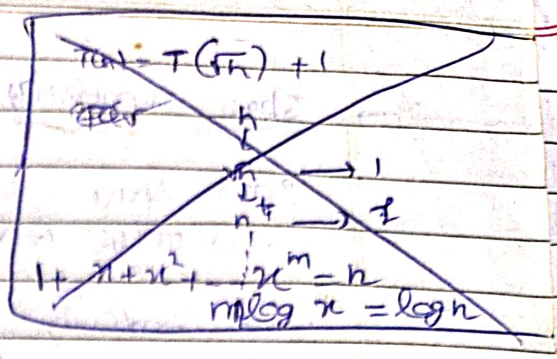


⇒ Analysis of Algorithms:

An algo. Depends on (i) Input Size
(ii) Input order

So, we use Asymptotic Analysis

↓

It evaluates the performance (time & space) of a algo. in terms of input size.
How does the time taken increases by input size increase.

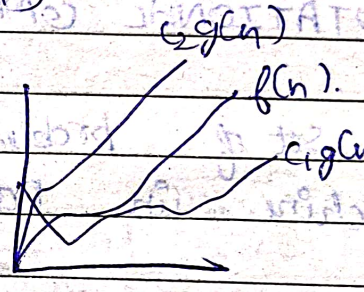
Three cases

(i) Best case → Lower Bound.
(ii) Worst case → Upper Bound.
(iii) Average case → All possible inputs & calculate computing time divided by sum of total no. of inputs.

$$= \frac{\sum_{i=1}^{n+1} \theta(i)}{(n+1)}$$ all possible inputs

Three Notations:

(i) θ Notation Both
(ii) O Notation $c_2 g(n)$
(iii) Ω Notation $c_1 g(n)$
(iv) Little o strict upper bound
(v) little omega strict lower bound



→ Amortized Analysis:

Used for algorithms where an occasional operation is very slow, but most of other operations are faster.

e.g. Dynamic Table (or Arrays).

Amortized cost for n operations $= \frac{(1+1+\cdots+n\text{ terms}) + (1+2+4+\cdots+2^{\log_2(n-1)})}{n}$

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

≤ 3

→ Space Complexity & Auxiliary Space :

Extra space or temporary space used by an algorithm. is called Auxiliary space.

Space Complexity is total space taken by the algorithm w.r.t. input size. It includes both auxiliary space & space used by input. e.g. Insertion Sort have O(n) space complexity but take O(1) auxiliary space.

→ Pseudo-polynomial :

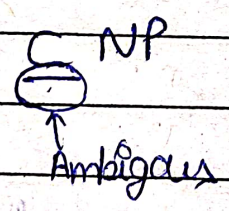
An algorithm whose worst case time complexity depends on numeric value of input (not no. of inputs) is called Pseudo-polynomial algorithm.

e.g. algorithm depending on max. value of inputs.

⇒ COMPUTATIONAL COMPLEXITY:

P → Set of problems solvable by deterministic Turing machine in Polynomial Time.

NP → Set of decision problems solvable by Non-Deterministic Turing Machine in Polynomial Time.



"Lucky" Algorithm which always makes a right choice.

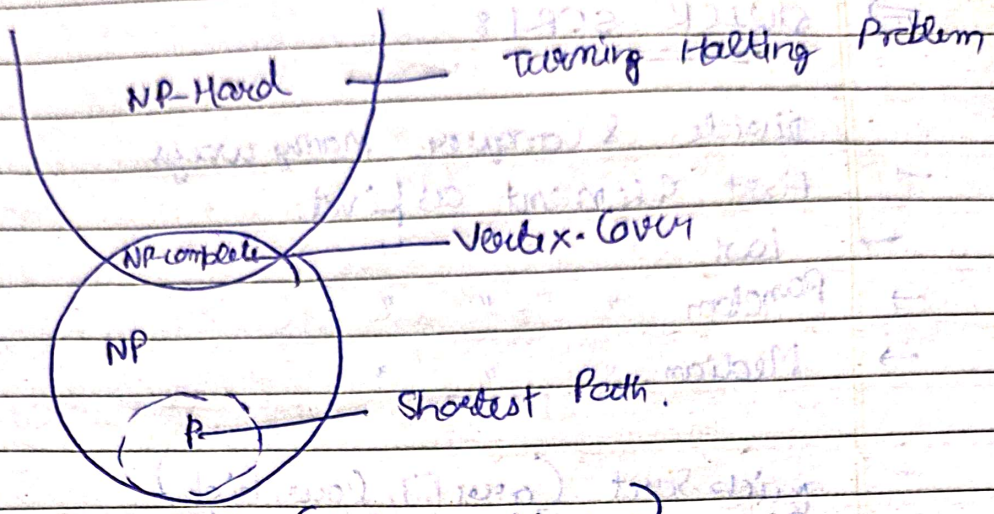
NP Complete : A decision problem L is NP-Complete if :

→ L is in NP.

→ Every problem in NP is reducible to L in polynomial time.

NP-Hard

So,



$P \neq NP$ (only considering)

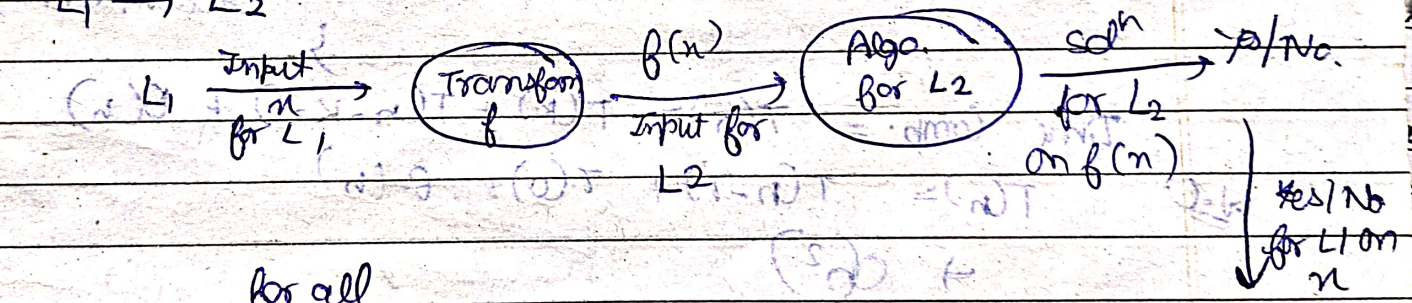
Note: Decision problems Vs Optimization Problems:

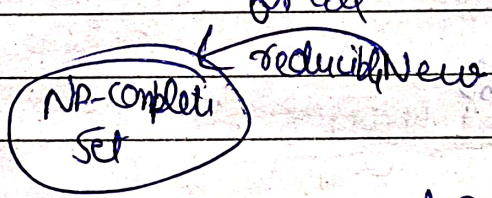
Note 1: Decision $\approx$ Optimization.

Note 2: Decision is easier than Optimization.

→ **REDUCTION:**

$L_1 \rightarrow L_2$





Reverse also apply.

1st NP-Complete problem was SAT (Boolean Satisfiability Problem

Selection, Quick & Heap are unstable.

⇒ QUICK SORT:

Divide & Conquer, Many ways

→ First element as pivot
→ Last " " "
→ Random " " "
→ Median " " "

```
quickSort (arr[], low, high)
{
  if (low < high) {
    pi = partition (arr, low, high)
    quickSort (arr, low, pi-1)
    quickSort (arr, pi+1, high)
  }
}
```

```
Partition (arr[], lg h)
{ i = low - 1;
  pivot = arr[high];
  for (j = low to high-1)
  {
    if (arr[j] <= pivot)
      i++
      swap (a[i], a[j])
  }
  swap (a[i+1], a[high])
  return (i+1);
}
```

Time comp. = $T(n) = T(K) + T(n-K-1) + \Theta(n)$

W.C. $T(n) = T(n-1) + T(0) + \Theta(n)$

$\Rightarrow \Theta(n^2)$

B.C. $T(n) = 2T(\frac{n}{2}) + \Theta(n)$

$\Rightarrow O(n \log n)$

A.C. $T(n) = \overbrace{N+1}^{\text{swaps}} + \frac{\sum_{1 \le K \le N} (C_{K-1} + C_{N-K})}{N}$

$$\Rightarrow C_N = N+1 + \frac{2}{N} \sum_{1 \le K \le N} C_{K-1}$$

$$NC_N = N(N+1) + 2 \sum_{K=1}^{N} C_{K-1} \quad -(1)$$

$$(N-1)C_{N-1} = (N-1)N \; 2 \sum_{K=1}^{N-1} C_{K-1} \quad -(2)$$

Subtracting (2) from (1)

$$NC_N = 2N + 2C_{N-1} + NC_{N+1} - C_{N-1}$$

$$NC_N = (N+1)C_{N-1} + 2N$$

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

$$= \frac{C_{N-2}}{N-1} + \frac{2}{N} + \frac{2}{N+1}$$

$$\frac{C_N}{N+1} = \frac{C_2}{3} + \sum_{K=3}^{N} \frac{2}{K+1}$$

$$\frac{C_N}{(N+1)} \approx \int_3^N \frac{2}{K+1} \, dK$$

$$\boxed{C_N \approx (N+1) \ln N}$$

⇒ 3-WAY QUICK SORT / DUTCH NATIONAL FLAG

```
while (mid <= high)
{
    if (a[mid] < pivot)   swap(&a[low++], &a[mid++]);
    else if (a[mid] == pivot)   mid++;
    else   swap(&a[mid], &a[high--]);
}
```

return {low-1, mid}
high

⇒ Merge Sort for linked list:

⇒ Merge sort :

```
mergeSort(0, n-1) {
  if (low < high) {
    int mid = (l+h)/2
    mergeSort(l, mid)
    mergeSort(mid+1, h)
    merge(l, mid, h)
  }
}
```

```
merge {
  arr1 = l, mid
  arr2 = mid+1, high
  merge them
}
```

nlgn for al

⇒ Insertion sort:

$O(n^2)$ → Worst
$O(n)$ → Best
$O(n^2)$ → Average

```
for (i=0; i<n; i++) {
    int num = a[i];
    int j = i-1;
    while (j>=0 && a[j+1] > a[j]) {
        swap (a[j+1], a[j]);
        j--;
    }
}
```

⇒ Selection sort:
Select min.

$O(n^2)$ → Worst
$O(n^2)$ → Best
$O(n^2)$ → Average

```
for (i=0; i<n; i++) {
    int mini = a[i]
    for (j=i+1; j<n; j++) {
        if (a[j] < mini)
            mini = a[j];
    }
    swap (mini_ind, a[i]);
}
```

⇒ Bubble Sort:

$O(n^2)$ → Worst
$O(n^2)$ → Best
$O(n^2)$ → Average

```
for (i=0; i<n; i++) {
    for (i=1; i<n; i++) {
        if (a[i-1] > a[i])
            swap
    }
}
```

⇒ Heap DS:

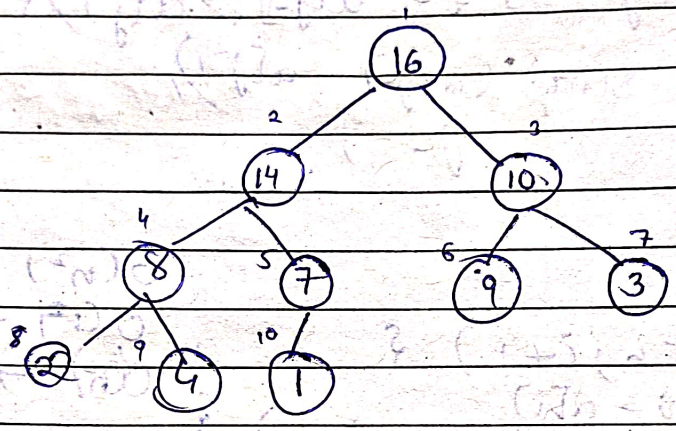
Tree-based DS in which tree is complete binary tree.

↓ Max-Heap, Min-Heap.

An array visualised ~~complete~~ nearly complete binary tree.

e.g.

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



root → first element (i=1)

Parent (i) = $i/2$

left (i) = $2i$

right (i) = $2i+1$

for min Heap

```
void heapify(int i) {
    int l = left(i)
    int r = right(i)
    int small = i
    if (l<n && a[i] > a[l])
        small = l
    if (r<n && a[small] > a[r])
        small = r
    if (small != i) {
        swap(a[i], a[small]);
        heapify(small);
    }
}
```

Other operations like insert, delete can be done by a while loop only.

Heap Sort

Heapify (make a heap)

for each N

heapify & extract

⇒ Time Complexity of heap build:

For a height h with n nodes in Binary tree atmost $= \left\lceil \frac{n}{2^{h+1}} \right\rceil$

So, $T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil * O(h)$

$= O\left(n * \sum_{h=0}^{\log n} \frac{h}{2^h} \right)$

$= O\left(n * \sum_{h=0}^{\infty} \frac{h}{2^h} \right)$

$= O(n * 2)$

$= O(n)$

$\sum_{n=0}^{\infty} x^n = \frac{1}{1-x}, \quad x < 1$

$\sum_{n=0}^{\infty} n x^{n-1} = \frac{1}{(1-x)^2}$

$\sum_{n=0}^{\infty} n x^n = \frac{x}{(1-x)^2}$

replacing x by $\frac{1}{2}$ & n by h

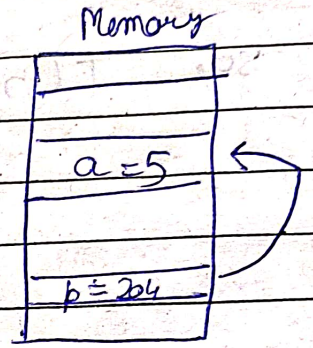
$\sum_{h=0}^{\infty} h \frac{1}{2^h} = \frac{\frac{1}{2}}{(\frac{1}{2})^2} = \frac{1}{\frac{1}{2}} = 2$

⇒ Linked List: Before studying any of it, let's study pointers:

POINTERS (size of pointer depends on architecture
8byte → 64 bit
2byte → 16 bit

⇒ Variables that store address of other variables.

```
int a;
int *p;
char c; char *pc;
p = &a
*p = 8
a // 8.
```



⇒ Pointer Arithmetic

```
int *p = &a   // 2002
(p+1)         // 2006.
```

↓
Now, it gets "dangerous"

(p-q) Also possible

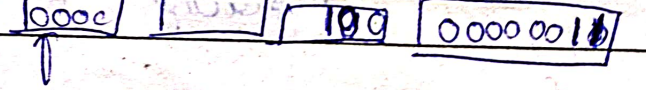
(p-1) 1998

→ Why we need strongly typed pointers?

Ans Due to dereferencing

Now, let's see a mind blowing trick:

```
int a = 1027
int *p = &a;
```

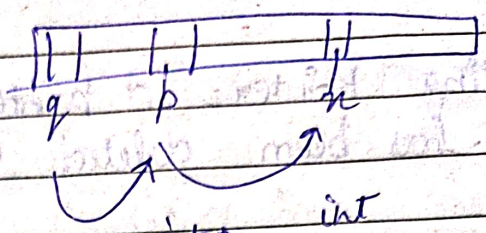



```
p    // 2002
*p   // 1027
char *p0 = (char*) p;
p0   // 2002
*p0  // 3
```

DATE

⇒ Pointer to Pointer:

```
int x = 5
int *p = &x
int **q;
q = &p;
```



⇒ Pointer & String: (Char array)

```
char *s = "Abc";
char *s = (char*) "Abc";
```

((++))

⇒ Pointers & Arrays:

A = &A[0]

Address — &A[i] or (A+i)

Value — A[i] or *(A+i)

Array as Function

↓

int A[] ⟺ int *A

⇒ 2-D array in fun^n

(int (*a)[n])

*((a + i*m) + h)

⇒ Pointer & Multi-dimensional Array:

B[i][j] = *(B[i] + j) = *(*(B+i) + j)

(Internal Pointer)

⇒ Reference Variables in C++: (Alias)

```
int x = 10;
int &y = x
```

x: 10

Reference variable (Internal pointer)

* Must be initialized with variables.
* Alternative name for existing variable

⇒ Types of pointers

(i) Dangling Pointer: A pointer pointing to a memory location that has been deleted (or freed)

(ii) Void Pointer: A pointer pointing to some data location in storage, which doesn't have any specific type.

(iii) Null Pointer: A pointer pointing to nothing

(iv) Wild Pointer: A pointer which has not been initialized to anything (not even NULL).

Beware of Wild Pointer.

⇒ Dynamic Memory Allocation:

Allocated on Heap just like global or static variables.

```
int *p = new int
int *p = new int[10];
    delete p;
    delete [] p;
```

Back to Linked List:

(i) Floyd's Cycle-Finding Algorithm.

Move one pointer by one then by two, if they are same at a moment return true.

* Try everything via two pointers, three pointers done.
You are pro !

→ Detect loop & Remove (detect loop + getLength)
→ Reverse a linked list
→ Swap two nodes

⇒ DMA

C

int *p = (int *) malloc (size of (required))
Memory allocation
returns void pointer

(int *) calloc (No. of Blocks, byte Size)
contiguous allocation

(int*) realloc (p , new Size) ;
address

free (p) ;

C++

new int ;
new int[10];

delete p;
delete [] p;

BINARY TREE

Hierarchical data structures

e.g. File System, provides moderate Access, provides moderate Insertion/deletion.

A tree with almost 2 children is called BT.

Properties

Note 1: The maximum no. of nodes at level 'l' of binary tree is 2^{l-1}.

Note 2: Maximum no of nodes in BT of height 'h' is $2^h - 1$

Note 3: Minimum No. of level with N nodes = $\log_2(N+1)$ with 1 as convention

Note 4: No. of levels with L leaves = $\log_2 L + 1$

Note 5: $L = T + 1$

L: No. of leaf nodes; T: No. of internal nodes with 2 children

Types:
- Full → (0 or 2 children)
- Complete → (As left as possible)
- Perfect (leaf at same level) → All with 2 children
- Balanced (log N)
- Degenerate (like LL)

Handshaking Lemma

No. of vertices with odd degree is even.

⇒ No. of binary trees with n nodes

Unlabeled = n^{th} catalan number

$= \sum_{i=1}^{N} C(n-i)\,C(i-1)$ = No. of BSTs

labeled = n^{th} catalan number $\times$ $(n)!$

✱ Also n^{th} catalan number = $\frac{(2n)!}{(n+1)!\,n!}$

(Inorder/Preorder/Postorder) (Level order)

DFS vs BFS in binary tree

	Extra space	Extra space
	Height of BT	width of BT
Normal	$O(h)$	$O(w)$
Worst	$O(n)$	$O(n)$

Preorder	Root	L	R
Inorder	L	Root	R
Postorder	L	R	Root

Iterative

right left

point, ,

If tmp → right push, and go to leftest

Push tmp → right & tmp.

⇒ Binary Search Tree:

* Inorder traversals produce sorted output
* We can create BST from preorder, postorder or inorder

Ops
- Range Search
- Nearest Neighbours
- Insert
- Delete

⇒ Delete in BST

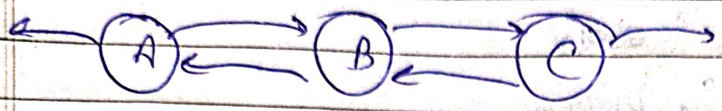
→ If leaf delete
→ If one child, assume it was never there
→ If two child, swap with its inorder successor and delete current one

⇒ Check if B.T. is BST:

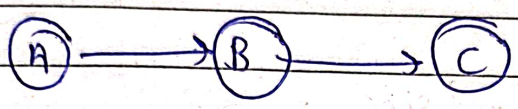
Given each node a range $(-\infty, \infty)$

$(-\infty, val)$ (val, ∞)

⇒ Doubly linked list



with single pointer



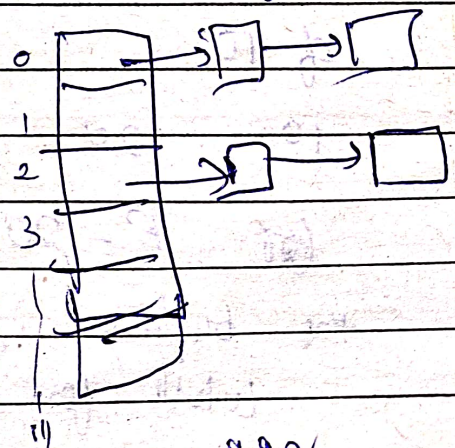
A next contains XOR of NULL & address of B
B next contains " " address of A & " " C.

→ Hashing:
Array, LL, Heaps, BSTs are not giving best search Insert & delete.

Hash gives (or Direct access table)
↓ Requires Huge Size

converts into
smaller index (using hash func^n)
- efficiently computable
- uniformly distribute keys.

Collisions may occur
- Use separate chaining
- Open addressing



(1) Separate chaining

1. Linear probing

index = hash % tableSize
index = (hash + 1) % tableSize
                  |
                  |

2. Quadratic Probing

index = hash % tableSize
index = (index + 1^2) % tableSize
                  |
                  | $+2^2$
                  |

3. Double Hashing

index = hash % tableSize
(hash + 1 * index/4) % tableSize
                  |
                  2R
                  |

4. Random Probing:

Note: Prime no. in modulus

→ B/C if data is some pattern based e.g multiple of 10

10 20 30 40 50

for P = 20

the buckets will be 10 & 0 only

while for P=7 3, 6, 2, 5, 1

1100 1010 1001 1000

⇒ BIT Manipulation:

$n \& (n-1)$ = removes rightmost 1.

i) Pairs of 2:

$n \;\&\&\; !(n \& (n-1))$

It needs to partig

ii) Count No. of ones:

```
while (n) {
    n = n & (n-1);
    cnt++;
}
```

iii) generate all subsets of A's

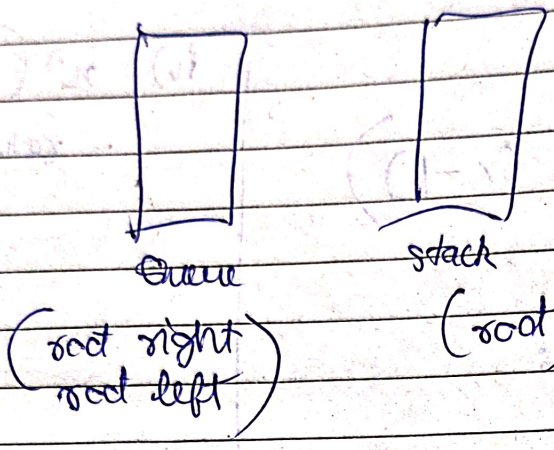
```
for (i=0; i<(2^N); i++) {
    for (j=0; j<N; j++) {
        if (i & (1<<j))
            print A[j];
    }
}
```

iv) $n \wedge (n \& (n-1))$

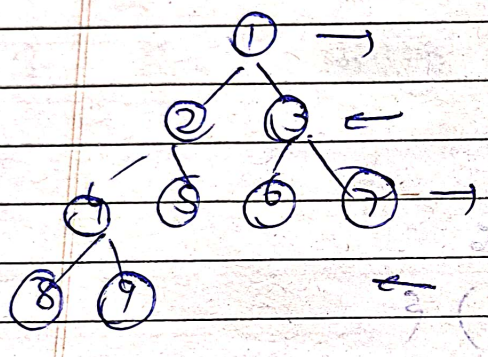
returns rightmost 1

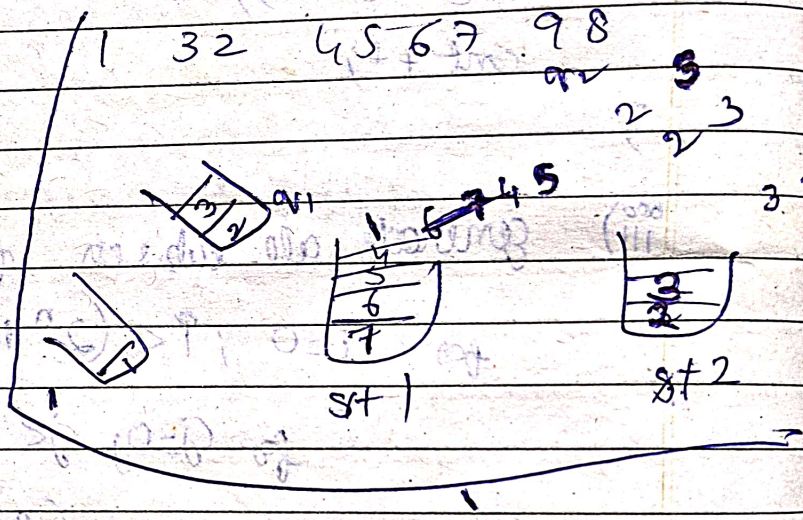
⇒ Advance Tree Algos:

I. Reverse Level Order:



II. Spiral level order





⇒ Deque.

III. LCA in BST:

if left < root → data & right > root → data

return root.

IV. LCA in BT:

if (curr == n && curr == y)
return curr;
curr == n || curr == y)
sol = 1

sol1 = curr → left
sol2 = curr → right

if either of 2 is true
return curr