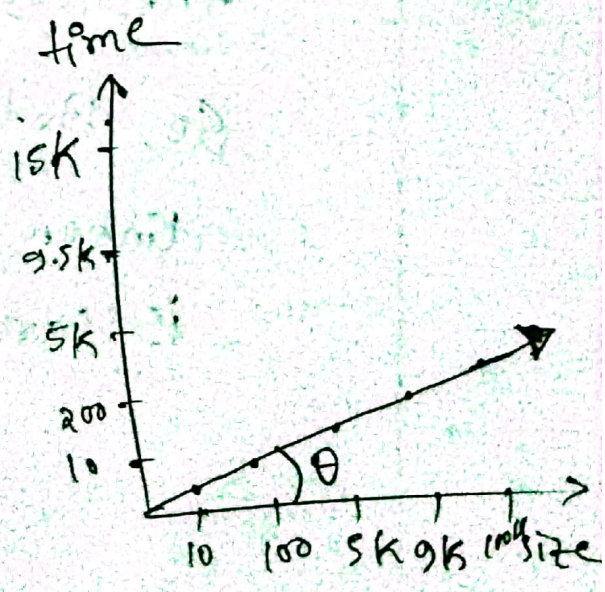
Time Complexity :

$\boxed{\text{Time complexity} \neq \text{Time taken}}$

Example : An old machine takes 10 sec to perform linear search.

A new machine takes 1 sec to perform linear search.





- Time complexity is graph.

<u>Time Complexity</u>: Time complexity is function that tells us how the time is going to grow as inputs grows.

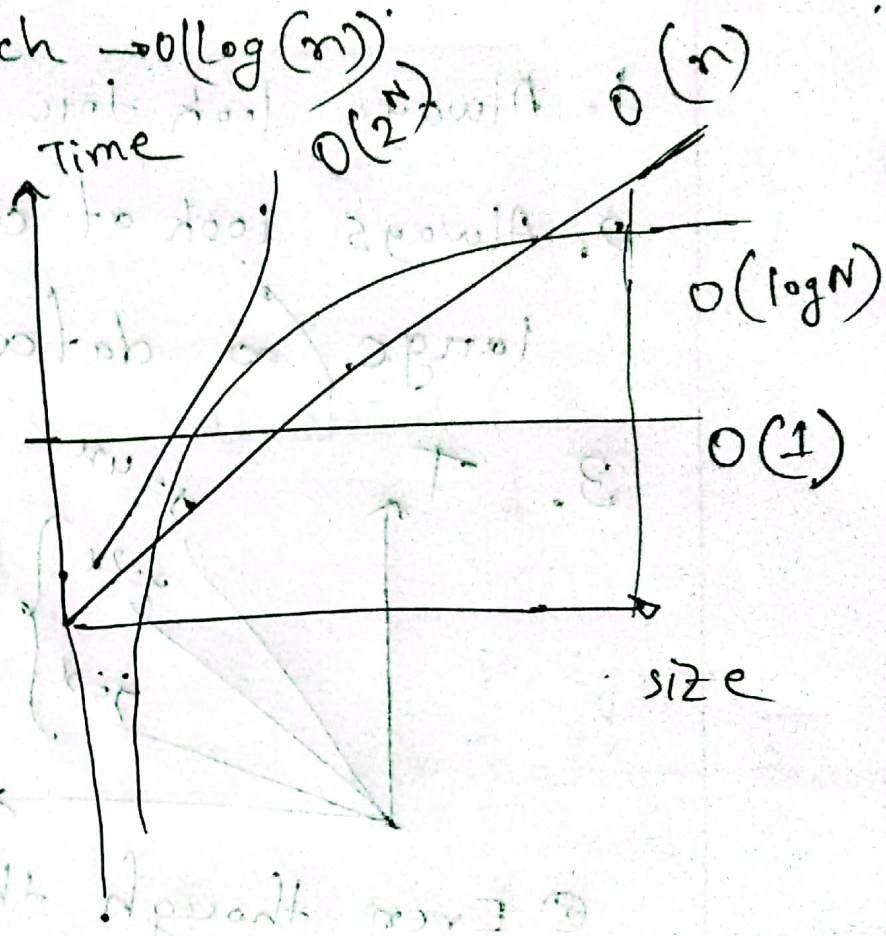
- In our example time is linearly growing as size growing. In Both the cases time is growing linearly. So, Both the machine for linear search time complexity is same.

CamScanner

Linear Search → $O(n)$

Binary Search → $O(\log(n))$

Comparison:

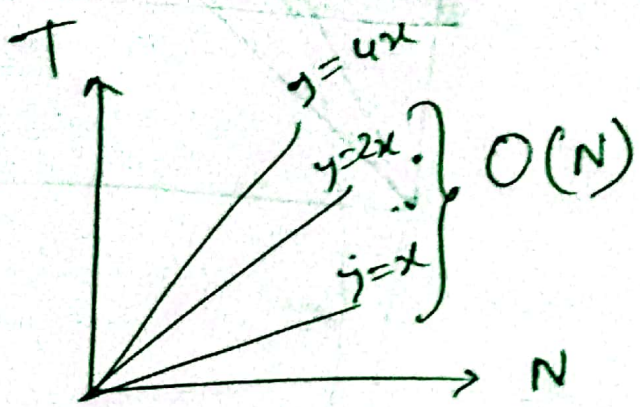


(*) For larger size linear search takes more time than binary search.

(*) N.B → we don't compare if for lower number

(*) $O(1) < O(\log N) < O(N) < O(2^N)$

Procedure for Analysing Complexity:

1. Always look for worst case complexity
2. Always look at complexity for large/∞ data
3. 

* Even though the value of actual time is different ~~by~~ but they are all growing linearly.
* We don't care about actual time.
* This is why we ignore all constants.

(4) $O(N^3 + \log N)$

Ex → N → 1 mil

$$\left((1\,mil)^3 + \log(1\,mil)\right)$$

$$= (1\,mil)^3 + 6$$

↓

very small

so, ignore

(*) Always ignore less dominating terms.

Ex: $O(3N^3 + 4N^2 + 6N + 9)$

$$= O(N^3)$$

CS CamScanner

Big-oh Notation:

Ex: $O(N^3)$. It's means that complexity can't be larger than N^3. It's the upperbound. It can be smaller or constant for particular case.

Mathmetical:

$$f(N) = O(g(N))$$

$$\boxed{\lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty}$$

Ex: $O(\underbrace{6N^3 + 5N^2 + 6N + 7}_{f(N)}) = O(\underbrace{N^3}_{g(N)})$

$$\lim_{N \to \infty} \frac{6N^3 + 5N^2 + 6N + 7}{N^3}$$

$$= \lim_{N \to \infty} \left(6 + \frac{5}{N} + \frac{6}{N^2} + \frac{7}{N^3}\right) = 6 < \infty$$

Big-omega Notation: Opposite of Big-oh

Ex: $\Omega(N^3)$, It's means that complexity can't be smaller than N^3. Complexity can be larger. It's the lower bound.

Mathmetics:

$$\boxed{\lim_{N \to \infty} \frac{f(N)}{g(N)} > 0}$$

CS CamScanner

Thefa - Notation: Combine both Big-oh and Big-omega

$\theta(N^2)$. It's means upperbound and lowerbound both is N^2

Mathmetics:

$$\boxed{0 < \lim_{N \to \infty} \frac{f(N)}{g(N)} < \infty}$$

CamScanner

Little o - Notation:

* This is also giving upperbound
* It's not strict upperbound

Big-oh	Little-oh
$f = O(g)$	$f = o(g)$
$f \leq g$	$f < g$, strictly (Stronger statement)

Maths:

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

$f(N) < g(N)$

Ex: $f(N) = N^2$, $g(N) = N^3$

$$\lim_{N \to \infty} \frac{N^2}{N^3} = \frac{1}{N} = 0$$

CS CamScanner

little Omega :

* This will b. give lowerbound
* It's not strictly lowerbound

$$f = \omega(g)$$

Maths:

Big Ω	little ω
$f = \Omega(g)$	$f = \omega(g)$
$f \geq g$	$f > g$

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

Ex: $f(N) = N^3$, $g(N) = N^2$

$$\lim_{N \to \infty} \frac{N^3}{N^2} = \infty$$

CamScanner

Space complexity :

Space Complexity of an algorithm is total space taken by the algorithm with respect to the input size. Space complexity includes both Auxiliary space and space used by input.

Auxiliary space: Auxiliary space is the extra space or temporary space used by an algorithm.

Linear Search, Binary search, Bubble sort, insertion sort, cyclic sort, selection sort all have space complexity $O(1)$.

Linear Search:

```
for ( i = 0 ; i < N ; i++ ) {
        //operation
}
```

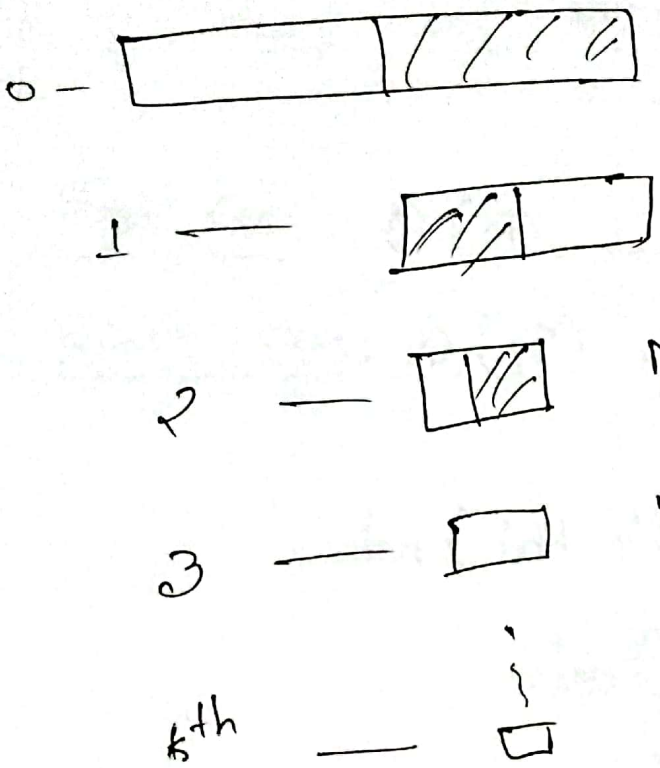
so, loops runing N times,

∴ worst case: Time complexity $O(N)$

Best case : Time complexity $O(1)$

CamScanner

Binary Search:



$N/2^0$

$N/2 = \frac{N}{2^1}$

$N/4 = \frac{N}{2^2}$

$N/8 = \frac{N}{2^3}$

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

```
while (s <= e){
    if (mid = target)
        { return mid; }
    else if (target > mid)
        { s = mid+1 }
    else {
        e = mid+1
    }
}
```

$\therefore \frac{N}{2^k} = 1 \Rightarrow N = 2^k$

$\Rightarrow \log(N) = \log(2^k) = k(\log 2)$

$\Rightarrow k = \frac{\log N}{\log 2} = \log_2 N$

* Total Numbers of comparison is k.

$\therefore$ $\boxed{\text{Time complexity} = O(\log N)}$

Bubble Sort:

- space complexity $O(1)$
  - no extra space required
  - also known as inplace sorting algorithms

Best Case: $O(N)$ → already sorted

Worst case: $O(N^2)$ → sorted in reverse

```
for(int i=0; i<N; i++) {
    for(int j=1; j<N-i; j++)
    {
        //swapping
    }
}
```

- outer loops runs : N − 1 times

- In the best case comparison happen only N − 1 times.

∴ Time Complexity $= O(N-1)$
$= O(N)$

Worst case:

Outer loops runing : N-1 times

Internal loops runing : N-i times

Total comparison:

$$(N-1) + (N-2) + (N-3) + (N-4) + \cdots + (N-(N-1))$$

$$= (N-1)(N) - (1+2+3+4 \cdots + N-1)$$

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

$$= \frac{2N^2 - 2N - N^2 + N}{2}$$

$$= \frac{N^2 - N}{2}$$

$$\therefore O\left(\frac{N^2 - N}{2}\right) = O(N^2)$$

CS CamScanner

Selection Sort :

```
for( i=0; i< N; i++){
    smallest = i
    for(j = i+1; j < N; j++){
        // operation
    }
    // operation
}
```

Comparison Taking: $N-1, N-2, N-3, \ldots 1, 0$

Total Comparison: $0+1+2+3+ \ldots N-1$

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

$$= \frac{N^2-N}{2}$$

$$\therefore O\left(\frac{N^2-N}{2}\right) = O(N^2)$$

CS CamScanner

Insertion Sort:

Worst Case: $O(N^2)$

Best Case: $O(N)$

```
for (i = 1 ; i < N-1 ; i++) {
    for (j = i+1 ; j > 0 ; j--) {
        //operation
        if (arr[j] < arr[j-1]) {
            //swap
        }
    }
}
```

Comparison taking: $N-1, N-2, N-3, \ldots 1$

Total Comparison: $1 + 2 + 3 + \ldots N-1$

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

$$O\left(\frac{N^2-N}{2}\right) = O(N^2)$$

Best Case: 1, 2, 3, 4, 5

Total comparison: $N-1$

$$O(N-1) = O(N)$$

CamScanner

Cyclic Sort: (1 to N)

```
i = 0
while (i < N) {
    correct = arr[i] - 1
    if (arr[i] != arr[correct])
    {        // swap
    }
    else {
        i++
    }
}
```

worst case: ~~5, 4, 3, 2, 1~~ 3, 5, 2, 1, 4

swaps = 4 and 3, 5, 2, 1, 4

last 5 swaps check 2, 5, 3, 1, 4

5, 2, 3, 1, 4

4, 2, 3, 1, 5

1, 2, 3, 4, 5

Total = 4 + 5

= (N-1) + N

= 2N - 1

$\therefore O(2N-1) = O(N)$

Example:

```
for (i = 1 ; i ≤ N ; i += k) {
    for (j = 1 ; j ≤ k ; j++) {
        // works
    }
}
```

Inner loop running, $O(k)$ time

Ans: $O(k * \text{how many times the outer loop is running})$

$i = 1, 1+k, 1+2k, 1+3k \ldots 1+xk$

and outer loop is running x times

$$1 + xk \le N$$

$$\Rightarrow x \le \frac{N-1}{k}$$

$$\therefore TC = O\left(k * \frac{N-1}{k}\right) = O(N)$$

CamScanner

Recursive Programs:

- Space complexity is not going to O(1). Cause in Recursive Programs functions are stayed at stack and capture some memory.

- In Recursive Programs ~~to~~ no two functions of same level execute at the same time and ~~to~~ no two functions of same level stayed at stack.

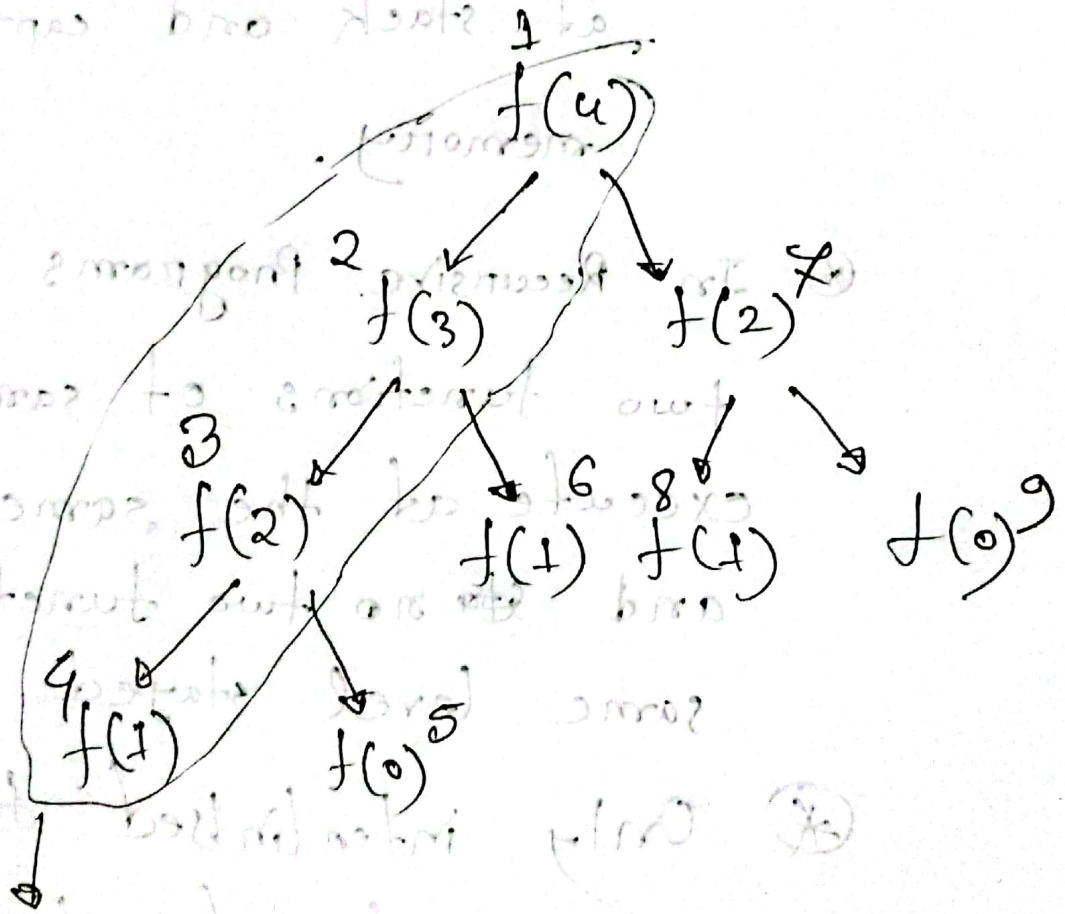
- Only interlinked functions are stayed at stack.

CamScanner

In Recursive Programs :

Space Complexity would be the height of the tree.

Ex: Fibonacci Numbers



height of the tree.

∴ Space Complexity here = $O(N)$

Two types of recursion:

1. Linear
   ↓
   Fibonacci

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

2. Divide and Conquer
   ↓
   Binary Search
   ↓
   $F(N) = F(\frac{N}{2}) + O(1)$

CamScanner

① Divide and Conquer Recurrences:

Form:

$$T(x) = a_1 T(b_1 x + \epsilon_1(n)) + a_2 T(b_2 x + \epsilon_2(n)) + \cdots + a_k T(b_k x + \epsilon_k(n)) + g(x) \quad \text{for } x \geq x_0$$

x_0 → constant

Binary Search:

$$T(N) = T(N/2) + C$$

comparing, $a_1 = 1$, $b_1 = \frac{1}{2}$, $\epsilon_1(n) = 0$, $g(x) = C$

another example:

$$T(N) = \underset{a_1}{9}\, T\left(\underset{b_1}{\frac{N}{3}}\right) + \underset{a_2}{\frac{4}{3}}\, T\left(\underset{b_2}{\frac{5N}{6}}\right) + \underset{g(N)}{4N^3}$$

$$T(N) = \underbrace{\underset{a_1}{2}\,T\left(\underset{b_1}{\frac{N}{2}}\right)} + \underset{g(N)}{(N-1)}$$

↳ (when · get ans from this + what ~~a~~ we are doing with ans)

— takes how much times → means $g(N)$

How to solve:

1. Plug and chug
2. Master's theorem
3. Akra - Bazzi

Akra - Bazzi:

$$T(x) = \Theta\left(x^p + x^p \int_1^x \frac{g(u)}{u^{p+1}} \cdot du\right)$$

$$p \rightarrow \boxed{\sum_{i=1}^{k} a_i b_i = 1}$$

Binary search:

$$T(N) = T\left(\frac{N}{2}\right) + C$$

$a_1 = 1 \qquad g(x) = C$

$b_1 = \frac{1}{2}$

we know, $\sum_{i=1}^{k} a_i b_i^p = 1$

$$a_1 b_1^p = 1$$

$$\Rightarrow 1 \times \left(\frac{1}{2}\right)^p = 1$$

$$\therefore p = 0$$

$$T(x) = \Theta\left(x^0 + x^0 \int_1^x \frac{c}{u^{0+1}} \, du\right)$$

$$= \Theta\left(1 + \int_1^x \frac{c}{u} \, du\right)$$

$$= \Theta\left(1 + c \cdot \log x\right)$$

$$= \Theta\left(\log x\right)$$

For N size we can say, it would be

$$\Theta(\log N)$$

Ex: $T(N) = 2T\left(\frac{N}{2}\right) + (N-1)$ → Merge sort

$a_1 = 2 \qquad g(N) = N-1$

$b_1 = \frac{1}{2}$

we know

$$\sum_{i=1}^{k} a_i b_i^p = 1$$

$$\therefore a_1 b_1^p = 1$$

$$\Rightarrow 2 \times \left(\frac{1}{2}\right)^p = 1$$

$$\Rightarrow p = 1$$

$$T(x) = \theta\left(x^1 + x^1 \int_1^x \frac{u-1}{u^{1+1}}\, du\right)$$

$$= \theta\left(x + x \int_1^x \left(\frac{u}{u^2} - \frac{1}{u^2}\right) du\right)$$

$$= \theta\left(x + x\left[\int_1^x \frac{1}{u}\, du - \int_1^x \frac{1}{u^2}\, du\right]\right)$$

$$= \theta\left(x + x\left[[\log u]_1^x + [u^{-1}]_1^x\right]\right)$$

$$= \theta\left(x + x\left(\log x + \frac{1}{x} - 1\right)\right)$$

$= \theta(\cancel{x} + x\log x + 1 - \cancel{x})$

$= \theta(x\log x + 1)$

$= \theta(x\log x)$ → Time complexity

For size of N, Time complexity of merge sort is $O(N\log N)$

Example:

$$T(N) = 2T\left(\frac{N}{2}\right) + \frac{8}{9}T\left(\frac{3N}{4}\right) + N^2$$

$a_1 = 2 \qquad a_2 = \frac{8}{9} \qquad g(N) = N^2$

$b_1 = \frac{1}{2} \qquad b_2 = \frac{3}{4}$

$$\therefore \quad 2 \times \left(\frac{1}{2}\right)^p + \frac{8}{9} \times \left(\frac{3}{4}\right)^p = 1$$

$$\therefore p = 2$$

$$T(x) = \theta\left(x^2 + x^2 \int_1^x \frac{u^2}{u^3}\, du\right)$$

$$= \theta\left(x^2 + x^2 \left[\log u\right]_1^x\right)$$

$$= \theta\left(x^2 + x^2 \cdot \log x\right)$$

$$= \theta\left(x^2 \log x\right)$$

$$\therefore \theta\left(N^2 \log N\right)$$

CamScanner

If we can't find value of p

Ex) $T(x) = 3T(x/3) + 4T(x/4) + x^2$

$\therefore 3 \times \left(\frac{1}{3}\right)^p + 4 \times \left(\frac{1}{4}\right)^p = 1$

$\boxed{1 < p < 2}$ $p = 1.5$ ~~60~~

NOTE: when $p <$ power of $(g(x))$

then answer will be $= g(x)$

$\therefore$ Answer here $= O(g(x))$

$= O(x^2)$

Solving Linear Recurrences:

Ex: $F(N) = F(N-1) + F(N-2)$

Form:

$$f(x) = a_1 f(x-1) + a_2 f(x-2) + a_3 f(x-3) + \cdots + a_n f(x-n)$$

$$\therefore f(x) = \sum_{i=1}^{n} a_i f(x-i)$$

, a_i and $n \to$ fixed

$n \to$ order of recurrence

Solution for fibonacci:

$$f(n) = f(n-1) + f(n-2)$$

Steps:

1. Put $f(n) = \alpha^n$ for some constant α

$$\therefore \alpha^n = \alpha^{n-1} + \alpha^{n-2}$$

$$\Rightarrow \alpha^n - \alpha^{n-1} - \alpha^{n-2} = 0$$

$$\Rightarrow \frac{\alpha^n}{\alpha^{n-2}} - \frac{\alpha^{n-1}}{\alpha^{n-2}} - \frac{\alpha^{n-2}}{\alpha^{n-2}} = 0$$

$$\Rightarrow \alpha^2 - \alpha - 1 = 0$$

$$\alpha = \frac{1 \pm \sqrt{5}}{2}$$

$$\alpha_1 = \frac{1+\sqrt{5}}{2}, \quad \alpha_2 = \frac{1-\sqrt{5}}{2}$$

CS CamScanner

(2) $f(n) = c_1 \alpha_1^n + c_2 \alpha_2^n$ → it's a sol^n

$= c_1 \left(\frac{1+\sqrt{5}}{2}\right)^n + c_2 \left(\frac{1-\sqrt{5}}{2}\right)^n$ — (2) → take constants no. of roots and multiply with roots power n.

(3) Facts: no. of roots = no. of ans we already know

$f(0) = 0$
$f(1) = 1$

$\therefore f(0) = 0$
$\Rightarrow c_1 + c_2 = 0$
$\Rightarrow c_1 = -c_2$

$f(1) = 1$
$\Rightarrow c_1\left(\frac{1+\sqrt{5}}{2}\right) + c_2\left(\frac{1-\sqrt{5}}{2}\right) = 1$
$\Rightarrow c_1\left(\frac{1+\sqrt{5}}{2}\right) - c_1\left(\frac{1-\sqrt{5}}{2}\right) = 1$
$\Rightarrow c_1 = \frac{1}{\sqrt{5}}$
$c_2 = -\frac{1}{\sqrt{5}}$

$\therefore$ puting c_1, c_2 in (2)

$$f(n) = \frac{1}{\sqrt{5}}\left(\frac{1+\sqrt{5}}{2}\right)^n - \frac{1}{\sqrt{5}}\left(\frac{1-\sqrt{5}}{2}\right)^n$$

$\downarrow$

formula for n^{th} fibonacci numbers

$$f(n) = \frac{1}{\sqrt{5}}\left[\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n\right]$$

as $n \to \infty$ $\left(\frac{1-\sqrt{5}}{2}\right)^n \to 0$, so

this less dominating term.

Hence ignore.

$\therefore$ Time complexity for n^{th} fibonacci numbers is: $O\left(\frac{1+\sqrt{5}}{2}\right)^n$ $\rightarrow$ Golden ratio

$$T(N) = O(1.6180)^n$$

Non-homogenous linear recurrence

$$f(n) = a_1 f(n-1) + a_2 f(n-2) + a_3 f(n-3) + \cdots + a_d f(n-d) + g(n)$$

when this extran function is there, it is non-homogenous

1. Replace $g(n)$ by 0 and follow usual step

2. Take $g(n)$ on one side and find particular solutions.

3. Add both solutions together

CS CamScanner

Ex:

$$f(n) = 4f(n-1) + 3^n, \quad f(1) = 1$$

$$f(n) = 4f(n-1)$$

$$\alpha^n = 4\alpha^{n-1}$$

$$\Rightarrow \alpha^n - 4\alpha^{n-1} = 0$$

$$\Rightarrow \alpha - 4 = 0$$

$$\Rightarrow \alpha = 4$$

$$f(n) = c_1 \alpha^n$$

$$f(1) = c_1 (4)^n$$

Then,

$$f(n) - 4f(n-1) = 3^n$$

Guess something that is similar to $g(n)$

<u>Guess:</u> $f(n) = c3^n$

$$c3^n - 4c3^{n-1} = 3^n$$

$$\Rightarrow c = -3$$

$$\therefore f(n) = -3 \cdot 3^n = -3^{n+1}$$

~~3 Add both~~ Sol

$$f(n) = c_1 4^n - 3^{n+1}$$

$f(1) = 1$,

$$1 = c_1 4 - 3$$

$$\Rightarrow c_1 = \frac{5}{2}$$

$$\therefore f(n) = \frac{5}{2} 4^n - 3^{n+1}$$

Time complexity: $O\left(\frac{5}{2} 4^n - 3^{n+1}\right)$

CamScanner

How do we guess particular sol^n:

- If $g(n)$ is exponential, guess of same type:

  Ex: $g(n) = 2^n + 3^n$

  Guess: $f(n) = a2^n + b3^n$.

- If it is polynomial, guess of same degree.

Ex: $g(n) = n^2 - 1 \Rightarrow$ guess of same degree

$f(n) = an^2 + bn + c$

- combination; $g(n) = 2^n + n$

  $f(n) = a2^n + (bn + c)$

Lets say we guess $f(n) = a2^n$ and if it fails, then try $f(n) = (an+b)2^n$, if it's also fail, then increase the degree

$$f(n) = (a^2n + bn + c)2^n$$

Ex: $f(n) = 2f(n-1) + 2^n$; $f(0) = 1$

① $\boxed{2^n = 0}$

$$f(n) = 2f(n-1)$$

$$f(n) = \alpha^n$$

$$\therefore \alpha^n = 2 \cdot \alpha^{n-1}$$

$$\therefore \alpha = 2$$

② Guess particular sol^n.

$$f(n) = a2^n$$

$$a2^n = 2a2^{n-1} + 2^n$$

$\Rightarrow$ $a = a+1$ (wrong) ✗

then, guess again

$$f(n) = (an+b)\cdot 2^n$$

$$(an+b)\cdot 2^n = \cancel{2\cdot(an+b)\cdot 2^n}\ 2(a(n-1)+b)\cdot 2^{n-1} + 2^n$$

$$\Rightarrow \cancel{an} + \cancel{b} = \cancel{an} - a + \cancel{b} + 1$$

$$\Rightarrow a = 1 \quad \boxed{\text{Discard } b}$$

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

<u>General:</u>

$$f(n) = n\cdot 2^n + c_1 2^n$$

$$f(0) = 1$$

$$\Rightarrow c_1 = 1$$

$$\therefore f(n) = 2^n + n\cdot 2^n$$

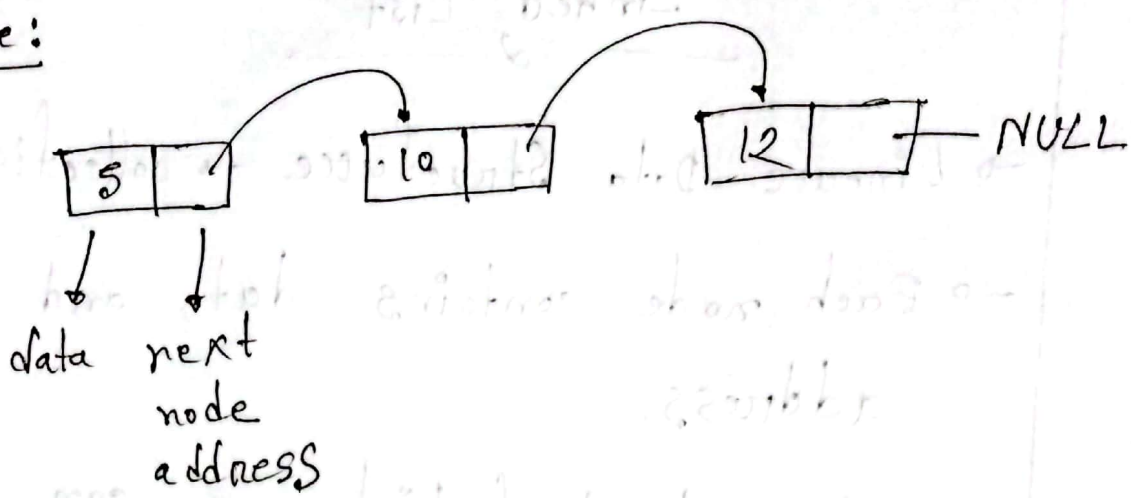
CamScanner

Linked List

- Linear Data Structure → collection of nodes
- Each node contains data and next node address.
- Using Linked List we can insert or remove any elements of any position in the linked list.

	Linked List	Array
1. Data Structure	Non-contiguous	Contiguous
2. Memory Allocation	individual elements	whole array
3. Insertion/Deletion	Efficient	Inefficient
4. Access	Sequential	Random

Example:




```
class Node {
    public:
        int data;
        Node* next;
    //constructor
    Node (int data){
        this -> data = data;
        this -> next = NULL;
    }
```

create a Node:

```
Node* newnode = new Node(val);
```

Access:

```
newnode -> data
newnode -> next
```

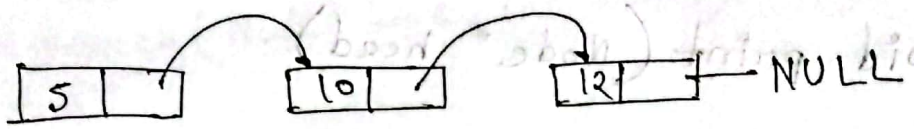
Types of Linked List

- Singly Linked List
- Doubly Linked List
- Circulare Linked List
- Circulare Doubly Linked List

CS CamScanner

Singly Linked List

Ex:



Operations on Singly Linked List:

- → Traversal
- → Searching
- → Length
- → Insertion:
  - → at the beginning (head)
  - → at the end (tail)
  - → at specific position.
- → Deletion:
  - → from the beginning (Head)
  - → from the end (tail)
  - → at specific node

Traversal:

```
void print (Node* head)
{
    // Start from the beginning
    Node* temp = head;

    // traverse untill null pointers
    while (temp != NULL)
    {
        cout << temp -> data << " ";
        temp = temp -> next;
    }
    cout << endl;
}
```

Time Complexity	: O(N)
Space Complexity	: O(1)

call function:

```
print (head);
```

Searching in Singly Linked List:

```
bool search(Node* head, int target)
{
    Node* temp = head;
    while(temp != NULL)
    {
        if(temp -> data == target)
        {
            return true;
        }
        temp = temp -> next;
    }
    return false;
}
```

Time Complexity : O(N)

Space Complexity : O(1)

call function:

```
search(head, 5);
```

Finding Length in Singly Linked List:

```
int Length(Node* head)
{
    int cnt = 0;
    Node* temp = head;
    while (temp != NULL)
    {
        cnt++;
        temp = temp -> next;
    }
    return cnt;
}
```

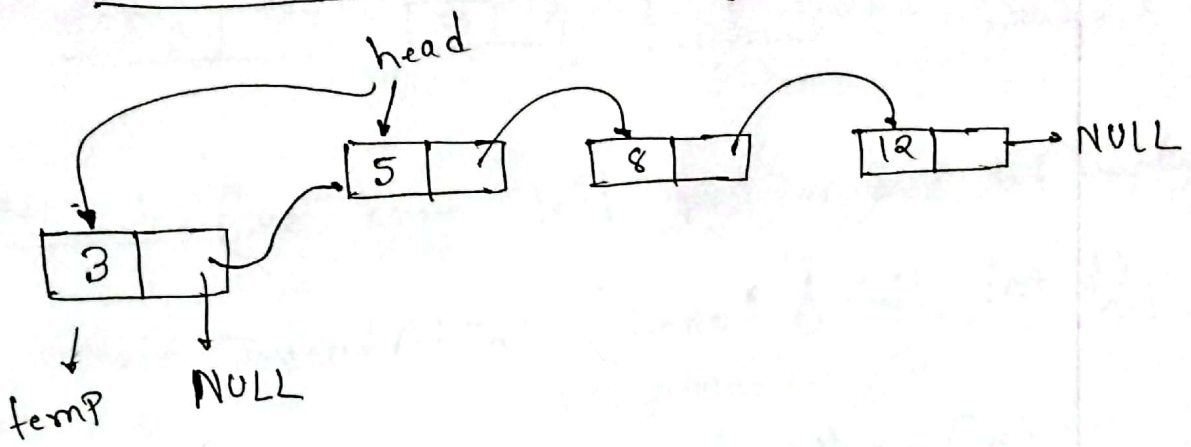
Time Complexity: O(N)
Space Complexity: O(1)

//call functions

```
Length(head);
```

Insertion in Singly Linked List:

1. Insert at the beginning:



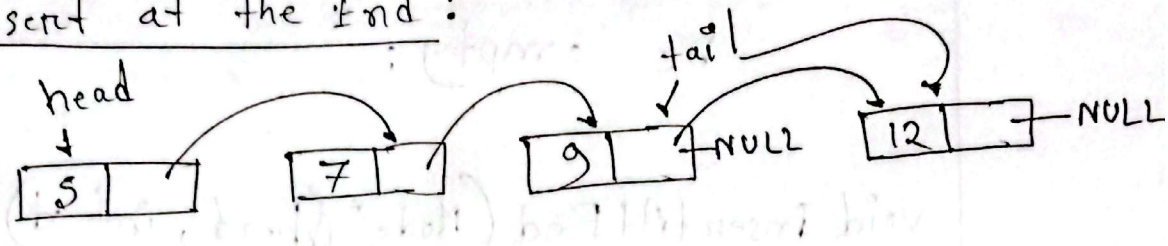

```
void InsertAthead (Node* &head, int d)
{
    Node* temp = new Node (d);   → Create new node
    temp → next = head;
    head = temp;
}
```

Time Complexity : O(1)
Space Complexity : O(1)

call function:

```
InsertAthead (head, 3);
```

2. Insert at the End :



Case - 1 : (we know tail position and Not empty)

```
void InsertAtTail (Node* &tail, int d)
{
    Node* temp = new Node (d);
    tail -> next = temp;
    tail = temp;
}
```

call function :

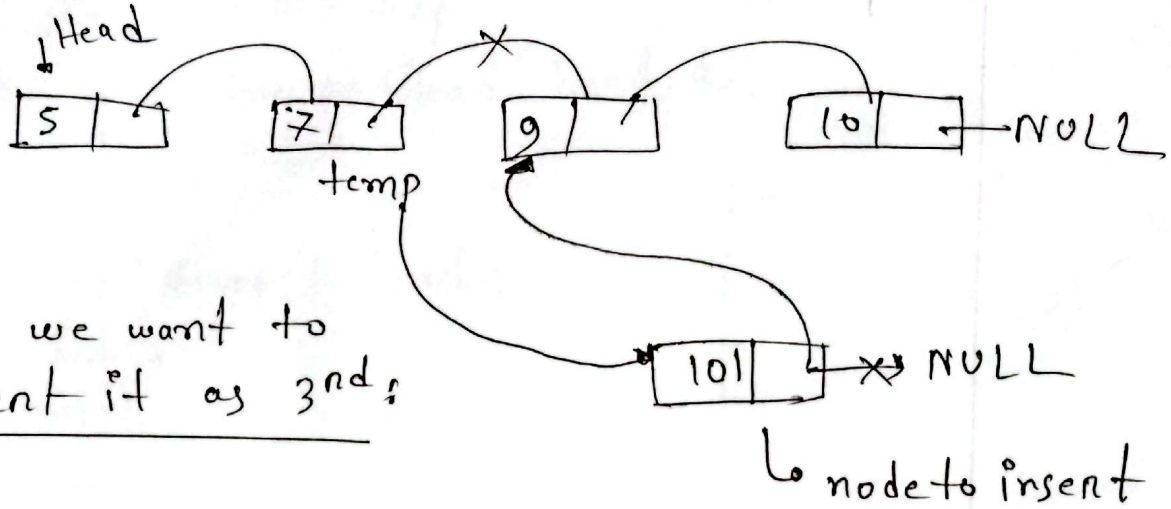
```
InsertAtTail (tail, d);
```

Case-2: we don't know tail and it can be empty:

```
void InsertAtEnd (Node* &head, int d)
{
    Node * temp = new Node(d);

    //empty list
    if (head == NULL)
    {
        ~~return temp~~;
        head = temp;
        return;
    }

    Node * curr = head;
    while (curr -> next != NULL)
    {
        curr = curr -> next;
    }

    curr -> next = temp;

}
```

T.C = O(N)
S.C = O(1)

Insert at a Specific Position of the Singly Linked List:



Suppose we want to insert it as 3rd:

n → position

traverse (n-1)

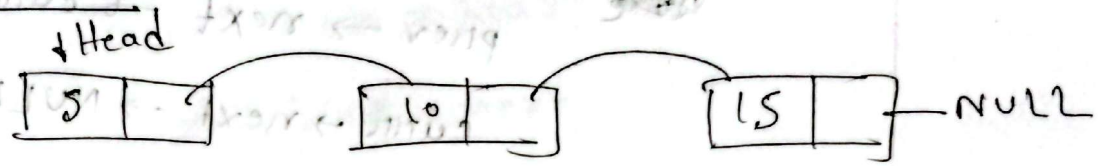
```
void InsertAtPosition(Node* &head, Node* &tail, int pos, int d)
{
    if (pos == 1){
        InsertAtHead(head, d);
        return;
    }

    Node* temp = head;
    int cnt = 1
    while (cnt < pos-1)
    {
        temp = temp -> next
        cnt++;
    }
    if (temp -> next == NULL) {
        InsertAtTail(tail, d);
        return;
    }

    Node* nodetoinsert = new Node(d);
    nodetoinsert -> next = temp -> next;
    temp -> next = nodetoinsert;
}
```

T.C = O(N)
S.C = O(1)

<u>call function:</u> InsertAtPosition(head, tail, 5, 7);

Deletion in Singly Linked List

First Node :



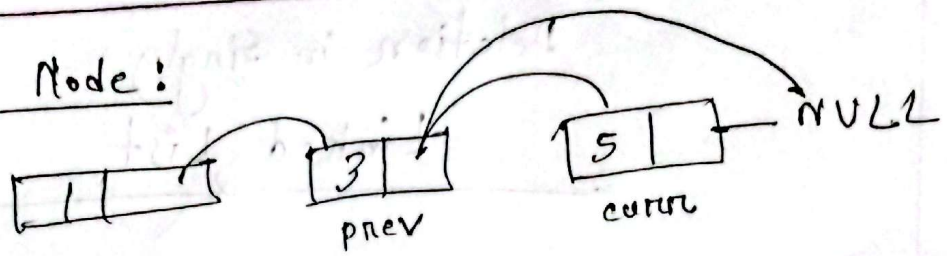

```
Node* temp = head;
head = head -> next;
temp -> next = NULL;
delete temp;
```

Time Complexity → O(1)

Space Complexity → O(1)

CamScanner

Last Node:



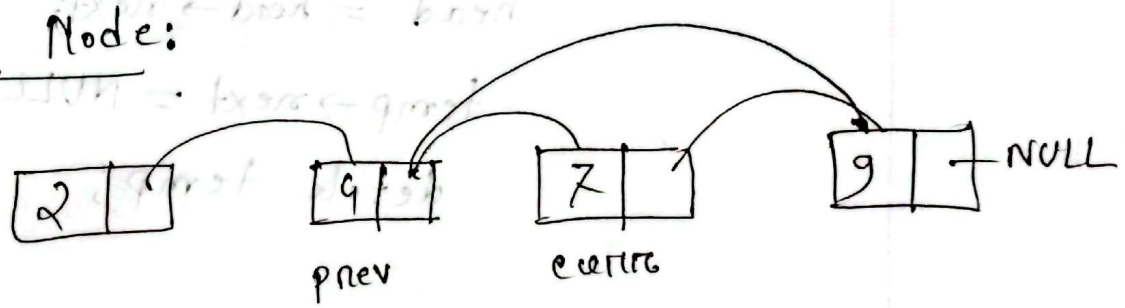
prev → next → curr → next

curr → next → NULL

delete curr;

T.C → O(N)
S.C → O(1)

Middle Node:



prev → next → curr → next

curr → next → NULL

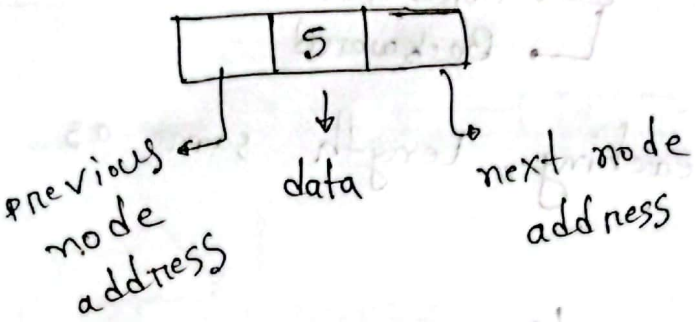
delete curr;

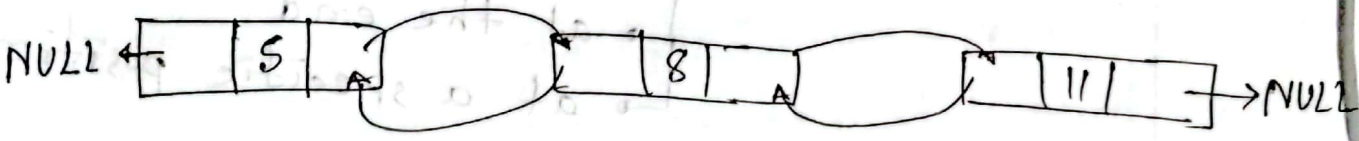
T.C → O(N)
S.C → O(1)

```
void DeleteNode (Node* &head, int pos)
{
    if (pos == 1) {
        Node* temp = head;
        head = head -> next;
        temp -> next = NULL;
        delete temp;
    }
    else {
        Node* curr = head;
        Node* prev = NULL;
        int cnt = 1;
        while (cnt < pos) {
            prev = curr;
            curr = curr -> next;
            cnt++;
        }
        prev -> next = curr -> next;
        curr -> next = NULL;
        delete curr;
    }
```

CamScanner

Doubly Linked List






```
class Node {
    public:
        Node* prev;
        int data;
        Node* next;
    Node (int d){
        this -> data = d;
        this -> prev = NULL;
        this -> next = NULL;
    }
}
```

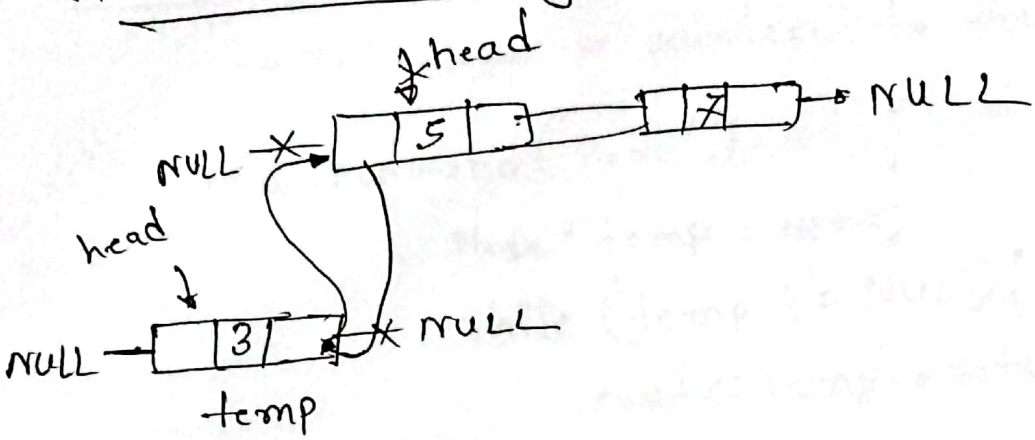
Operations:

- Traversal
  - Forward
  - Backward
- Searching, Length same as Singly Linked List
- Insertion
  - at the beginning
  - at the end
  - at a specific position
- Deletion
  - From the beginning
  - From the end
  - at a specific position

CamScanner

Insertion in Doubly Linked List:

At the begining:



→ Node * temp = new Node (d)

→ temp → next = head

→ head → prev = temp

→ head = temp

If head is NULL, then,

```
If (head != NULL) {
        head -> prev = temp
}
```

Traversal in Doubly Linked List:

Forward Traversal:

→ Initialize a pointer to the head.

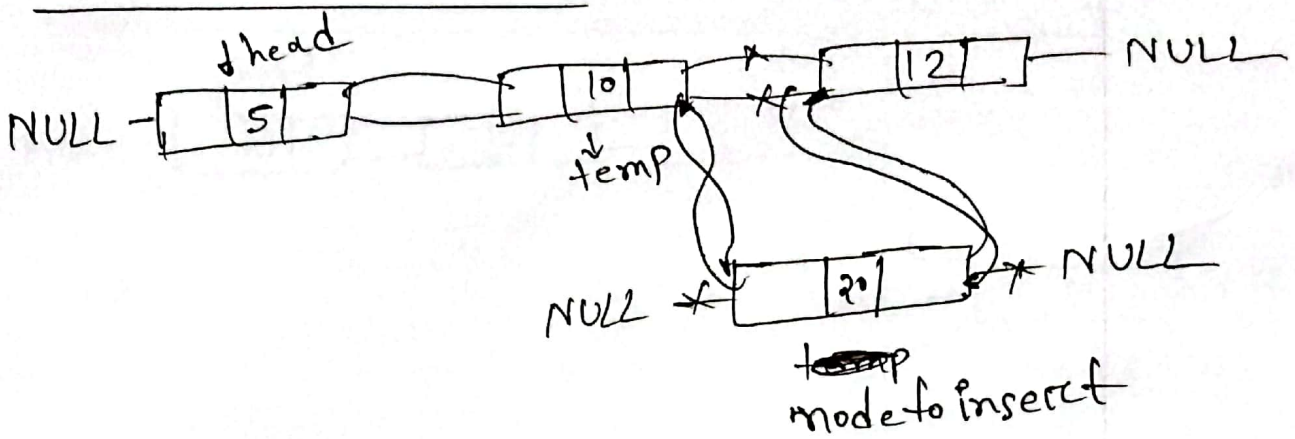
```
void Forward (Node *&head){
        Node * temp = head;
        while (temp != NULL){
                cout<<temp -> data<<" ";
                temp = temp -> next;
        }
        cout <<endl;
}
```

Backward Traversal:

→ Initialize a pointer to the tail

```
void Backward (Node* &tail){
        Node* temp =tail;
        while(temp != NULL){
                cout << temp-> data<<" ";
                temp = temp-> prev;
        }
        cout<<endl;
}
```

Insert at middle:



→ Node * node to insert = new Node(d)

node to insert → next = temp → next

temp → next → prev = node to insert

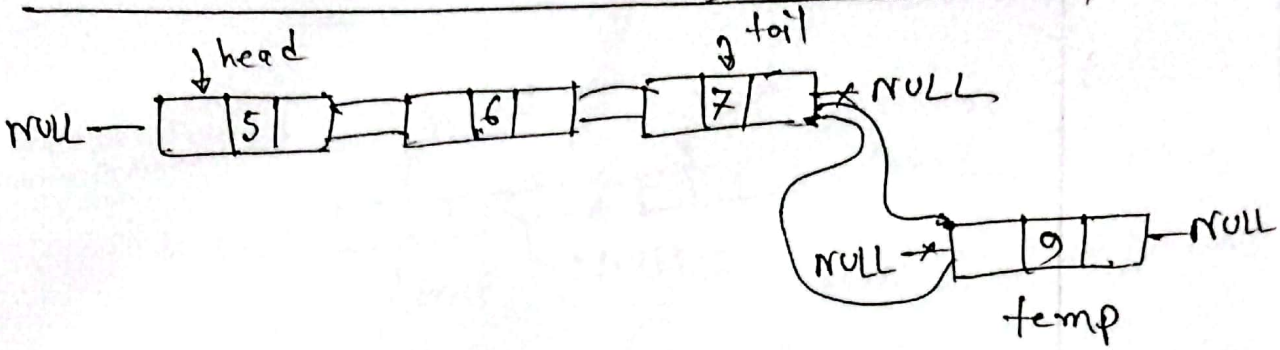
temp → next = node to insert

node insert → prev = temp

If pos = 1, Insert At head

pos = Last, Insert At tail

Insert At the End of Doubly Linked List:



→ Node *temp = new Node (d);

→ tail → next = temp

→ temp → prev = tail

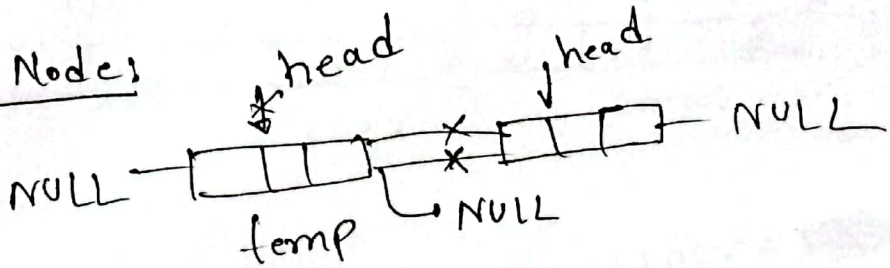
→ tail = temp

For empty list,

```
if (tail == NULL) {
        tail == temp;
        return;
}
```

Delete

First Node:



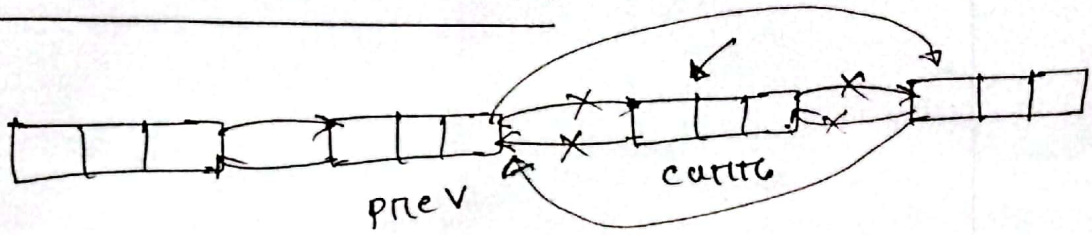
→ temp → next → prev = NULL

→ head = temp → next

→ temp → next = NULL

delete temp;

Delete the Middle or Last:



curr->next->prev = prev

curr->prev = NULL

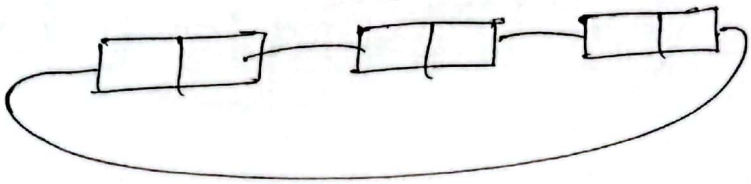
prev->next = curr->next

curr->next = NULL

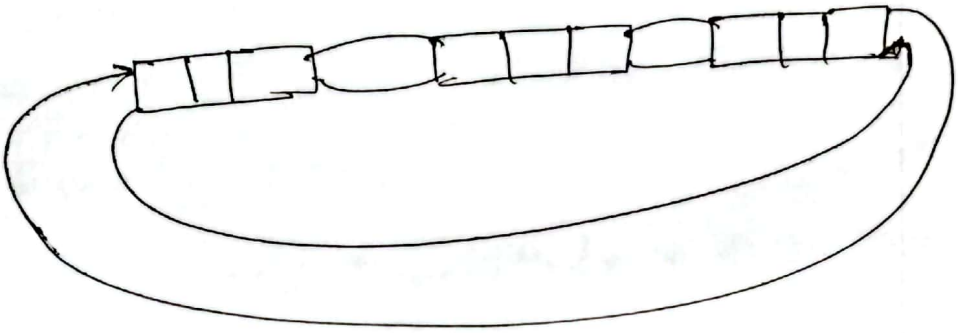
delete curr;

Circular Linked List

Singly :

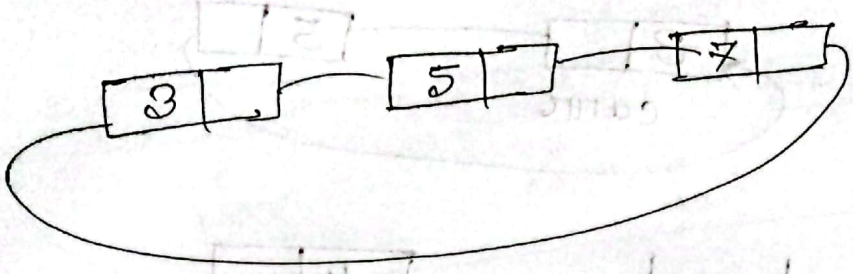


Doubly :



CamScanner

Circulare Singly Linked List

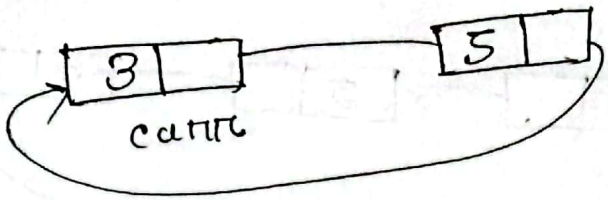


Insert:

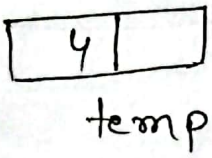
Empty list:

```
Node* newNode = new Node(d)
tail = newNode;
tail -> next = newNode
```

Non - empty:



i/p → element



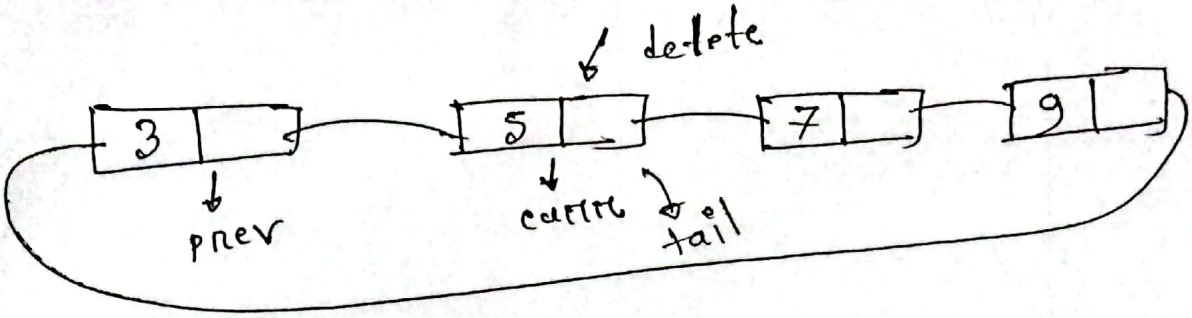
$\boxed{\text{here} = 3}$

→ Node * temp = new Node(4);

→ temp → next = curr → next

→ curr → next = temp

Delete



i/p element = 5, prev → next = curr → next

1 Node

prev == curr

tail = NULL

≥ 2 Node

tail == curr

tail = prev

curr → next = NULL;

delete curr

Pattern Printing:-

```
r { 4 - * * * * ← 4 }
  { 3 - * * * ← 3   } c
  { 2 - * * ← 2     }
  { 1 - * ← 1       }
```

```
so, if (r = 0) return;
    if (c < r) {
        print (*)
        triangle (r, c+1)
    }
    else,  new line
           triangle (r-1, 0)
```

Bubble sort: → each pass (swap if previous > next)

arr = {4, 3, 6, 2, 8}

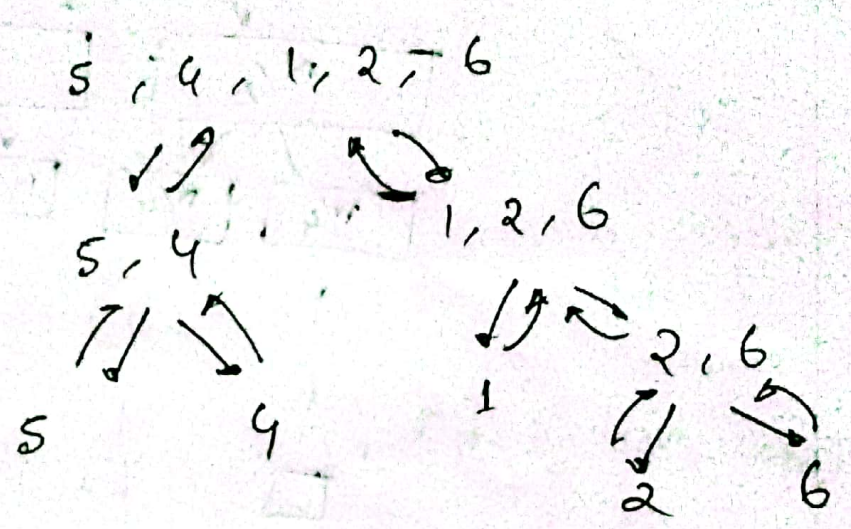
bubble. (arr, arr.length-1, 0)
↳ index

```
if ( r == 0) return;

if (c < r) {
        if (arr[c] > arr[c+1]) {
                swap (arr[c], arr[c+1])
        }
        bubble (arr, c, c+1);
    }
    else, bubble (arr, r-1, 0);
```

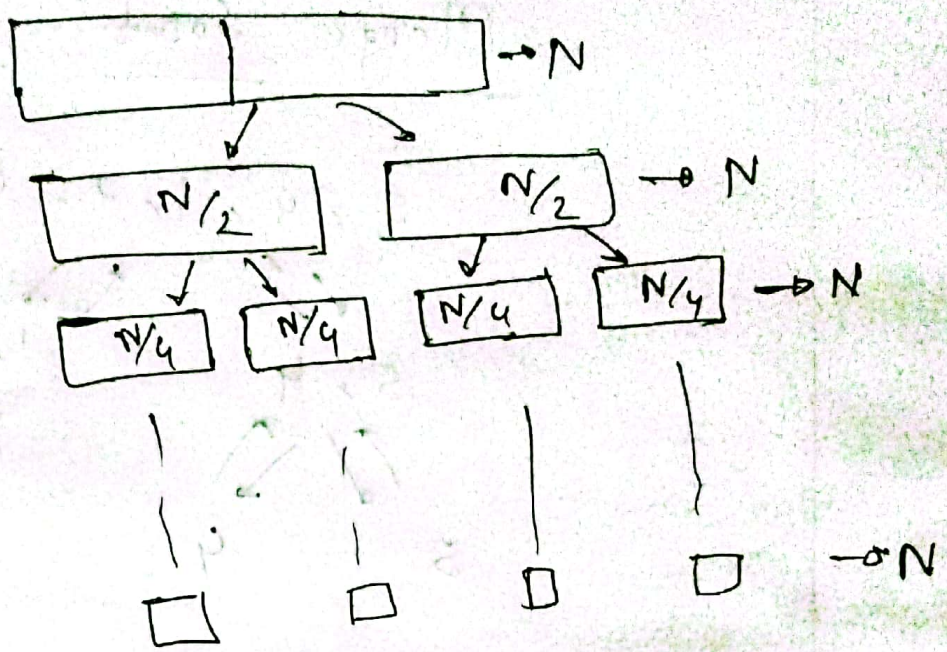
Selection Sort : each pass → max element right pos.

```
sort(arr, length, 0, 0)
          r       c   max

if(r==0)    return

if (c < r) {
        if (arr(c) > arr[max]) {
            sort(arr, r, c+1, c)
        }
        else {
            sort(arr, r, c+1, max)
        }
    }
else {
    swap(arr(max), arr(r-1))
    sort(arr, r-1, 0, 0)
}
```

Merge Sort



- First divide 2 parts
- sort each parts
- merge the 2 parts



At every level, N element are being merged.

Levels

0 → N N

1 → N/2

N/4

k^{th} 1 1

$$\therefore \quad 1 = \frac{N}{2^k} \Rightarrow \boxed{k = \log_2 N}$$

$\therefore$ Total Time Complexity : $O(N * \log N)$

Space Complexity :

Auxiliary Space : $O(N)$

$$T(N) = T(N/2) + T(N/2) + (N-1)$$

$$= 2T(N/2) + (N-1)$$

from Akna-Bazzi.

$$a = 2, \; b = \frac{1}{2}$$

$$2 \times \frac{1}{2^p} = 1 \Rightarrow p = 1$$

$$T(x) = x + x \int_1^x \left(\frac{u+1}{u^{1+1}}\right) du$$

$$= x + x \int_1^x \left(\frac{1}{u} - \frac{1}{u^2}\right) du$$

$$= x + x \left[\log u + \frac{1}{u}\right]_1^x$$

$$= x + x \left[\log x + \frac{1}{x} - 1\right]$$

$$= x + x \log x + 1 - x.$$

$$= x \log x$$

$\therefore$ T.C $= O(N * \log N)$

Quick Sort

- Pick Pivot

  In left side of pivot bring all elements that are < pivot and In righ side of pivot bring all elements that are > pivot

- 5, 4, 1, 3, 2

Let's pivot = 4

1, 3, 2, $\boxed{4}$, 5

so, each pass pivot element will come at it's right position

Put pivot at it's correct position

s p e

5, 4, 3, 2, 1

s,p e

1, 4, 3, 2, 5

s,e

1, 2, 3, 4, 5

low

High

pivot comes at correct position

while → arr[s] < p
s++;

while → arr[e] > p
e--;

pivot →
1. random
2. corner
3. mid

$$\underbrace{\qquad\qquad}_{k}\ p\ \underbrace{\qquad\qquad}_{N-k-1}$$

$$T(N) = T(k) + T(N-k-1) + O(N)$$

<u>worst case:</u> pivot → largest / smallest

$\boxed{k = 0}$

$$T(N) = T(N-1) + O(N)$$

$$= O(N^2)$$

<u>Best case:</u> pivot → mid

$$T(N) = T(N/2) + T(N/2) + O(N)$$

$$= 2T(N/2) + O(N)$$

$$= O(N * \log N)$$

→ Quick sort

- Not stable
- In-place (preffered over merge sort because merge sort taking O(N) extra space)
- Merge sort is beffere in linked lists due to memory allocation.

Hybrid Sorting Algorithms (Tim sort)

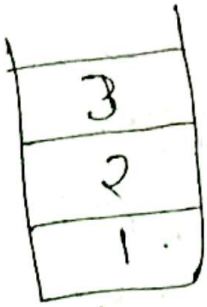
Merge Sort + Insertion Sort

↓

works beffere for partially sorted arrays

stack

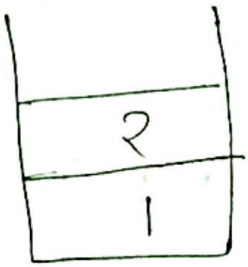
st. push (1)
st. push (2)
st. push (3)



stack → Lifo → Last in first out

st.top() → 3

st.pop() →



Queue

q.push(1)
q.push(2)
q.push(3)

queue → fifo → first in first out

1	2	3

q.front() → 1

q.pop() → | 2 | 3 |

Counting Sort

2

input array:

2	5	3	0	2	3	0	3

count array:

0	0	0	0	0	0	0	0	0	0
0	1	2	3	4	5	6	7	8	9

After counting element,

~~cout~~

count array:

2	0	2	3	0	1	0	0	0	0
0	1	2	3	4	5	6	7	8	9

CamScanner

Counting Sort

input Array :

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

Step-1: max = 5

step-2: count Array → length (5+1)

count Array

0	0	0	0	0	0
0	1	2	3	4	5

Step-3: store count of each element

countArray

2	0	2	3	0	1
0	1	2	3	4	5

Step-4: prefix sum.

countArray[i] = countArray[i-1] + countArray[i]

countArray

2	2	4	7	7	8
0	1	2	3	4	5

CamScanner

from end of Array

Step-5: outputArray [countArray [inputArray [i]] - 1] = inputArray [i]

countArray [inputArray [i]] = countArray [inputArray [i]] --

~~output~~ input Array :

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

countArray :

0	1	2	3	4	5
2	2	4	7	7	8

7-1

outputArray :

0	1	2	3	4	5	6	7
						3	

Step-6: input :

0	1	2	3	4	5	6	7
2	5	3	0	2	3	0	3

countArray:

0	1	2	3	4	5
2	2	4	6	7	8

2-1

output :

0	1	2	3	4	5	6	7
	0					3	

CamScanner

Step-7:

input:

2	5	3	0	2	3	0	3
0	1	2	3	4	5	6	7

3

count:

1	2	4	6	7	8
0	1	2	3	4	5

5-1

output:

	0				3	3	
0	1	2	3	4	5	6	7

repeating this:

output:

0	0	2	2	3	3	3	5
0	1	2	3	4	5	6	7

Time Complexity: $O(N+M)$

N → size of input array

M → size of output array

Radix Sort

input array: 170 45 75 90 802 24 2 66

Step-1: get max of array → 802

802 → 3 digits

we have to iterate three times.

Step-2: Sort based on specific position.

for 0 position:

0	1	2	3	4	5	6	7	8	9
90, 170		2, 802		24	75, 45	66			

for 10th position:

0	1	2	3	4	5	6	7	8	9
802		24, 2		45		66	75, 170		90

for 100th position:

0	1	2	3	4	5	6	7	8	9
2	170							802	
24									
45									
66									
75									

output:

2 24 45 66 75 170 802

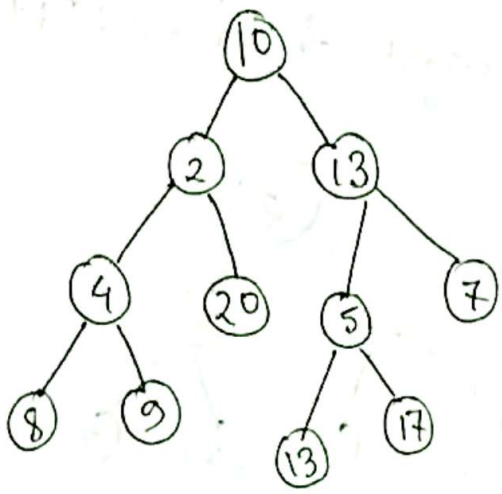
Time complexity: $O(d * (n+b))$

d → no. of digits

n → size of input array

b → the base of number system

Binary Tree



Inorder: 8 4 9 2 20 10 13 5 17 13 7

Pre order: 10 2 4 8 9 20 13 5 13 17 7

Post Order: 8 9 4 20 2 13 17 5 7 13 10

Inorder → left – root – right

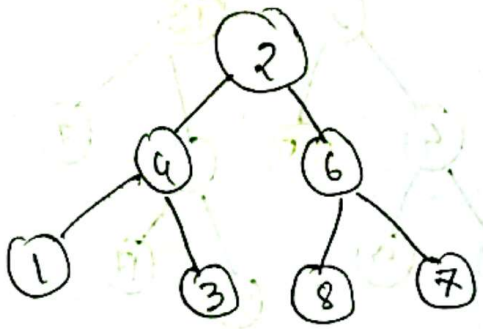
Pre order → root – left – right

Postorder → left – right – root

CamScanner

$i \rightarrow$ Left child $\rightarrow 2*i+1$

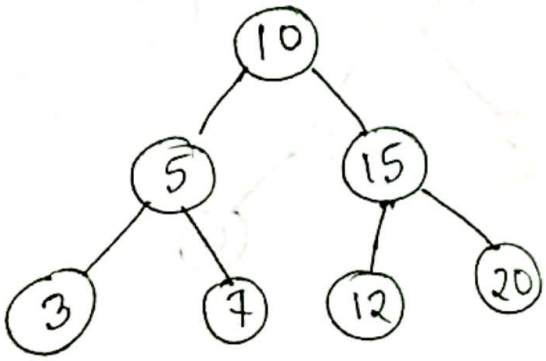
$i \rightarrow$ right child $\rightarrow 2*i+2$



2	4	6	1	3	8	7
0	1	2	3	4	5	6

Binary Search Tree (BST)

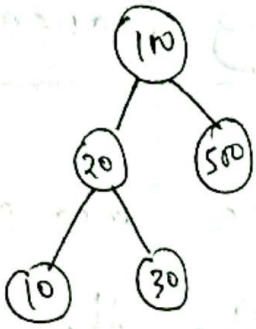
Binary Search Tree means that the left child is smaller than the root and the right child is larger that the root.



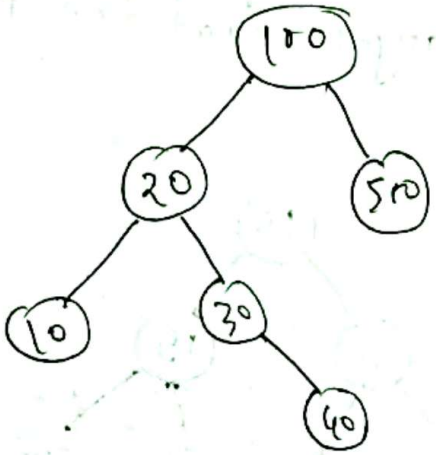
Inorder: 3 5 7 10 12 15 20

10 → Inorder Successor — 12

10 → Inorder Predecessor — 7

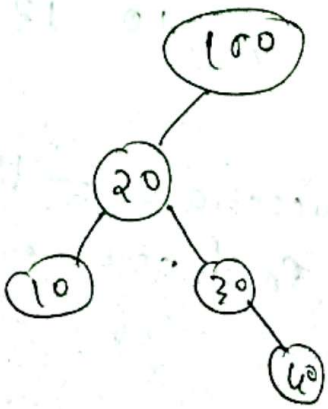


Insert - 40

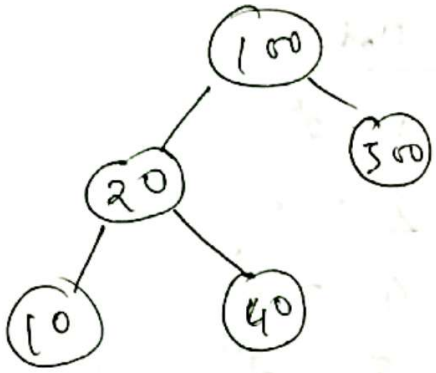


Delete:

1. Leaf Node - 500

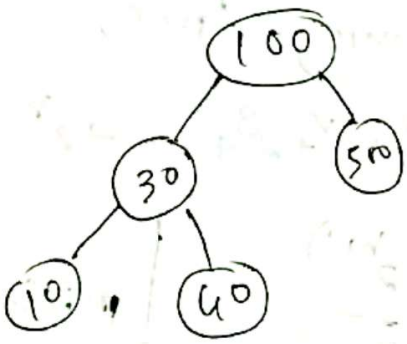


2. ~~Dele~~ Node with one child. → 30



[swap with child and then assign null]

3. Node with 2 child → 20



replace with inorder successor, make it a leaf node and then delete.

Height of a Binary Tree

Level	Max Nodes
0	$1 = 2^0$
1	$2 = 2^1$
2	$4 = 2^2$
3	$8 = 2^3$
⋮	
h	2^h

For complete binary tree,

$$n = 1 + 2^1 + 2^2 + 2^3 + \cdots + 2^h$$

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

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

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

$$\log_2 n \approx h+1$$

$$h \approx O(\log n)$$

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

Minimum → $O(\log n)$

Maximum → $O(n)$ ← skewed tree

Heap

1. Binary Tree (complete)

2. For Max Heap
   → root > left and right child

3. For Min Heap
   → root < left and right child

⊗ we used array to represent.

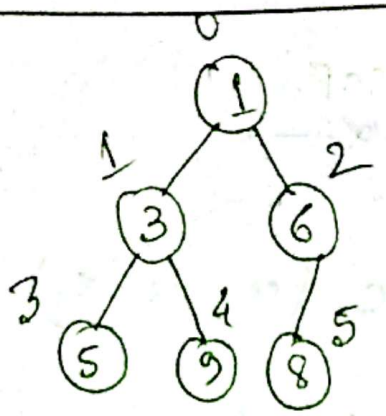
$\frac{i-1}{2}$ → parent

$2*i+1$ → left child

$2*i+2$ → right child

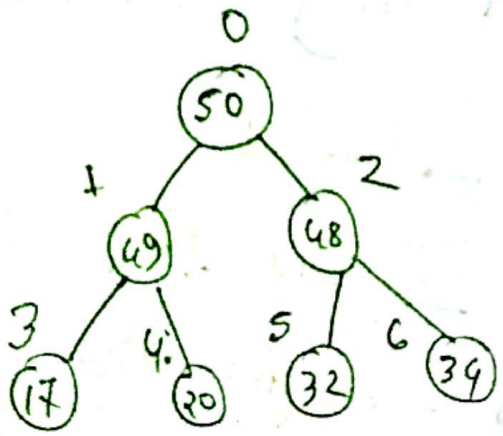
CamScanner

Min Heap



1	3	6	5	9	8
0	1	2	3	4	5

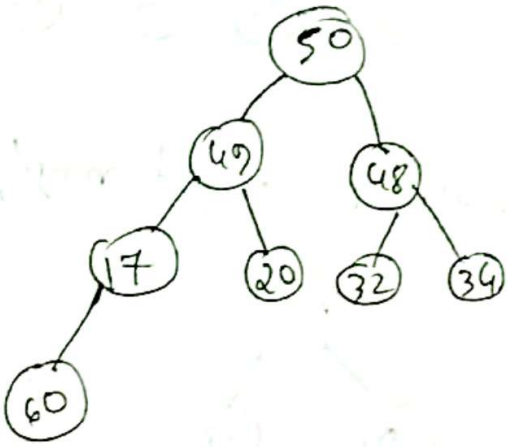
Max Heap

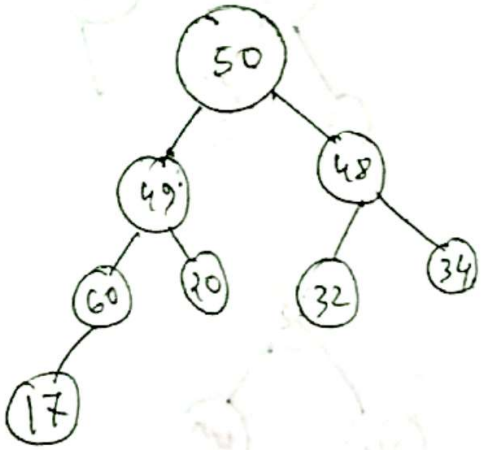


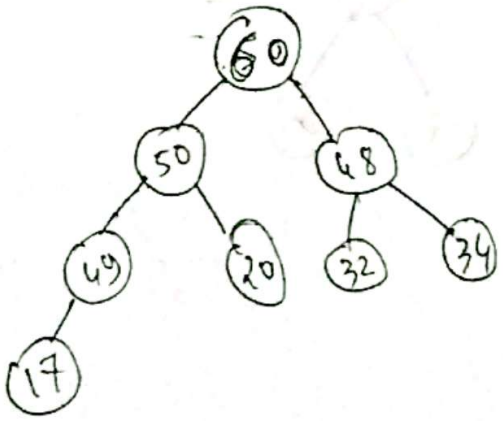
50	49	48	17	20	32	34
0	1	2	3	4	5	6

Insert 60:

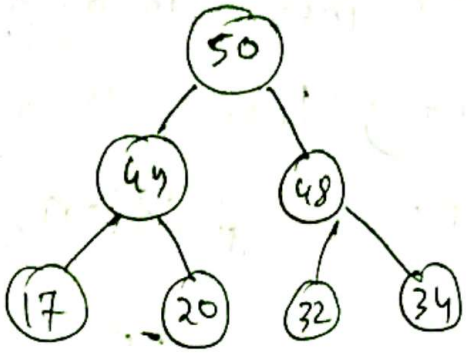
First insert into a position in complete binary tree; then take it to correct position.





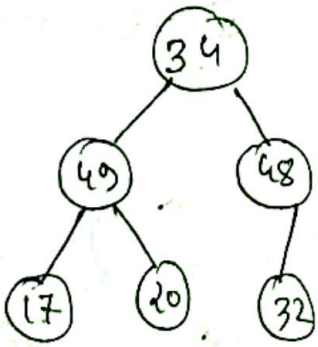


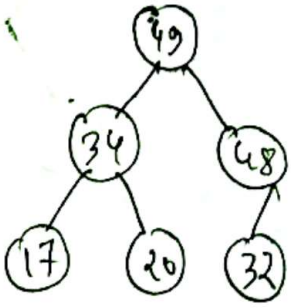
Extract Max:



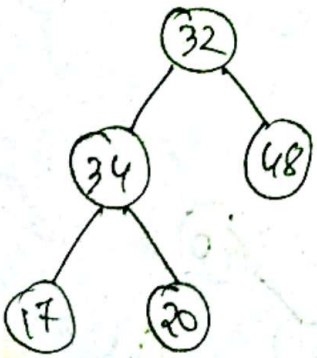
50
49
48
34
32
20
17

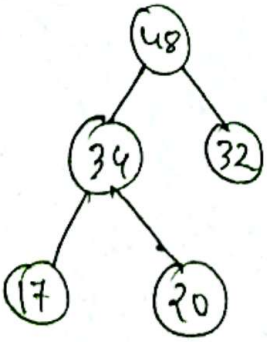
* swap root, with right most,

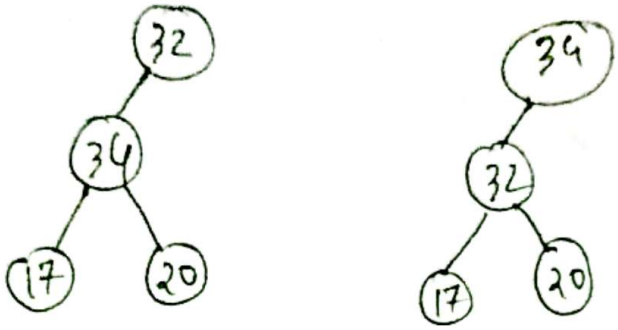


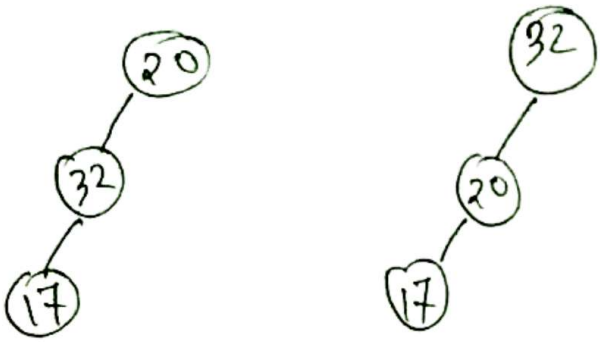


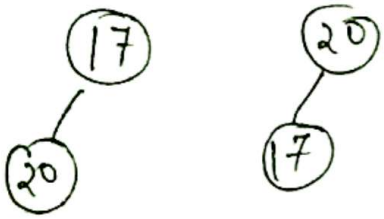
*



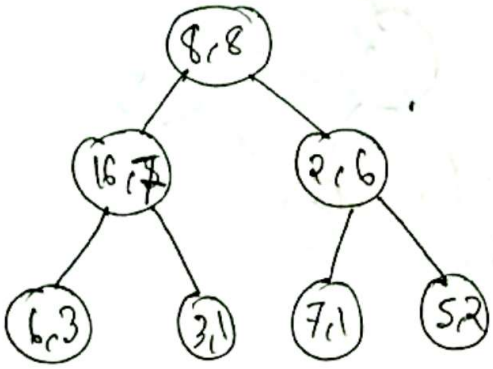



32
34
17
20
34
32
17
20

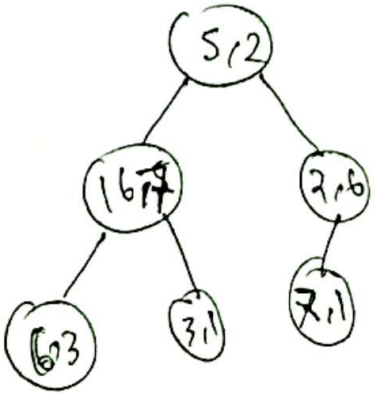

20
32
17
32
20
17

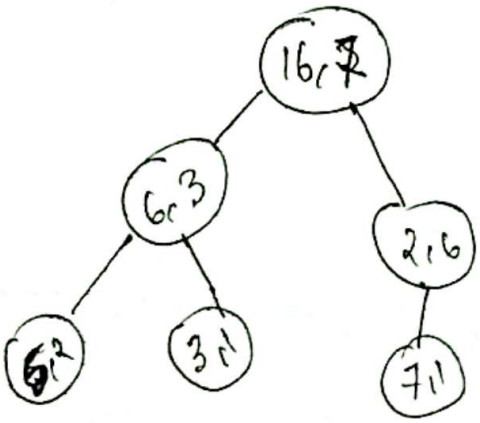

17
20
20
17

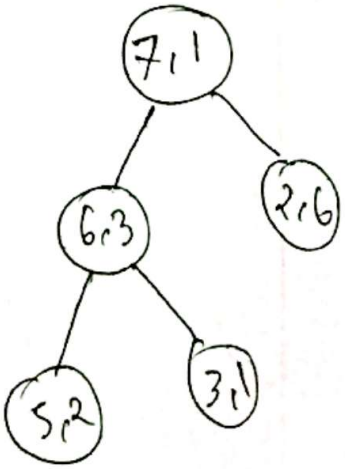
Priority Queue

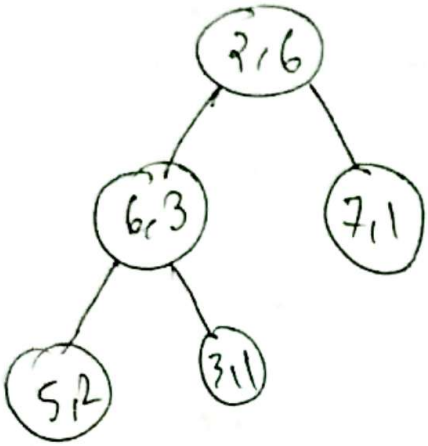


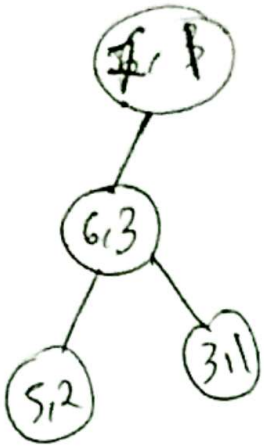
8, 8
16, 7
2, 6
6, 3
5, 2
7, 1
3, 1

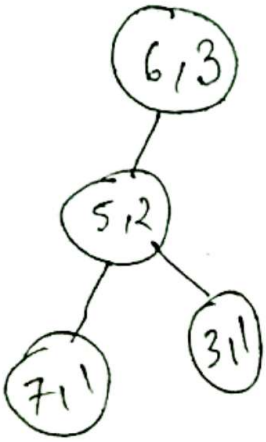







2,6
6,3
7,1
5,2
3,1


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


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

7,1
3,1

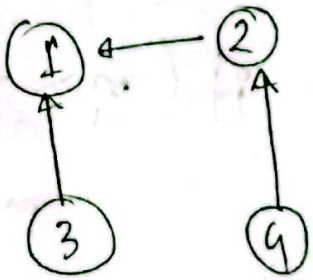
Disjoint Set

{1} {2} {3} {4}

union (1, 3)

union (2, 4)

union (1, 2)



- parent array,
  rank array,
  size array,

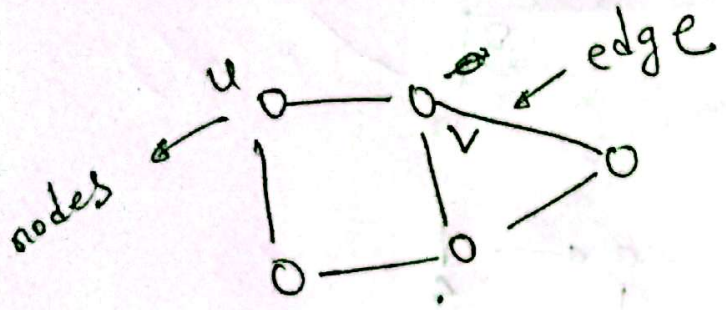
Union by rank:

```
union_by_rank (u, v):
        u = find_root(u)
        v = find_root(v)

        r1 = rank(u)
        r2 = rank(v)

if (u == v) return; (same set)

if (r1 > r2):
        parent[v] = u;

else if (r1 < r2):
        parent[u] = v;

else :
        parent[v] = u
        rank[u]++;
```

```
union_by_size (u,v);
    u = find_root (u)
    v = find_root (v)
    if (u==v) return;
    if (size[u] > size[v]):
        parent[v] = u;
        size[u] += size[v];

    else:
        parent[u] = v;
        size[v] += size[u];
```

CamScanner

Graph



$u - v$

↓

undirected

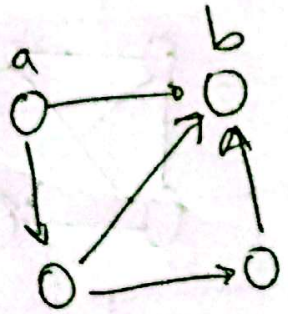
$u \rightarrow v$

↓

directed

degree (u) = 2

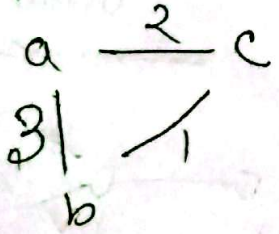
degree (v) = 3



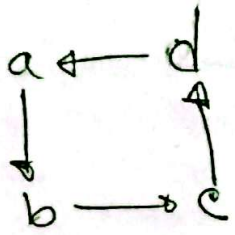
Indegree (b) = 3

outdegree (b) = 0

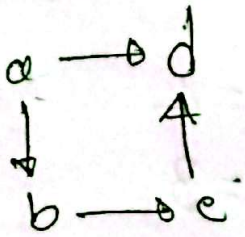
weighted graph:



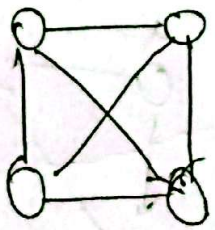
cyclic graph:



acyclic graph:



complete graph:

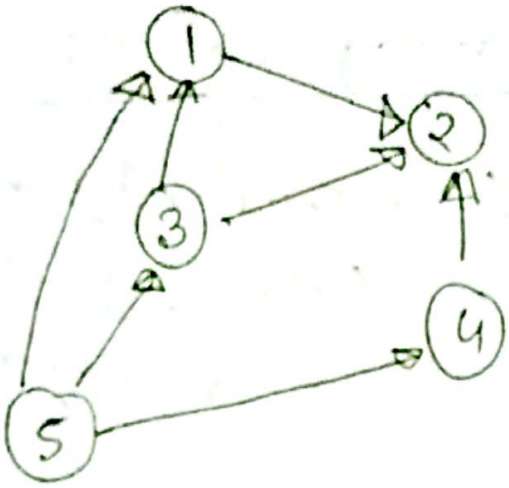


close to complete → densed graph

far from complete → sparse graph

densed graph → adjacency matrix

sparse graph → adjacency list



adjacency matrix:

	1	2	3	4	5
1	1	1	0	0	0
2	0	1	0	0	0
3	1	1	1	0	0
4	0	1	0	1	0
5	1	0	1	1	1

adjacency list:

[1] → 2

[2]

[3] → [1] → [2]

[4] → [2]

[5] → [1] → [3] → [4]

adjacency matrix:

Space complexity → $O(n^2)$
(storing)

n → vertices

Time complexity: $O(1)$
(Find)

adjaceney list:

$n \rightarrow$ vertices

$E \rightarrow$ edge

space complexity:

$O(n+2E) \rightarrow$ undirected

$O(n+E) \rightarrow$ directed

Time complexity: $O(n)$

For, dense graph,

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

$$E = \frac{n^2-1}{2}$$

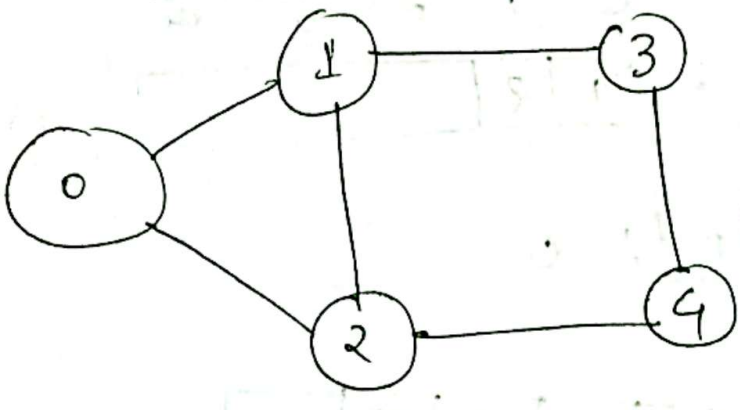
$$E \approx n^2$$

$$O(n+n^2) \approx O(n^2)$$

so, that's why densed → matrix

but → sparse → List

BFS



visited:

F	F	F	F	F
0	1	2	3	4

queue :

output :

step-1 :

T	F	F	F	F
0	1	2	3	4

queue

0	

output :

0	

Step-2:

T	T	T		
0	1	2	3	4

queue:

1	2

output: 0

Step3:

T	T	T	T	
0	1	2	3	4

queue:

2	3

output: 0 1

Step-4:

T	T	T	T	T
0	1	2	3	4

queue:

3	4

output: 0 1 2

then:

output: 0 1 2 3 4

Huffman Coding

— size compression technique

Message → BCCABBDDAECCBBAEDDCC

Length → 20

ASCII → 8 bits

A — 65 — 01000001
B — 66 — 01000010
C — 67 — 01000011
D — 68 — 01000100
E — 69 — 01000101

Total costs here 8 X 20 = 160 bits

CamScanner

Fixed size Code Encoding:

character	count/frequency	code
A	3	000
B	5	001
C	6	010
D	4	011
E	2	100
	20	

costs: $20 \times 3 = 60$ bits

we have to send table also,

character $\rightarrow 5 \times 8 = 40$ bits

code $\rightarrow 5 \times 3 = 15$ bits

Total $= 60 + 40 + 15$

$= 115$ bits

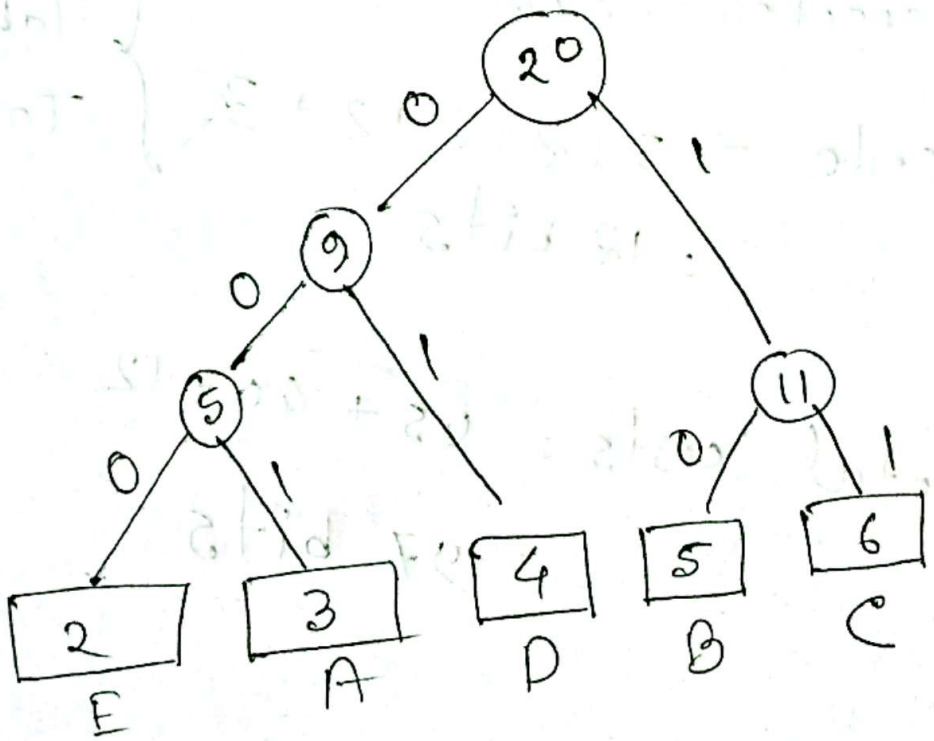
Variable size encoding:

Huffman code follows optimum merge pattern.

Alphabet should be arranged in increasing order based on count.

merge smallest two.

char	count	code
A	3	001
B	5	10
C	6	11
D	4	01
E	2	000
	20	



Costs:

A → 3×3 = 9

B → 5×2 = 10

C → 6×2 = 12

D → 4×2 = 8

E → 2×3 = 6

45 bits

Message = 45 bits

character = 5×8 = 40 bits

code = 3+2+2+2+3 = 12 bits

} Table/Tree

Total costs = 45 + 40 + 12

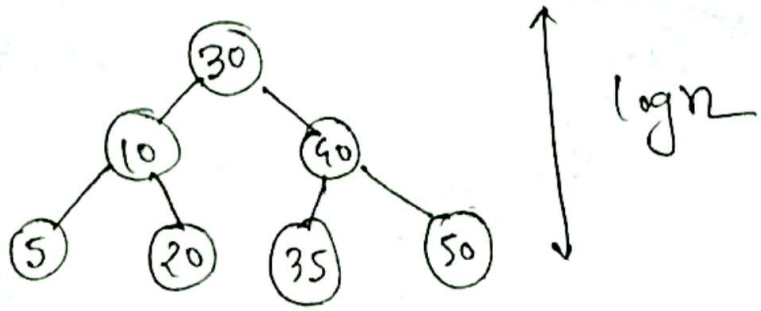
= 97 bits

In BST for searching a key maximum comparison needed ~~for BST~~ is height of BST.

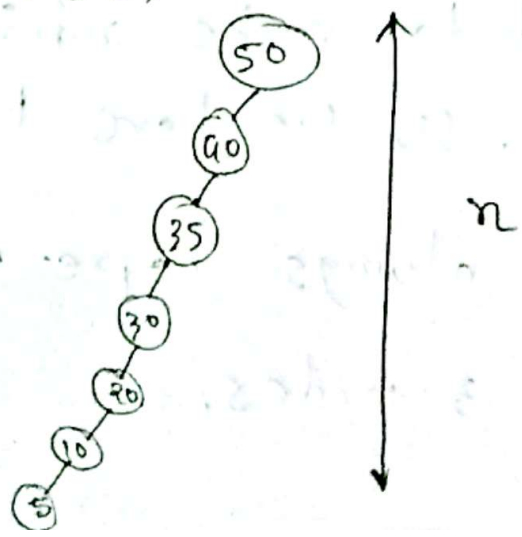
<u>height:</u> min → $\log n$

max → n

<u>keys:</u> 30, 40, 10, 50, 20, 5, 35



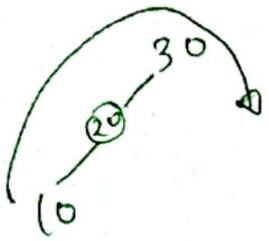
<u>keys:</u> 50, 40, 35, 30, 20, 10, 5



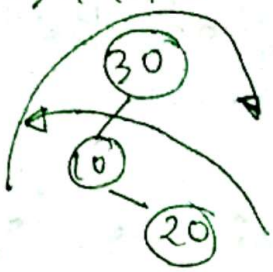
keys → 30, 20, 10

No. of ways to make tree = 3! = 6

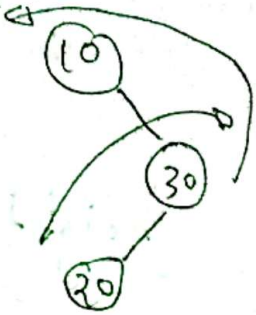
30, 20, 10



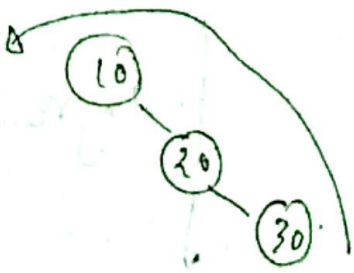
30, 10, 20



10, 30, 20

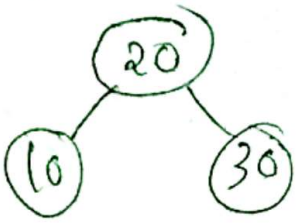


10, 20, 30



20, 10, 30

20, 30, 10



we want to make minimum ~~to~~ height. so, we have to use rotation

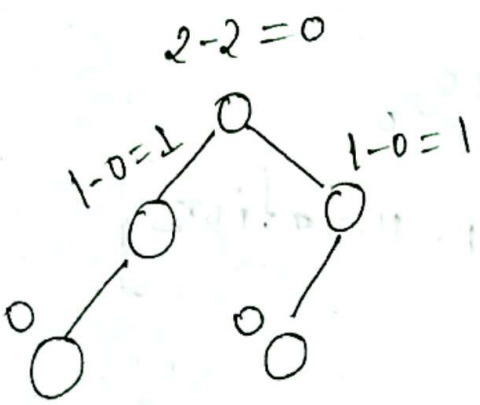
rotation always ~~have~~ will be between 3 nodes.

AVL Trees

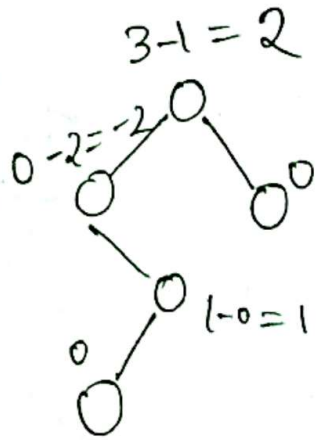
balance factor = height of left subtree − height of right subtree

$$bf = h_l - h_r = \{-1, 0, 1\}$$

$$|bf| = |h_l - h_r| \leq 1$$

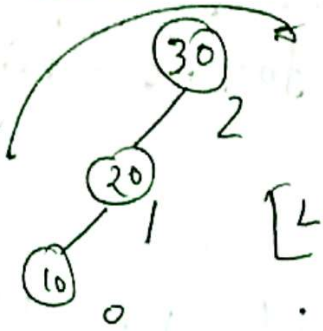


balance



Initially:

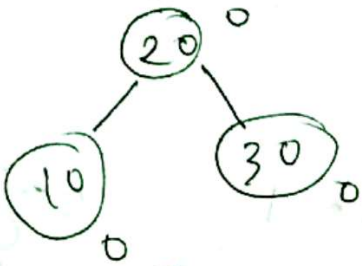
Insert 10



[LL- imbalance]

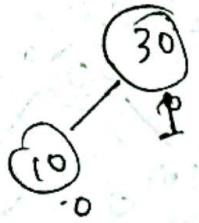
after - Rotation

+ve → heavy on Left

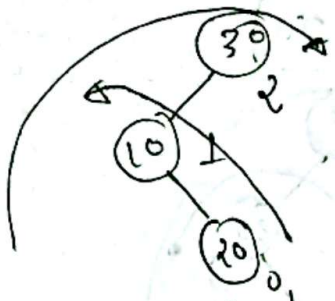


[LL- rotation]

Initially:

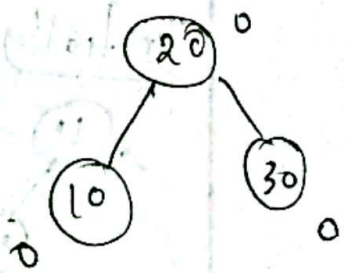


Insert 20



LR-imbalance.

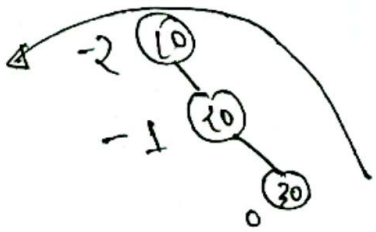
After - Rotation



2- step rotation
(Double rotation)
[LR-rotation]

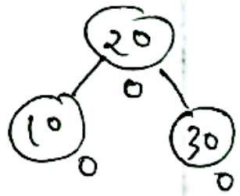
Initially:

Insert 30



RR- imbalance

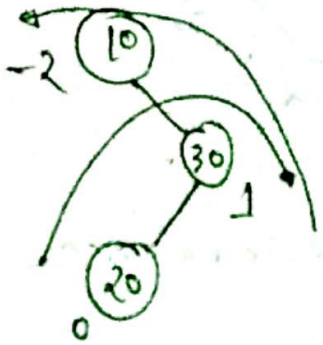
After - rotation



RR-rotation

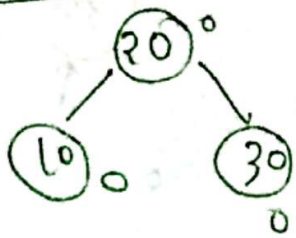
Initially:

Insert 20



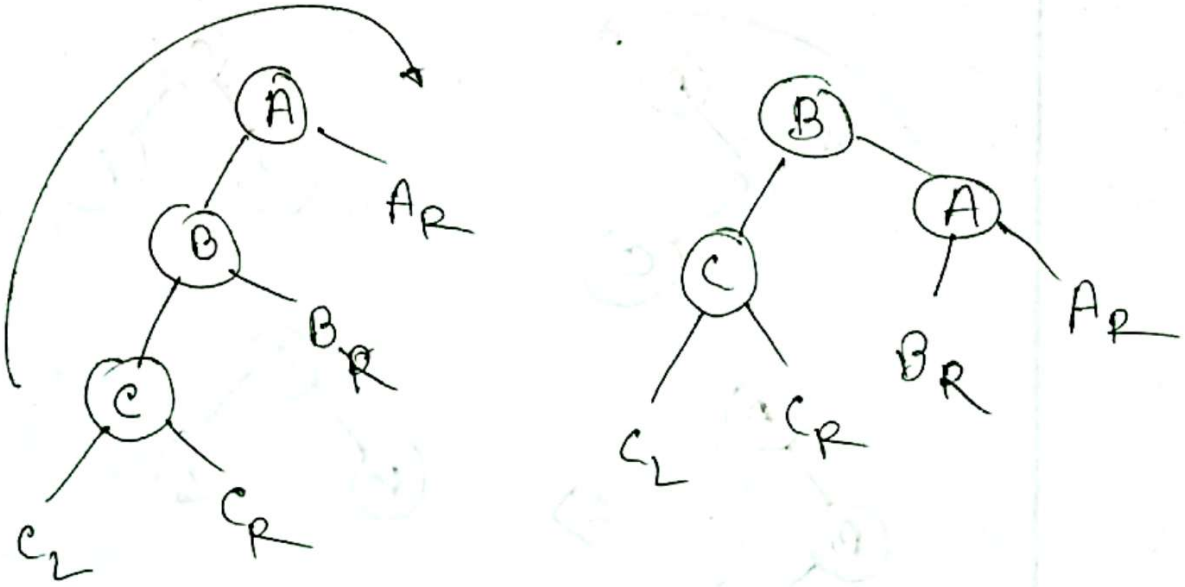
RL- imbalance

After rotation

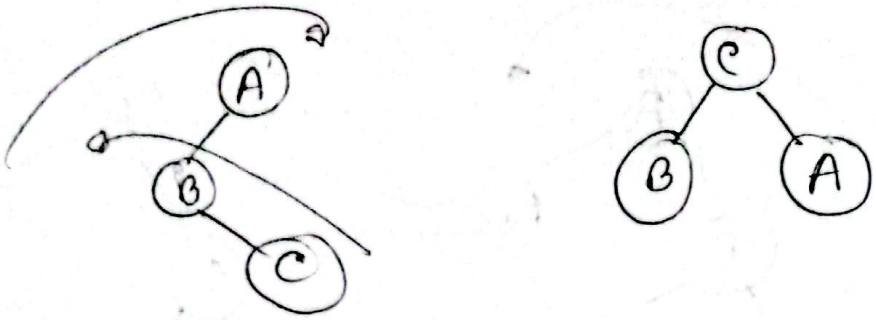


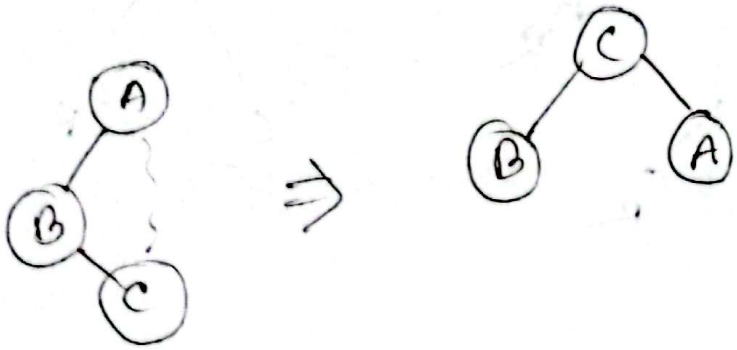
2- step rotation
Double rotation
(RL- rotation)

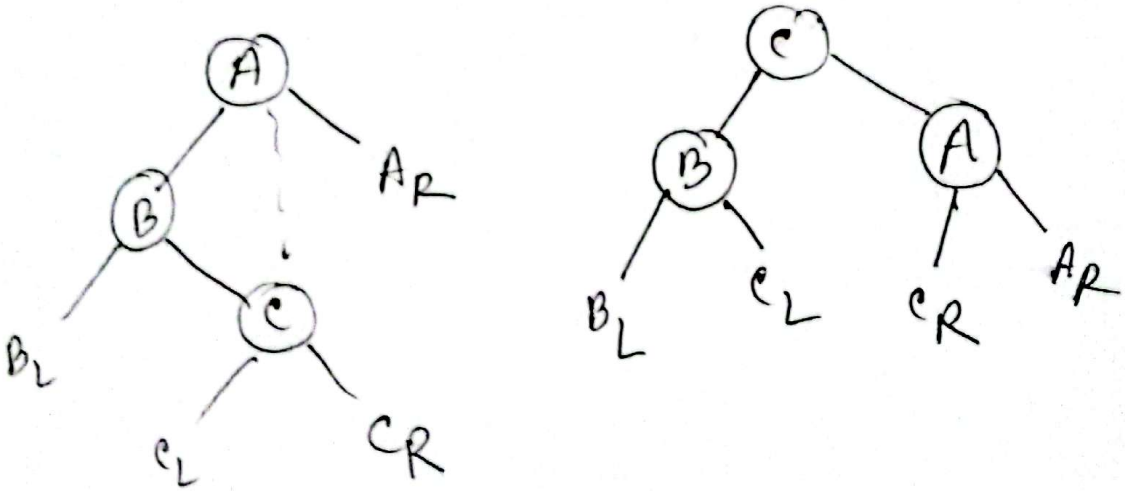
LL- Rotation



LR- rotation





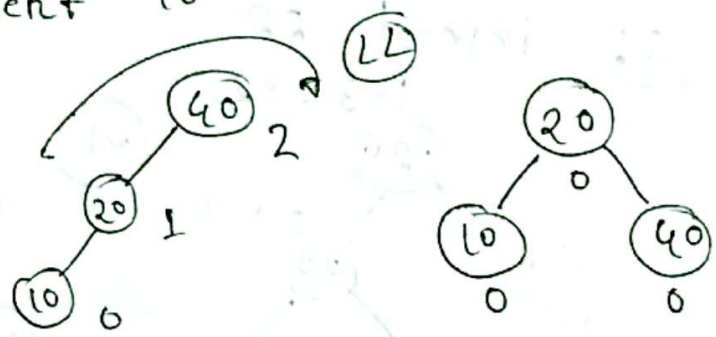


Create AVL Tree

keys → 40, 20, 10, 25, 30, 22, 50

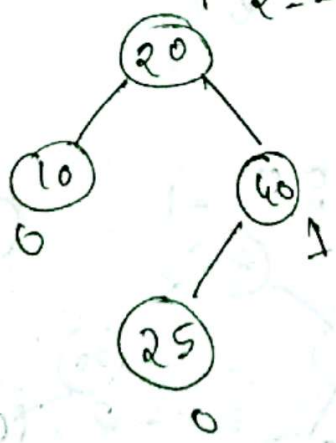
Step-1:

Step-2: Insert 10



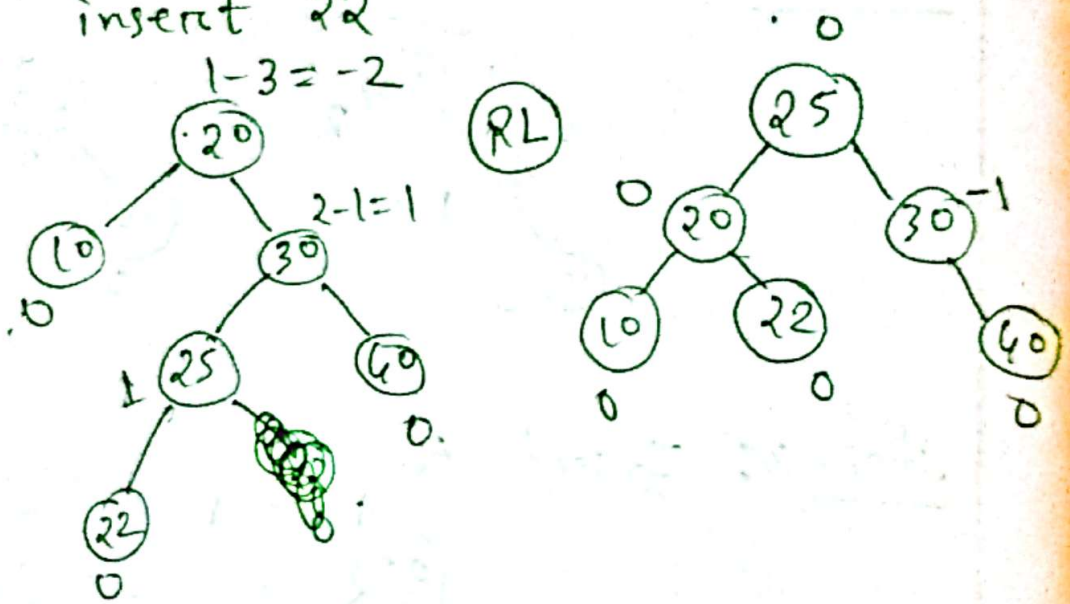
Step-3: Insert 25

1 - 2 = -1

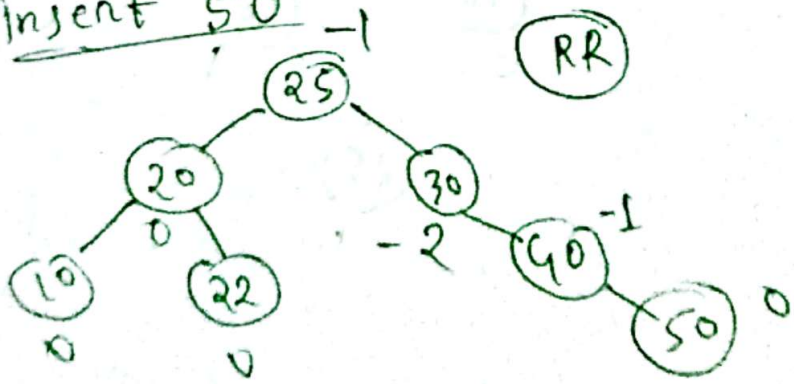


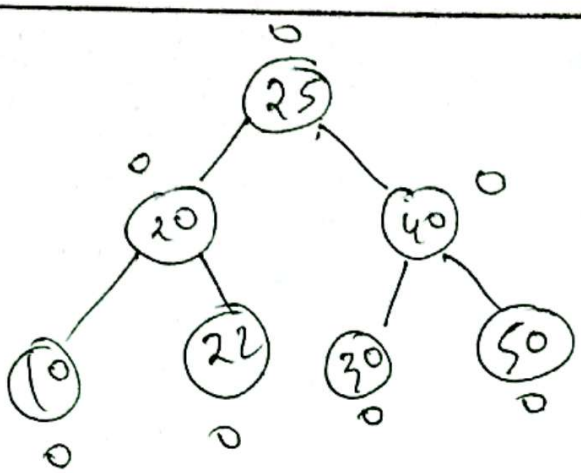
Step-4: insert 30

Step-5: insert 22



Step-6: insert 50





CamScanner