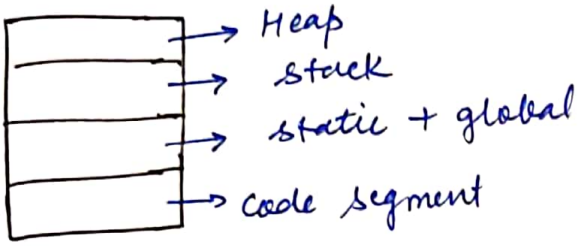
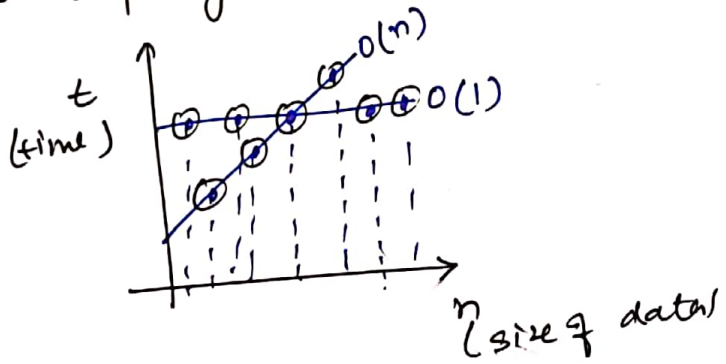
Data Structure And Algorithm

main memory : RAM



* Heap : allocates dynamic memory
  → request with help of pointer from stack to heap

* Time Complexity – time req to run ur algorithm



* Asymptotic Notations : O, Ω, Θ

* Big O : $0 \leq f(n) \leq c \cdot g(n)$ $\forall$ $n \geq n_0$
  ↑ size of input
  f" of time
  $f(n) \rightarrow O(gn)$

* Big Ω : $0 \leq c\,g(n) \leq f(n)$ $\forall$ $n \geq n_0$

* Big Θ : $c_2\,g(n) \leq f(n) \leq c_1\,g(n)$ $\forall$ $n \geq n_0$
  lower bond
  upper bound

* For Sorted Array * → Array Traversal

1) Best Case : $T_n = O(1)$ = const.
2) Worst Case : $T_n = O(n)$
3) Avg. or Expected Case : $T_n = O\left(\frac{\Sigma \text{ all pass.}}{\text{Total no of pass.}}\right) = O(n)$

* For Sorted Array * → Binary Search

1) Best : $O(1)$
2) Worst : $O(\log n)$ ↑ हमेशा आधा कर लेके

* Trick to Time Complexity Ques *

1) drop the non-dominant term (without -n wala)
2) drop the const. term
3) Break code into fragments/parts

eg: for any no. of loops (seprate loop) ⇒ $T_n = O(n)$

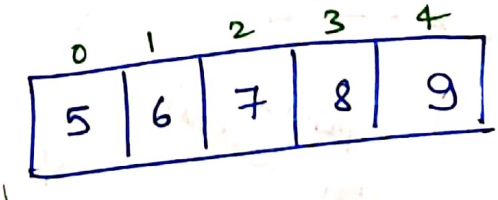
for nested loop → eg: for & ⇒ $T_n = O(n^2)$

```
for(){
    for(){
    }
}
```

Abstract data type : (MRF) + (operations)
↑
Minimal Requirement functionality

* Array ADT *

- Contigous block of memory
- Resize of it not poss.
- Traversal/accessing its elements
- Insert/delete → costly



operations

① Traversal :- visiting every element

② Insertation :

Case I : insert while maintaing the order → O(1) → best
→ O(n) → worst

Case II : don't have to maintain order → O(1)

③ Deletion :

Case I : maintain order → O(1) → Best
→ O(n) → worst

Case II : No need to keep order → O(1)

④ Searching :

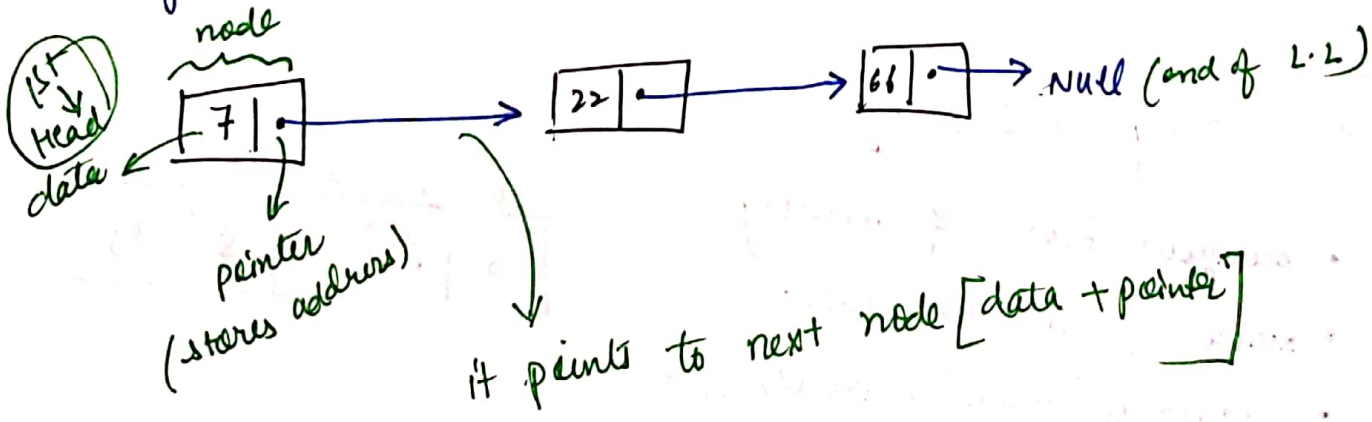
(i) Linear → works both → (sorted + Unsorted)
→ Array Traversal

(ii) Binary → only for sorted

$$mid = \left[\frac{low + high}{2}\right]$$

* Linked List *

- replacement to array
- Insert / delete is easy
- no constrain of contigous memory locations
- Only access of element is tough here
- 1st node represents : Head

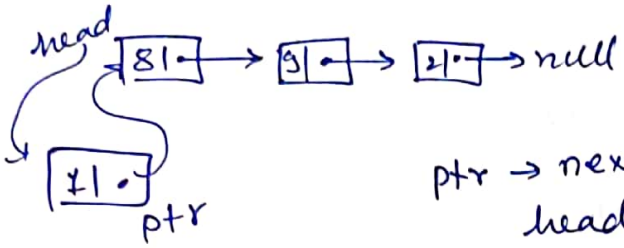


- Tn of Traversal is always = O(n)

Insertion in L.L

Case I : Insert at begging

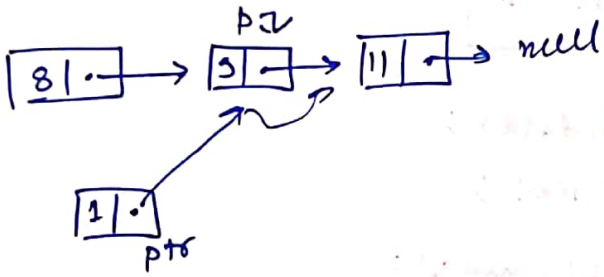
$T_n = O(1)$




```
ptr -> next = head ;
head = ptr ;
return head ;
```

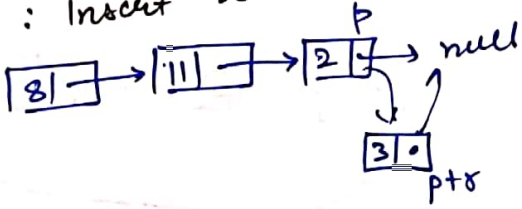
Case II : Insert in between

$T_n = O(1)$



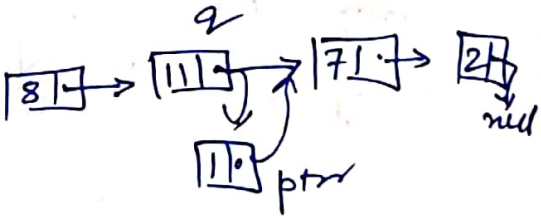

```
ptr -> next = p -> next ;
p -> next = ptr ;
```

Case III : Insert at end




```
p -> next = ptr ;
ptr -> next = null ;
```

Case IV : Insert after a given node




```
ptr -> next = q -> next
q -> next = ptr
```

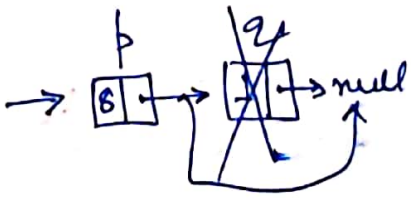
Deletion in L.L

Case I : delete 1st node

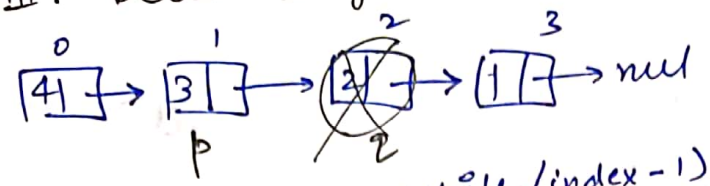
```
head = head -> next ;
free (head) ;
```

Case II : delete last node

```
p -> next = null
free (q) ;
```



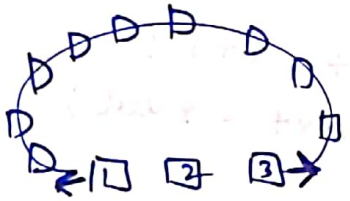
Case III: Delete a given node

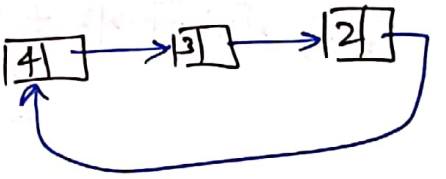


ind = 2 ; while (index - 1)
{
p = p → next;
}
p → next = q → next;
free(q);

* Circular Linked List *

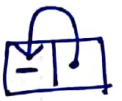
Traversal : do
{
print(p → data);
p = p → next;
}
while (p != head);





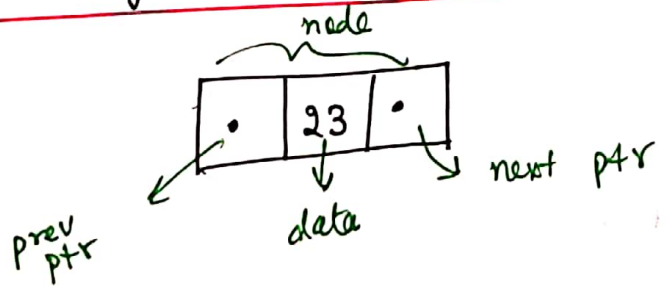
empty L.L : p → null

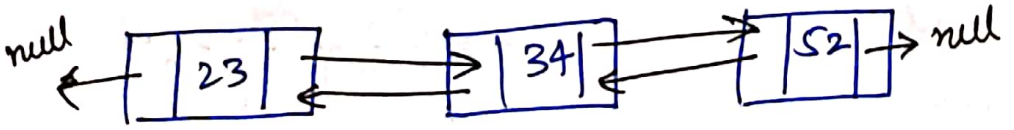
empty C.L :



* Doubly Linked List *

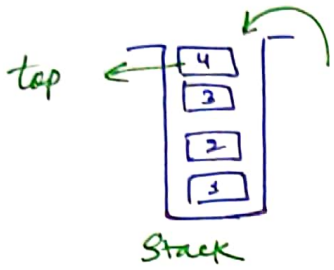
• Benefit of going (back + forward)
↓
But uses memory (extra space for (b + f))





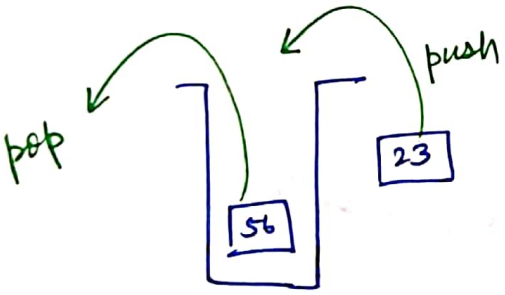
* Stack *

- Operation performed: LIFO or FILO
- implement using → Array, Linked List



Stack Using Array → $T_n = O(1)$ for all operations

- Collection of element (not Array)
- Top = -1 (no element)
- Top = 0 (1 element)
- Top = 1 (2 elemt)
- fixed array size creation
- Size of array ≠ Size of stack



size = arr.length ;

Empty Condition

top = -1
then stack → empty
else stack → full

Full condition

(top = size - 1)
then stack → full
else stack → empty

Push Op: → to insert element in stack

```
Push (value)
{
    if (isFull (s))
    {
        print ("Stack Overflow");
    }
```

```
else
{
    s->top++;
    s->arr[s->top] = value;
}//else
}//fn
```

Pop Operation → to remove element from stack

```
if (isEmpty(s))
{
    print("stack Underflow");
    return -1;
}
else
{
    int val = sp->arr[sp->top];
    s->top = sp->top - 1;
    return val;
}
```

Peak Operation → value at what position in stack

Position (i) : Array Index (Top - i + 1)

```
int peak(..)
{
    if((s->top - i + 1) < 0)
    {
        print("invalid position");
    }
    else
    {
        return s->arr[s->top - i + 1];
    }
}//fn
```

peak(5) → 5th position element return karo

Stack Top

top most value in stack

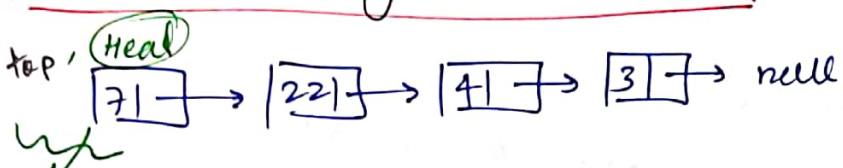
return s->arr[s->top];

Stack Bottom

bottom-most value in stack

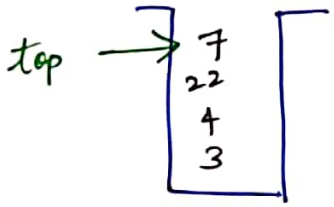
return s->arr[0];

Stack Using Linked List



it is used for push & pop

- Head is → top here
- Head is set as top → O(1) = tn for operation
  if 3 set as top → O(n) = tn for operation



Empty Condition

```
if (top == null)
  {
    return 1;
  }
else
  {
    return 0;
  }
```

Full Condition

- when can make it of unlimited size using L.L.

or

```
if(n == null)
  {
    return 1;
  }
else
  {
    return 0;
  }
```

Push Operation

```
Push(x)
{
  ~~node *n~~
  if (n == null)
    {
      print("overflow");
    }
  else
    {
      n->data = x;
      n->next = top;
      top = n;
    }
}// fn
```

Pop Operation

```
if (isEmpty)
  {
    print("overfl");
  }
else
  {
    struct node = top;
    top = top->next;
    int x = n->data;
    free(n);
    return x;
  }
```

Peak- Operation

```
Peak(int pos)
{
  struct node* ptr = top;
  for(i=0; (i< pos-1 && ptr!=null); i++)
    {
      ptr = ptr->next;
    }
  if(ptr!=null)
    {
      return ptr->data
    }
  else { return -1;
       }
}
```

stack-top

```
return top->data;
```

* Parenthesis Matching *

- brackets blanced or not → check
- 3 * 2 - (8 + 1)

0	1	2	3	4	5	6	7	8	9
3	*	2	-	(	8	+	1	)	\0

backslash 0 (zero)

* (? → push into stack (opening parenthesis)
*) ? → pop out of stack (closing Parenthesis)

- if anything beside parenthesis ; just ignore it

Condition for Balanced Expression

- While pop → stack shouldn't → underflow → unbalanced
- at the end of exp → stack must be empty
- just tells whether parenthesis is balanced or not
- doesn't give the idea about validity of expression

Algo:

- Use character array → memory → heap →

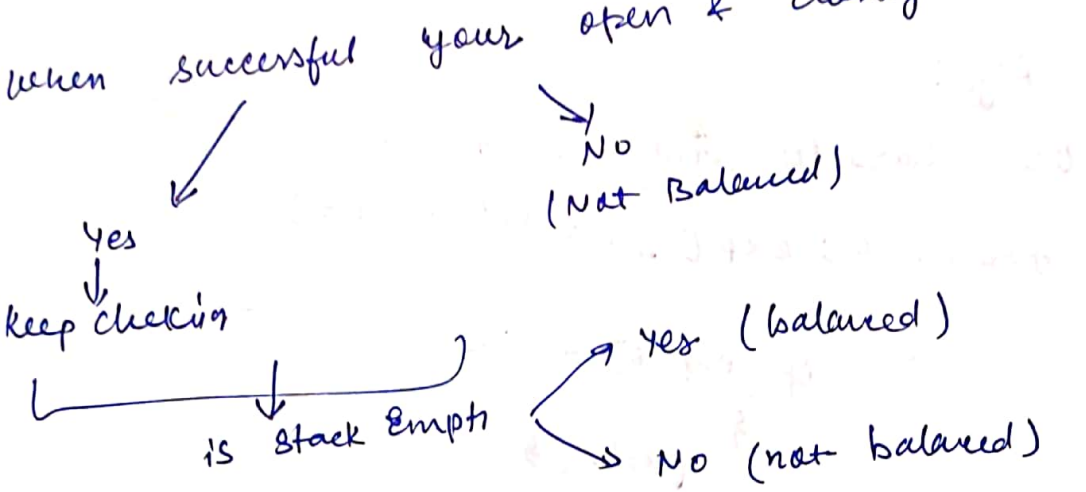
```
for (i=0; exp[i] != '\0'; i++)
{
    if (exp[i] == '(')
    {
        push ();
    }
    else if (exp[i] == ')')
    {
        if (isEmpty)
        {
            return 0;
        }
        pop();
    }
}//for
if (isEmpty())
{
    return 1;
}
else
{
    return 0;
}
```

Multiple Parenthesis

$a = \{ 7 - (3-2) + [8 + (99-11)] \}$

open : { ([→ push

close : })] → pop



* Infix, Prefix & Postfix *

Infix:	Prefix:	Postfix:
a+b	+ab	ab +
a*b	*ab	ab *

- Postfix is used by Computer/Machine

Infix to Prefix / Postfix :

1st : Parenthesis all

x - y * z

⇒ (x - (y * z))

⇒ (x - [* yz]) ⇒ [- x * yz]

Infix to Postfix :

$x - y * z$

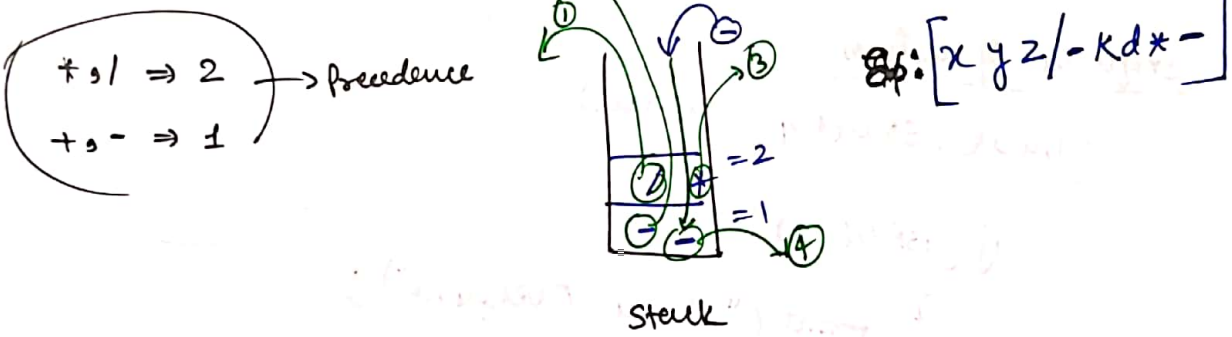
$\Rightarrow (x - (y * z)]$

$\Rightarrow (x - [yz*])$

$\Rightarrow [xyz*-]$

Infix to Postfix Using Stack

Exp: $x - y/z - k * d$



Ans: $[xyz/-kd*-]$

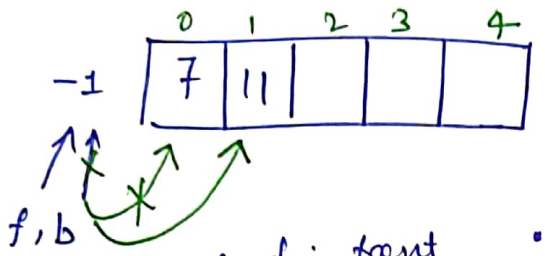
* Queue *

- FIFO (first in first out)

operations

- enqueue → add
- dequeue → remove
- first value → 1st place in Q
- last value → last " "
- Peak (pos) → pos. pe kaun sa element in Q
- is Empty → vaccant
- is Full → limit (full)

Queue Using Array



- f : front
- b : back
- initially both at -1 (when no element)

Enqueue operation

```
Enqueue (Struct* q , int val)
{
    if ( isFull (q) )
    {
        print (" Queue overflow");
    }
    else
    {
        q -> b = q -> b + 1 ;
        q -> arr [q -> b] = val ;
    }
}// fn
```

Empty Condition : f = b ; (front = back end)

Full Condition : b = (size - 1) ;
size = arr.length ;

- Introduced f & b both to reduce time complexity
front only at -1 , it don't store any element.

Dequeue Operation

```
dequeue (struct Que * q)
{
    int a = -1;
    if (q->f == q->b)
    {
        print ("empty ");
    }
    else
    {
        q->f++;
        a = q->arr[q->f];
    }
    return a;
}//fn
```

→ empty Queue condition

Queue full condition :

```
if (q->b == q->size-1)
{
    return 1;
}
```

Circular Queue

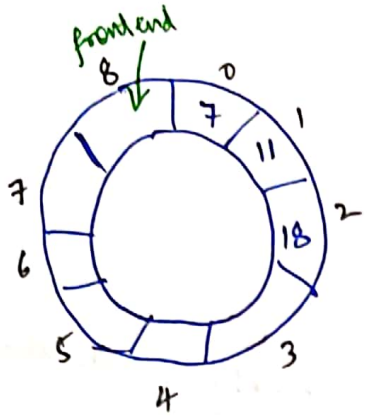
- when Q is not empty & still spaces are there, but we can't make use of that space.

⇓

Circular Queue

- Linear increment : `i = i + 1 ;`
- Circular increment : `i = (i+1) % size ;` → modulus

`size = arr. length`



Circular Queue

Enqueue operation

```
void enque(struct Q*q, int val)
{
  if((q->b+1)%(q->size)==q->f)
  {
    print("overflow");
  }
  else
  {
    q->b=((q->b+1)%(q->size));
    q->arr[q->b]=val;
  }
}//fn
```

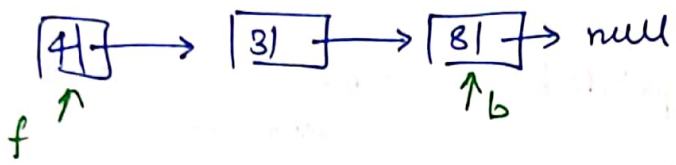
full condition

Dequeue operation

```
dequeue(struct Que*q)
{
  int val=-1;
  if(q->b==q->f)
  {
    print("Empty");
  }
  else
  {
    q->f=((q->f+1)%(q->size));
    val=q->arr[q->f];
  }
  return val;
}//fn
```

Empty condition

Queue Using Linked List



Enque - operation

```
void Enque (int val, struct Node * f,)
            struct Node * b
{
  N * n = struct node . . . . ;
  if (n == Null)
  {
    print ("full");
  }
  else
  {
    n -> data = val;
    n -> next = null;
    if (f == null)
    {
      f = b = n ;
    }
    else
    {
      b -> next = n;
      b = n ;
    }
  }
}
```

queue full

Empty Condition : f (front) == null

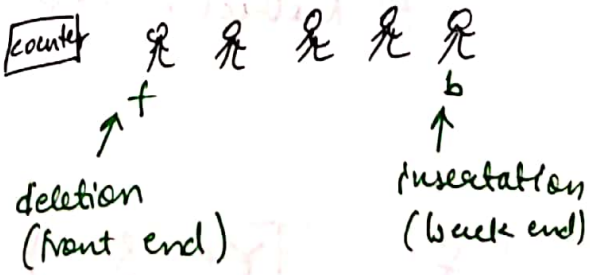
Full Condition : n == null
↑
jo node bana

Dequeue - Operation

```
int dequeue (struct Node * f)
{
  int val = -1;
  N * ptr = f;
  f = f -> next ;
  val = ptr -> data ;
  free (ptr);
  return val;
}
```

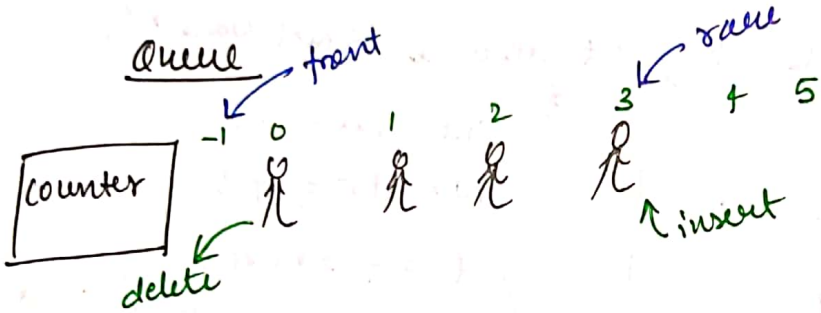
if Q not empty

Queue



DEQueue
Doubly Ended Queue

- Queue : FIFO principle
- DEQueue : Not FIFO
- not to be confused with dequeue



DEQueue

- Insertation from (rear + front)
- deletion from (rear + front)
- not follow FIFO rule

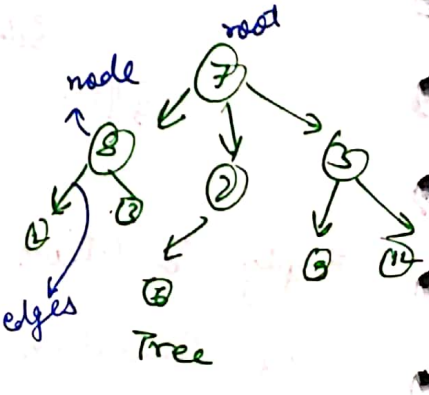
2 types

① Restricted Input DEQueue → insert from front not allowed

② Restricted Output DEQueue → deletion from rear not allowed

* Trees *

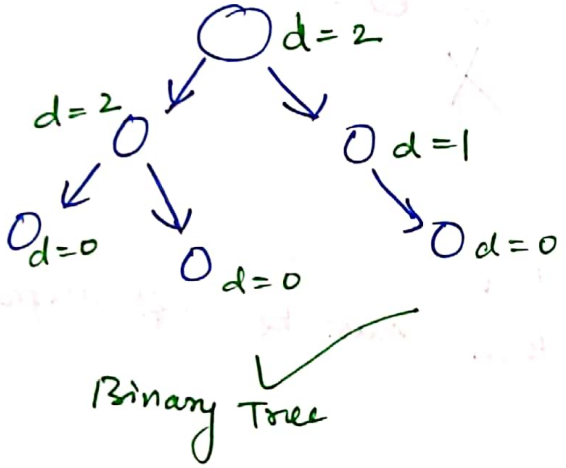
- Made up of nodes + edges
- represent the hierarchy of elements
- non-linear data structure

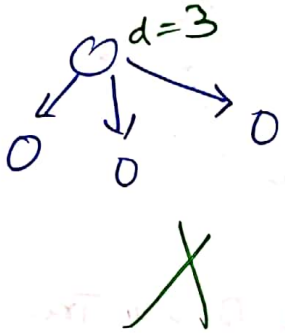


- internal node : node with atleast 1 child
- ext. node / leaf : node with zero/no child
- degree of node : no. of children of that node
- depth : no. of edges from root to that node

* Binary Tree *

- each node with degree ≤ 2 [0, 1 or 2]
- atmost 2 children

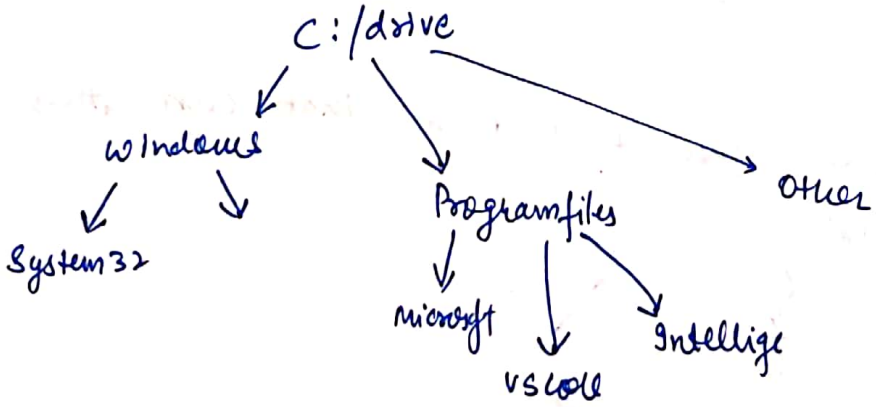




Binary Tree

- if n-nodes, then (n-1) edges

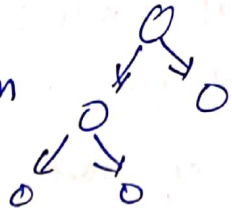
Types of Binary Tree



- we use tree to represent this type of data

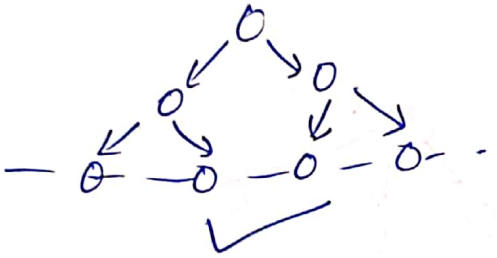
(1) Full or Strict Binary Tree :

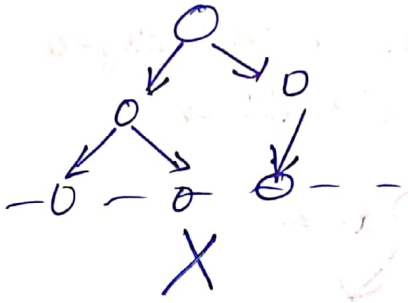
- All nodes either 0 or 2 children
- degree = 0 or 2



(2) Perfect Binary Tree

- All its internal node with degree = 2
  And all its leaf node on same level

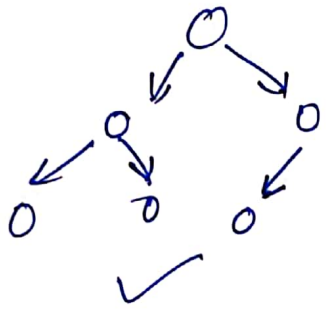


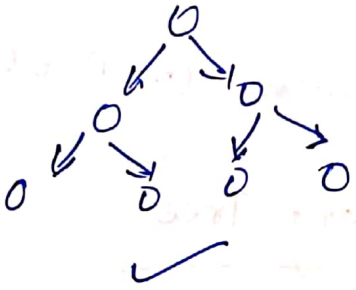


(3) Complete Binary Tree

i) all its level completely filled

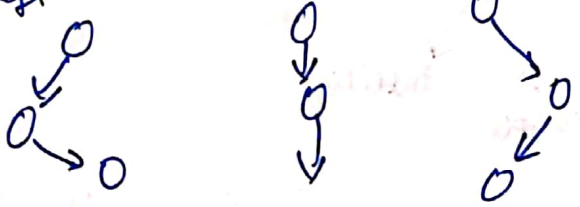
ii) if last level not filled then must be left-aligned





(4) Degenerate Tree

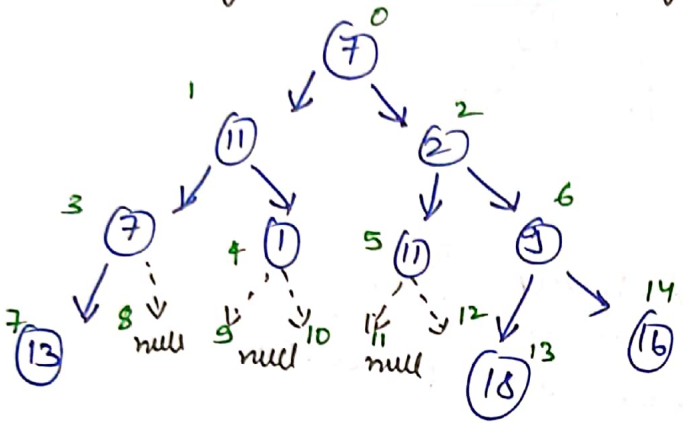
- every parent node just 1 child & that can either be left or right

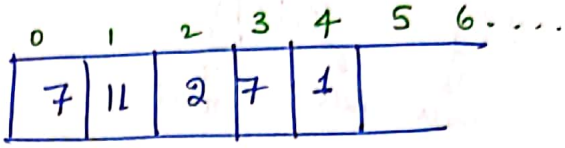


(5) Skewed Tree

- each parent → single child → strictly left or right

Array Representation of Binary Tree





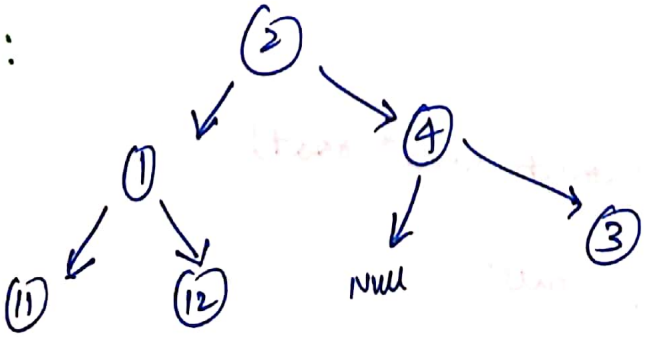
- If no element take it as 'null'

- Uses: → Search element faster
  → easily extendable
  → easy remove/add element
- But operation of it in array is costly → size can't be changed later
- So, array is not basically preferred, as to extend this, we have to make a new array for that.

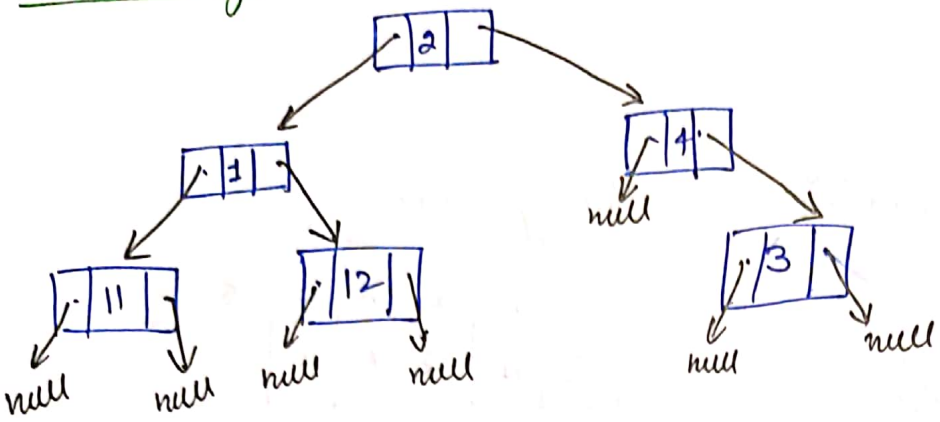
Linked Representation of Binary Tree

- null → no element
- Best way to represent tree
- Doubly linked list Used

Logical :

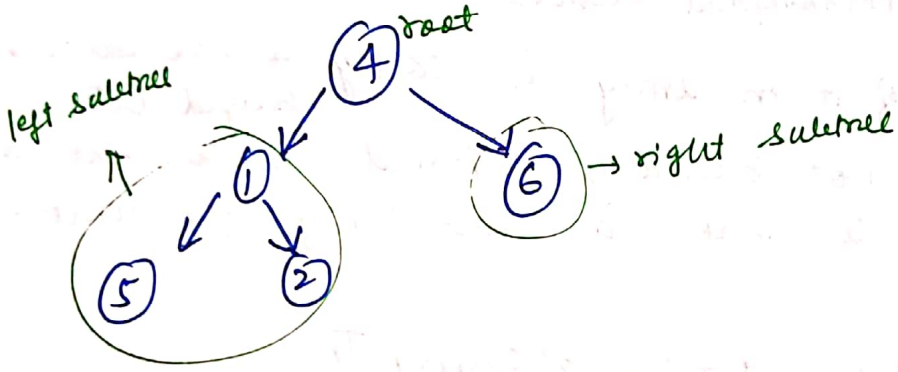


In Memory :



Traversal in Binary Tree

1) Pre-order : Root → left subtree → right subtree
2) Post-order : left subtree → right subtree → root
3) In-order : left subtree → root → right subtree



1) Pre : $4 \rightarrow [1 \rightarrow 5 \rightarrow 2] \rightarrow 6$
2) Post : $[5 \rightarrow 2 \rightarrow 1] \rightarrow 6 \rightarrow 4$
3) In : $[5 \rightarrow 1 \rightarrow 2] \rightarrow 4 \rightarrow 6$

Algorithm :

(1) Pre :

```
void preorder (struct node* root)
{
    if (root != null)
    {
```

```
print (root → data);
preorder (root → left);
preorder (root → right);
}
else
{
}
}
```

(2) Post :

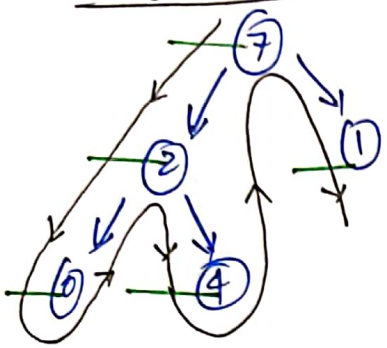
```
void postorder (struct node* root)
{
  if (root != null)
  {
    postorder (root → left);
    post order (root → right);
    print (root → data);
  }
}
```

(3) In :

```
void inorder (struct node* root)
{
  if (root != null)
  {
    inorder (root → left);
    print (root → data);
    inorder (root → right);
  }
}
```

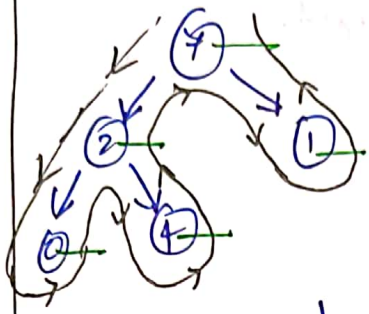
Trick to find :

(1) Inorder :



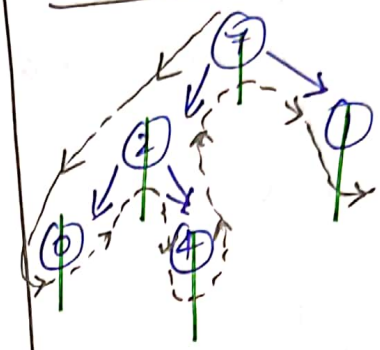
(7 2 0 4 1)

(2) Post-order :



(0 4 2 1 7)

(3) In-order :

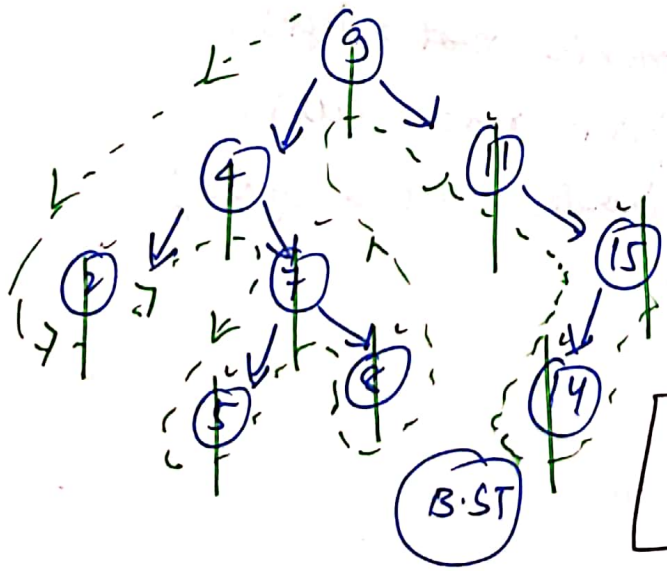


(0 2 4 7 1)

* Binary Search Tree *

B.S.T

- no duplicate node (of same value)
- left & right sub-tree also binary tree
- all node of right sub > all node of left-subtree
- inorder Traversal of BST → Ascending sorted Array



⇒ 2, 4, 5, 7, 8, 9, 11, 14, 15

- $T_n = h \times t$

$\log n \leq h \leq n$

Best : O(log n)
Worst : O(n)

Algo for Search :

```
while (root != NULL)
{
    if ((root -> data) == key)
    {
        return root;
    }
    else if (key < (root -> data))
    {
        root = root -> left;
    }
    else if (key > (root -> data))
    {
        root = root -> right;
    }
} // while
```

Insert :- • Create 2 pointer → used traversal searching technique

Deletion in B.S.T

Case I : node is leaf :

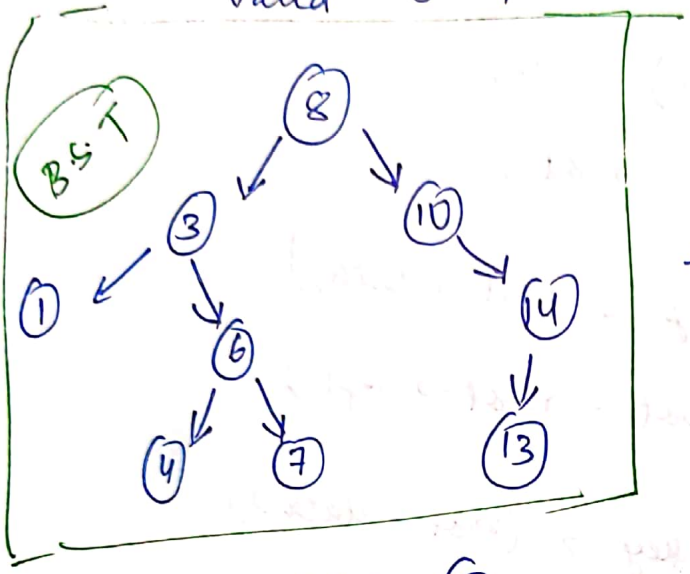
- Search the node
- delete the node
- make its parent node point to null

Case II : Node is non-leaf node :

- Search that node
- delete that node

and replace it position with its :
inorder predecessor or inorder successor

- substitute any & the tree will still be valid B.S.T



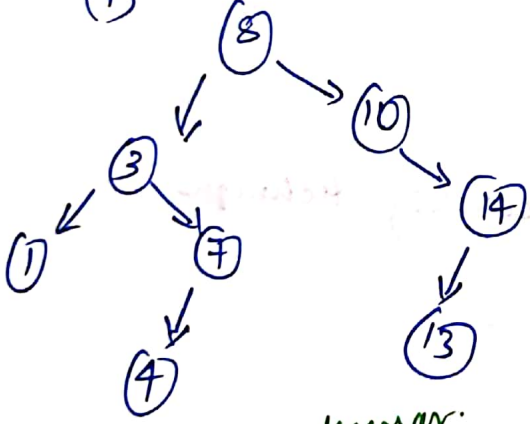
inorder :

1 → 3 → 4 → 6 → 7 → 8 → 10 → 13 → 14

predecessor ↑ 6 ↑ successor

delet : 6

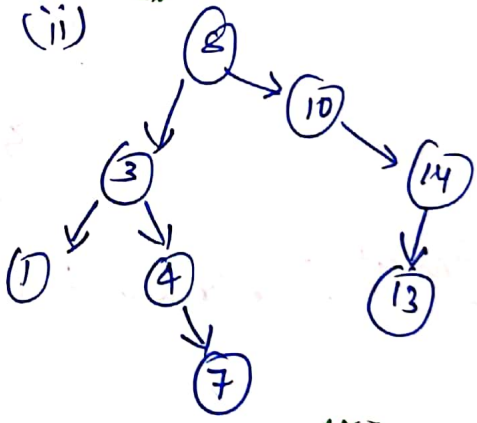
(i)



pre-decessor.

B.S.T

(ii)



successor.

B.S.T.

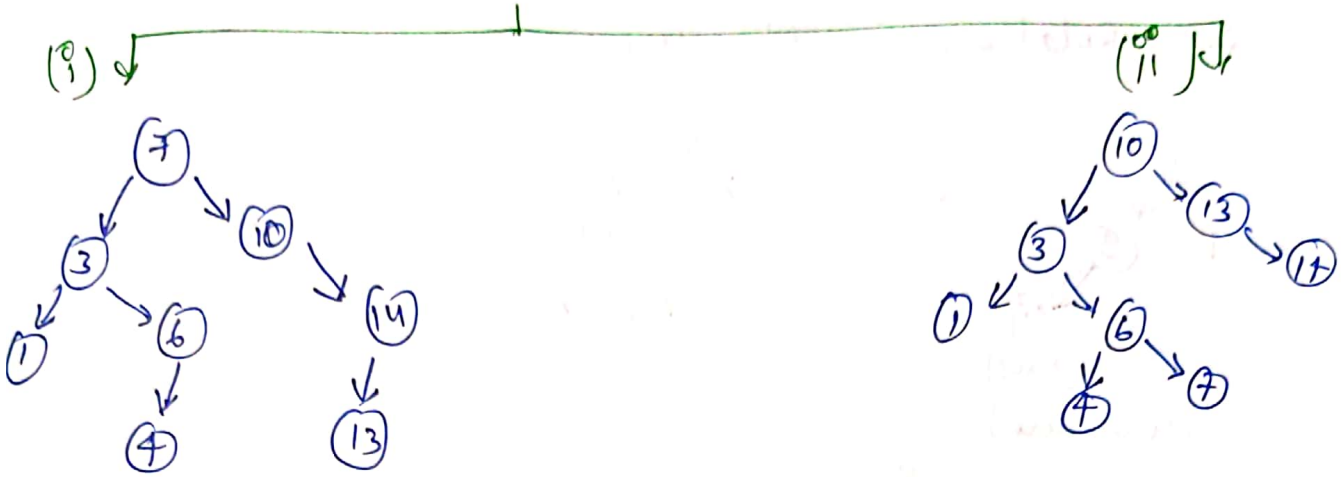
Case III : delete the root node

- write inorder traversal of whole tree

delete (8) → root node

- inorder :⇒ 1 → 3 → 4 → 6 → 7 → 8 → 10 → 13 → 14

predecessor ↑ 8 ↑ successor



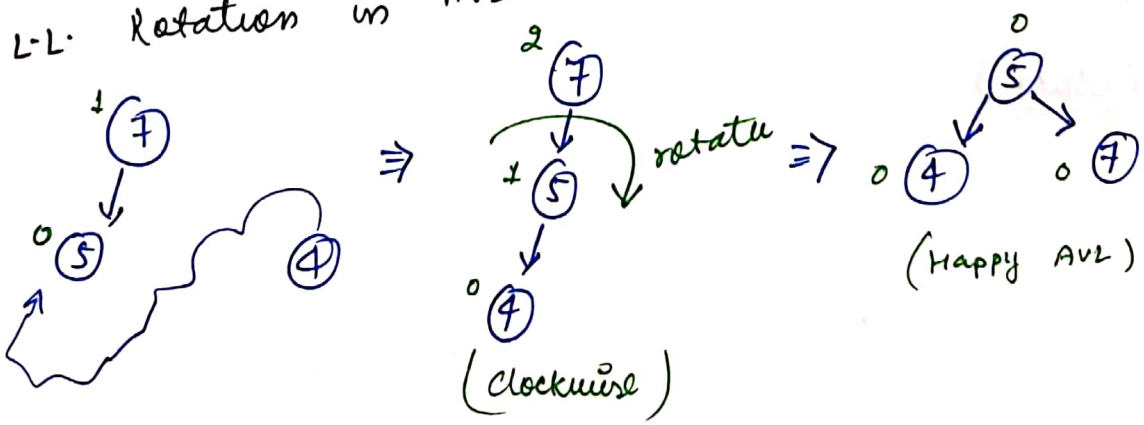
* AVL - Tree *

- $T_n = O(h)$ → in B.S.T
  - ↑ Ht. of tree
- for eg: skewed tree me tn increase ho jata hai

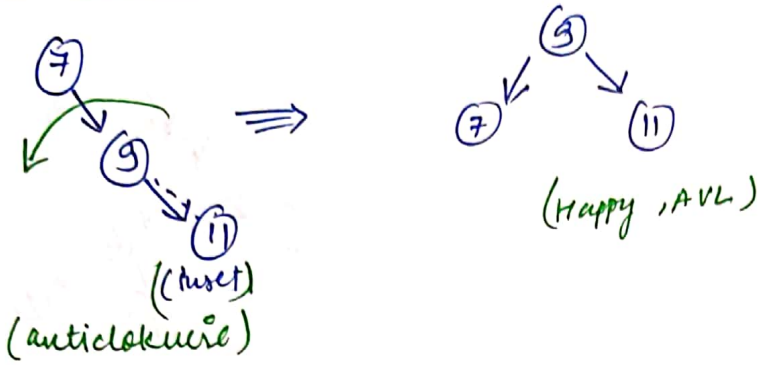
AVL
- Height balanced tree
- Balance Factor (BF) = (Ht. of right subtree) − (Ht. of left subtree)
- $|BF| \leq 1$ $\quad BF = \{-1, 0, 1\}$

Insertation

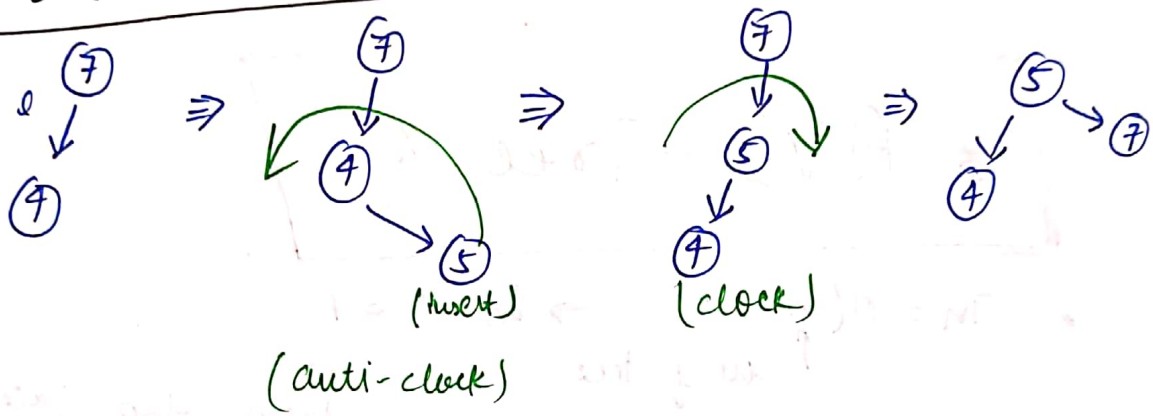
(1) L-L. Rotation in AVL Tree



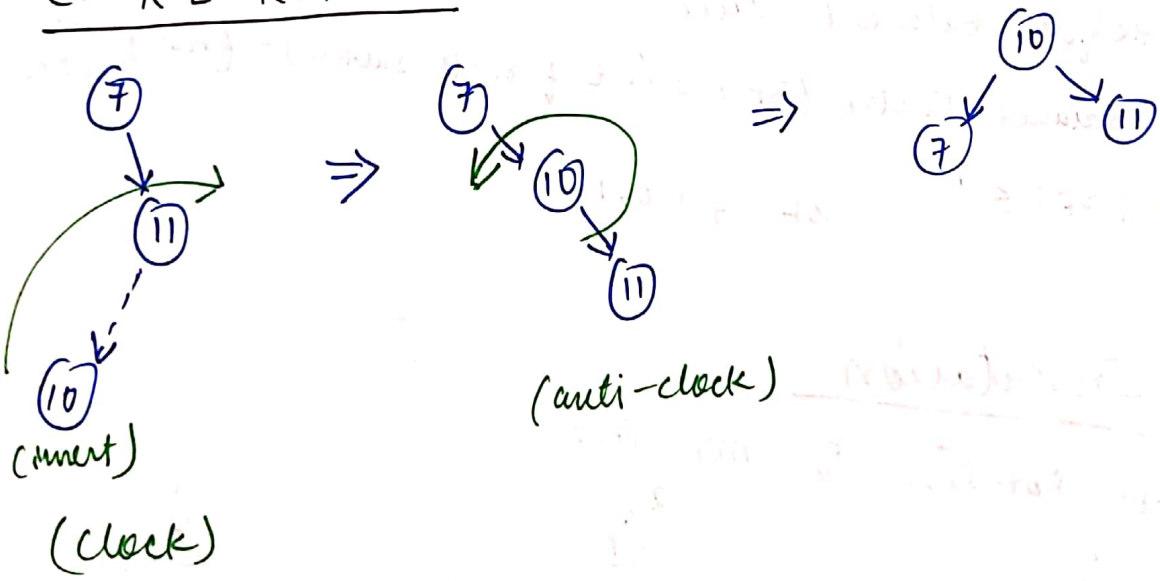
(2) R·R Rotation in AVL Tree



(3) L·R Rotation

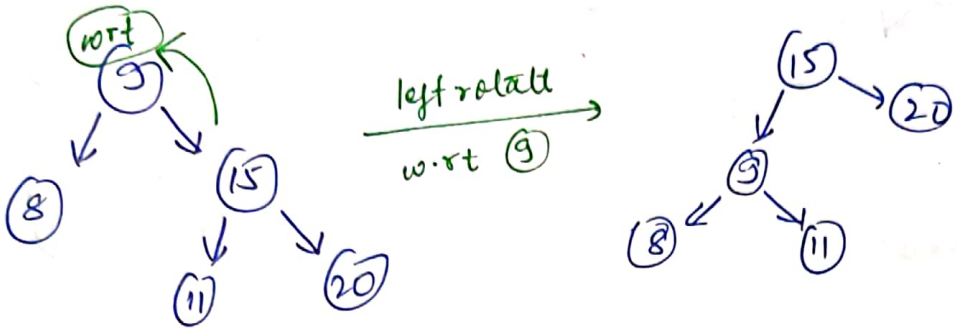


(4) R·L Rotation

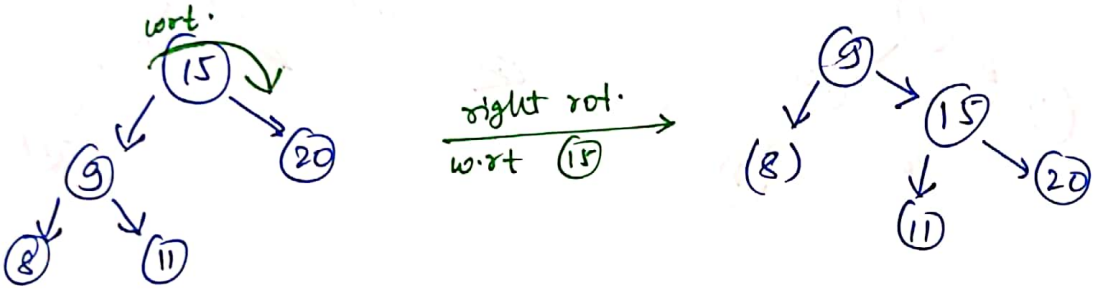


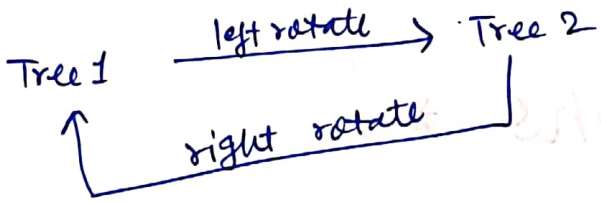
Rotation with multiple Nodes:

(1) Left Rotate wrt a node



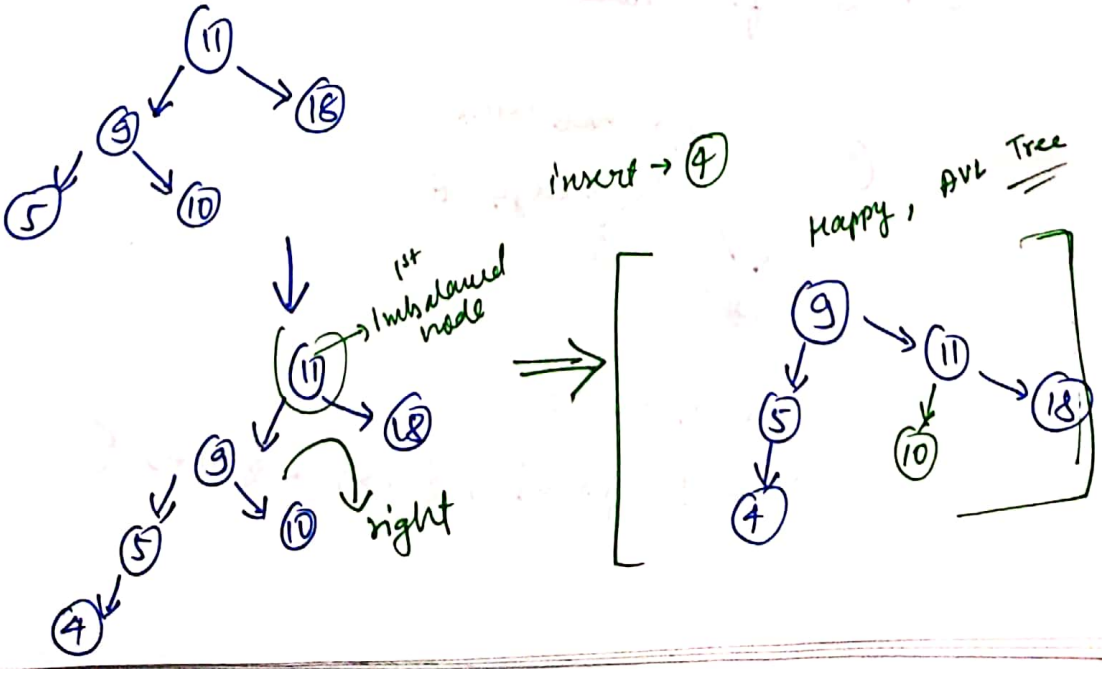
(2) Right Rotate wrt a node



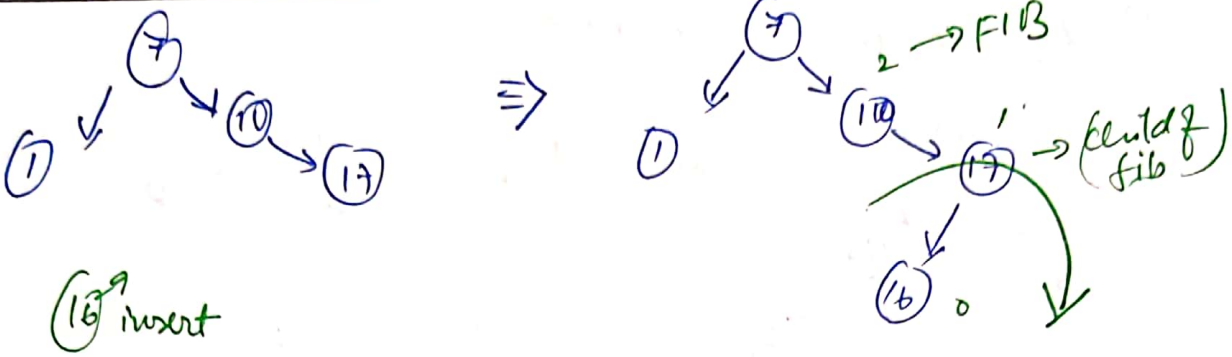


* Balancing AVL after Insertation *

(1) Left - Left Insertation



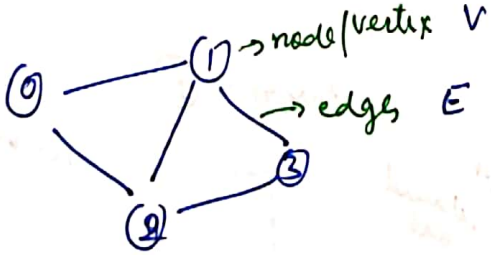
(2) Right - Left Insertation



FIB: first imbalanced node

* Graphs *

- Linear D.S : Array, L.L, Stack, Queue
- Non-linear D.S: Tree, Graph
- Collected of nodes/vertices connected through edges



$V = \{0, 1, 2, 3\}$

$E = \{\{0,1\}, \{0,2\}, \{1,3\}, \{3,2\}\}$

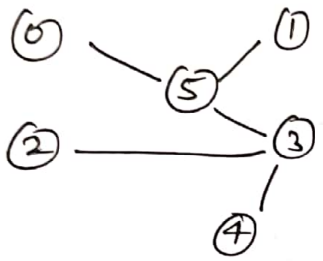
- indegree → no of edges going out
- out degree → no of edges into the node

Ways to Represent Graph

(1) Adjacency list (2) Adjacency matrix

(3) Cost Adjacency Matrix

Eg :-



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

⇒ ~~cost~~ Adjacency Matrix

⇓

if connected → '1'

otherwise → '0'

0 : 5
1 : 5
2 : 3
3 : 2 → 4 → 5
4 : 3
5 : 0 → 1

Adjacency List

Cost Adjacency Matrix

if connected → 'cost'

otherwise → '-1'

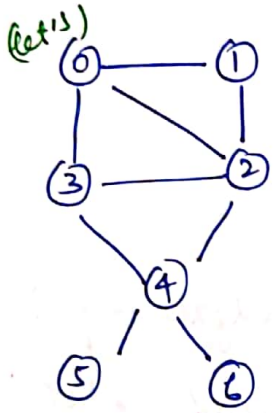
similar to Adjacency matrix

- Sequence of steps known as 'Graph Traversal Algorithm'
  - * (i) BFS → Breadth First Search → (Queue)
  - * (ii) DFS → Depth First Search → (Stack)

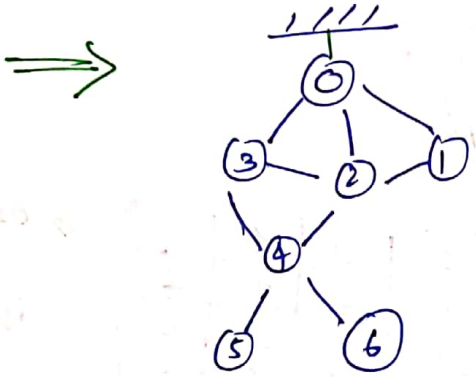
B.F.S

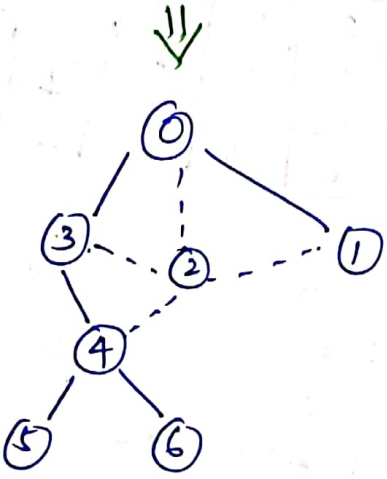
Method-1

(Let's)



* start with any node



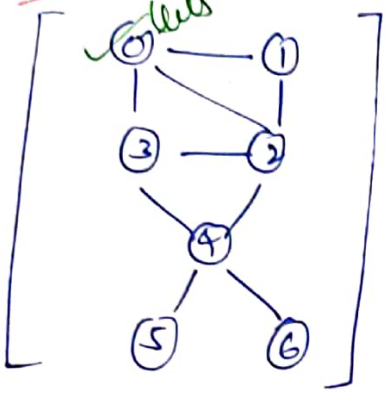


Level Order Traversal :- simply write the node in the same level from left to right

LOT => [0 → 3 → 2 → 1 → 4 → 5 → 6]

∴ BSF

Method-2



Exploration Q: holds the nodes we'll be exploring one by one

Visited Q: array holding the states of whether a node is V or not

• start with any node

① V: 0
Ex: ~~0~~, 1, 2, 3

② V: 0, 1
Ex: ~~1~~, 2, 3

③ V: 0, 1, 2
Ex: ~~2~~, 3, 4

④ V: 0, 1, 2, 3
Ex: ~~3~~, 4

⑤ V: 0, 1, 2, 3, 4
Ex: ~~4~~, 5, 6

⑥ V: 0, 1, 2, 3, 4, 5
Ex: ~~5~~, 6

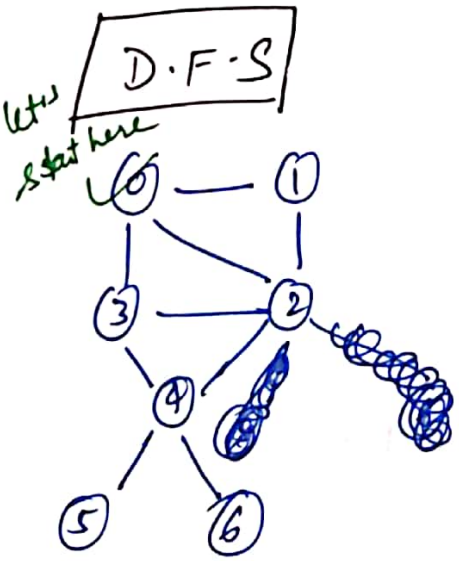
⑦ V: 0, 1, 2, 3, 4, 5, 6
Ex: ~~6~~

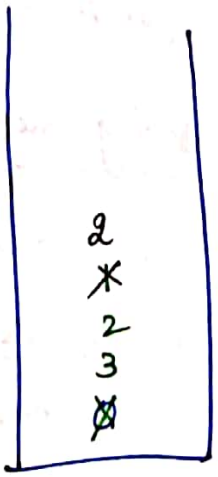
∴ BFS = [0 → 1 → 2 → 3 → 4 → 5 → 6]

• the order of BFS can be any.

D·F·S

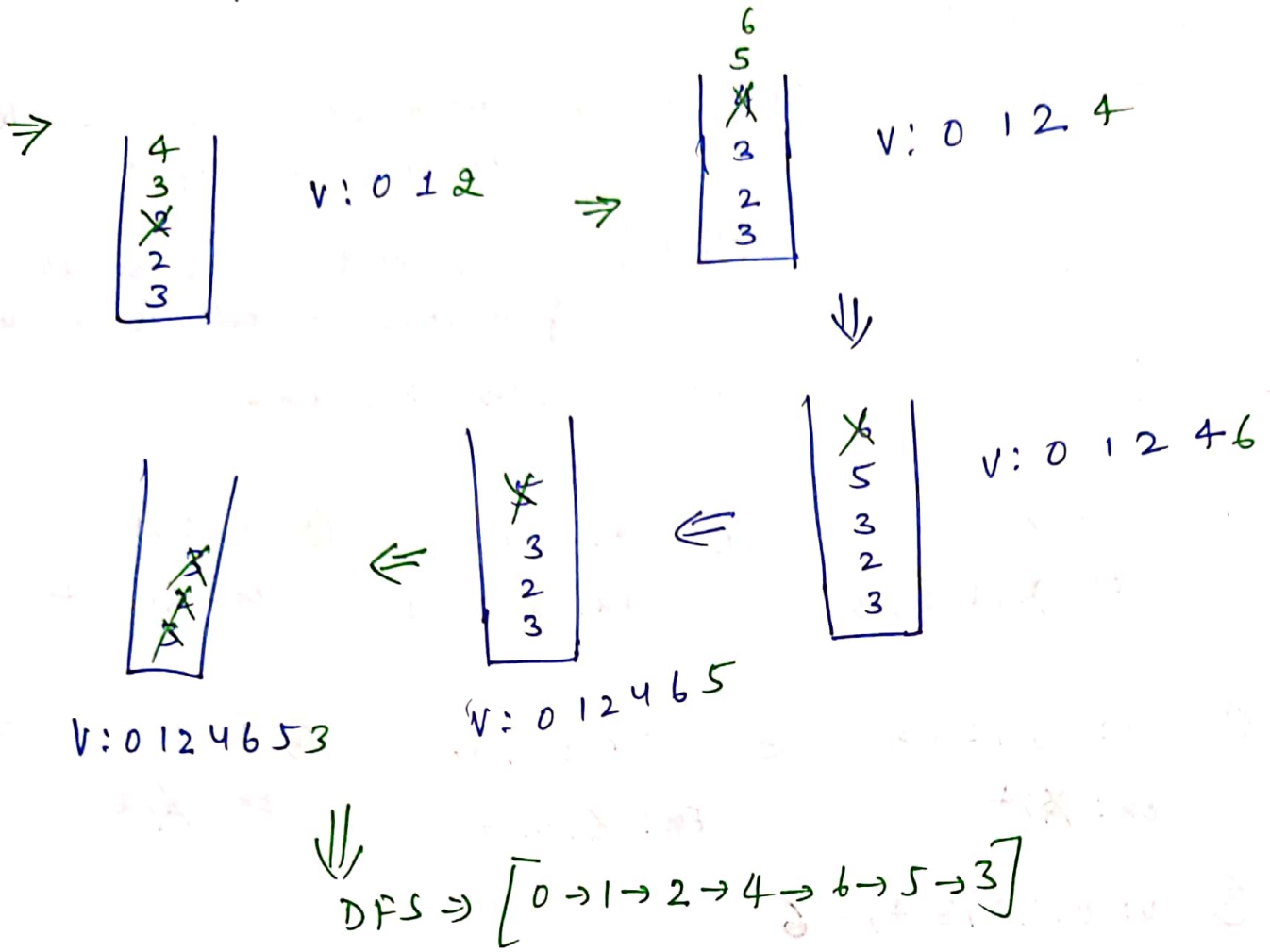
(Let's start here)





Stack

Visited: 0 1

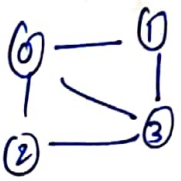


Spanning Trees

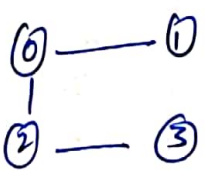
(i) all vertices of graph must be present

(ii) no of edge = (vertex − 1)

Graph



Spanning tree



no of edg = 4 − 1 = 3

* No. of spanning tree = (n^{n-2})

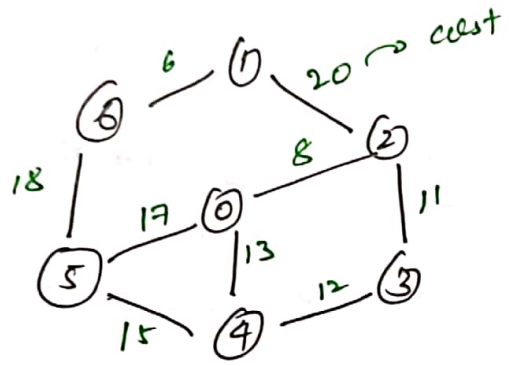
where n = no of vertices in graph.

- Cost of Spanning tree : sum of wt. of all the edges in tree
- Min. spanning tree : spanning tree with minimum cost

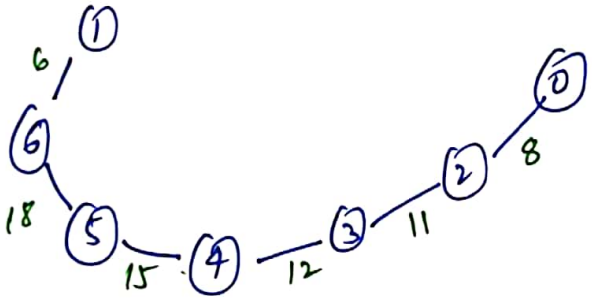
Prim's Algorithm

- approach to find min. spanning tree

Eg :-



choose any start pt ⇒ 0 (zero) - let's say



Prim's Algo cost = 70 (6+18+15+12+11+8)

Not included	Included
{0,1,2,3,4,5,6}	= {1}
{0,2,3,4,5}	{1,6}
{0,2,3,4}	{1,5,6}
{0,2,3}	{1,4,5,6}
{0,2}	{1,3,4,5,6}

{0}

{1, 2, 3, 4, 5, 6}

Steps :-

① 1 → 2 : 20 X
1 → 6 : 6 ✓

② 1 → 2 : 20 X
6 → 5 : 18 ✓

③ 1 → 2 : 20 X
5 → 0 : 17 X
5 → 4 : 15 ✓

④ 1 → 2 : 20 X
4 → 3 : 12 ✓
4 → 0 : 13 X
5 → 0 : 17 X

⑤ 1 → 2 : 20 X
3 → 2 : 11 ✓
4 → 0 : 13 X
5 → 0 : 17 X

⑥ 0 → 5 : 17 X
0 → 4 : 13 X
0 → 2 : 6 ✓