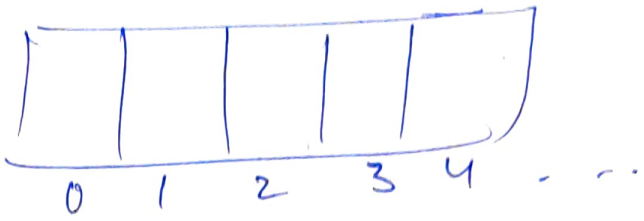
1-D array



location of element-

LA → linear array

$$LOC(LA[K]) = Base(LA) + W(K - LB)$$

W → size of data type

Base address

LB = Starting index

UB = Higher index − 1

a[5] → LB = 0

∴ HB = 4

size of array =

no. of elements = $\boxed{UB - LB + 1}$

Q. Suppose array with LB = 1.
Arr [10], base address = 1020,
size of each element = 2, find address of arr [5].

(2)

→ traversal
→ insertion
→ deletion

2-D array

a [5] [2]
- [5] → rows
- [2] → column

00	01
10	11
20	21
30	31
40	41

→ insertion
→ transpose

Address of 2-D

(1) Row major

→ elements are stored row wise
→ rows are listed on the basis of columns.

$$a \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

→ | a_{00} | a_{01} | a_{02} | a_{10} | a_{11} | a_{12} | ...

r_1 r_2 r_3

Add of element-

$$A[a[i][j]] = B.A + W(C(i - LB_r) + (j - LB_c))$$

- $B.A$ → base addr.
- W → size of element
- C → no. of column
- LB_r → LB of row
- LB_c → LB of column

(4)

no. of column = $UB_c - LB_c + 1$

no of row = $UB_r - LB_r + 1$

Ques

Arr [4:7, -1:3], size = 2.

B.A = 100, find add of Arr[6][2] using Row major.

(2) Column Major

$$A = \begin{pmatrix} a_{00} & a_{01} & a_{02} \\ a_{10} & a_{11} & a_{12} \\ a_{20} & a_{21} & a_{22} \end{pmatrix}$$

a_{00}	a_{10}	a_{20}	a_{01}	a_{11}	a_{21}	a_{02}	a_{12}	a_{22}

C_1 C_2 C_3

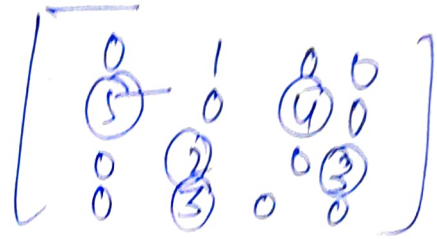
$$A[arr[i][j]) = B.A + W((i - LB_r) + R(j - LB_c))$$

W → size of element; LB_r → LB of row; R → no. of rows; LB_c → LB of column

Sparse matrix

→ 2D array
→ min. no. of non zero elements
→ (mostly zeroes)
→ reduces traversing (scanning time).

arr a.



only 6 non zero elements

3 col. representation of sparse matrix

First row contains total no. of row, col & non zero.

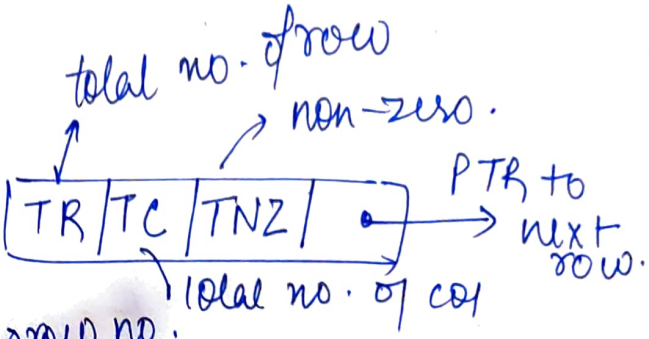
	Rows	Column	Values
1	4	4	6.
2	0	1	1
3	0	0	5
4	1	2	4
5	2	1	2
6	2	3	3
7	3	1	3

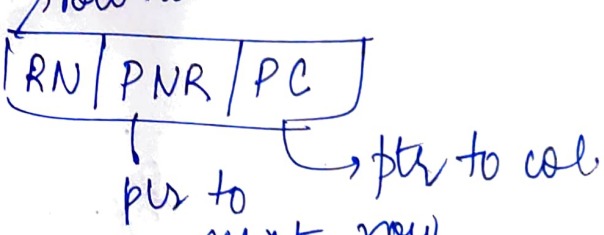
→ Sparse matrix.

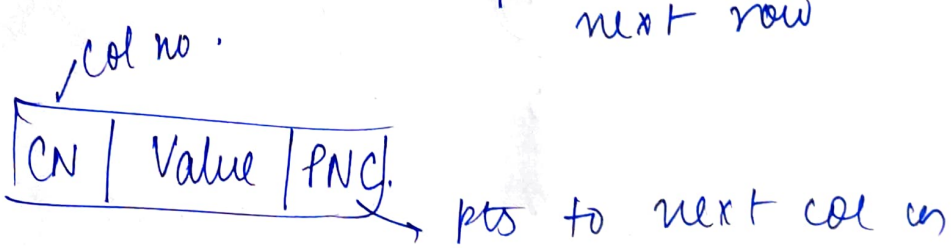
(6) Linked list representation of sparse matrix.

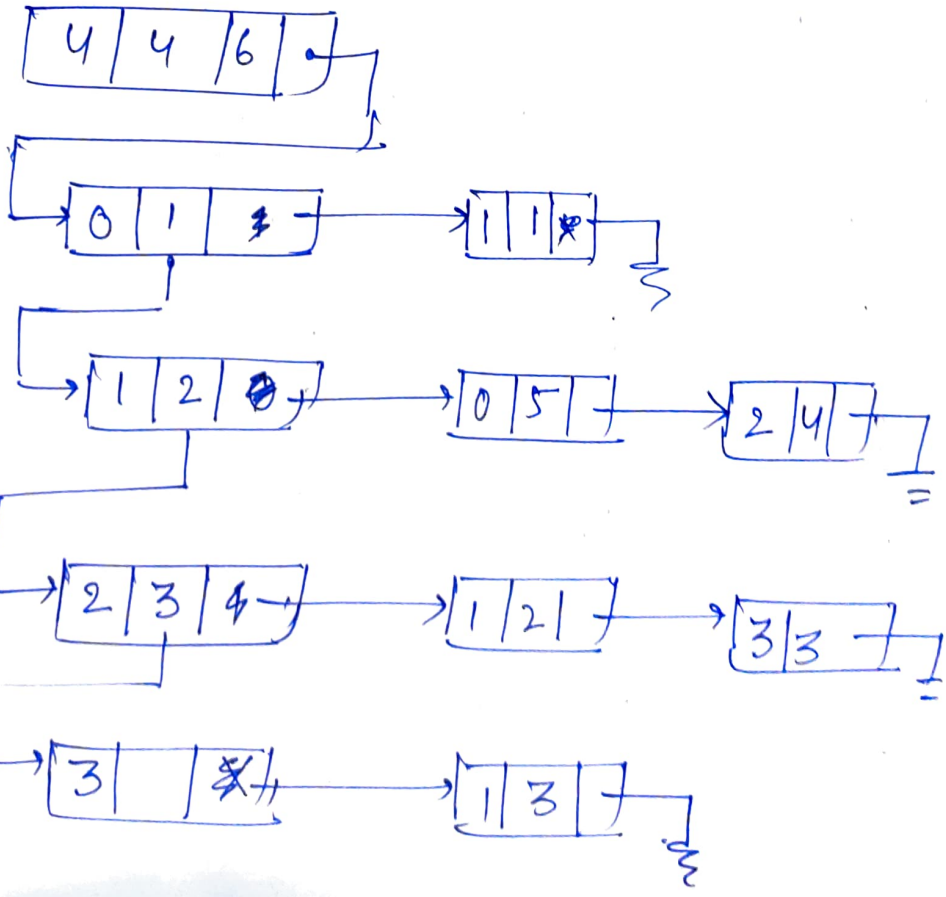
three structures

- Head node
- Row node.
- Col node.



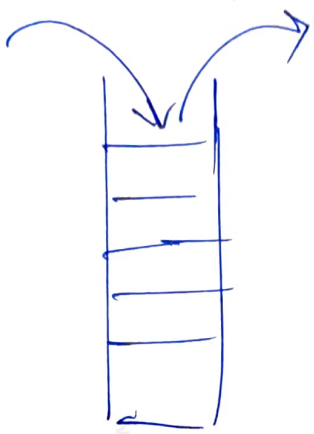






Stacks : LIFO

↳ last in first out



push : insertion
pop : deletion.

} from top of the stack.

Initial condition of top

top = -1. ⟶ stack empty.

Push ⇒ top = top + 1.

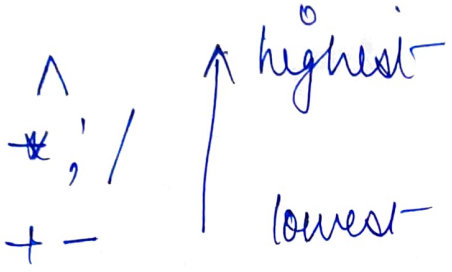
Pop ⇒ top - - ;

(8) Application of stack

(A) Infix → postfix.

A+B/C → ABC/+.

Precedence.



(B) Postfix → Infix.

ABC/+ → A+B/C.

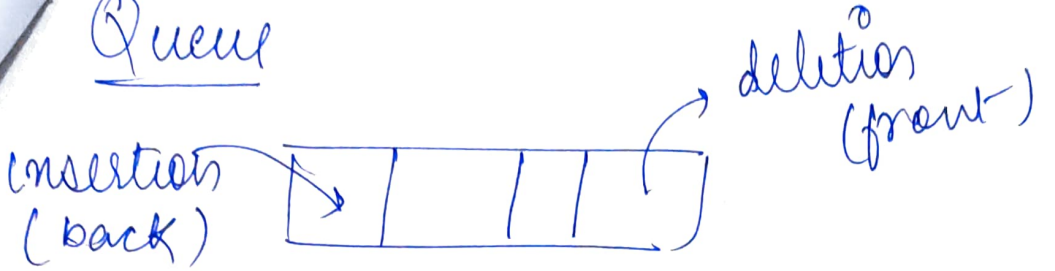
(C) Infix → Prefix

reverse given expression
↳ find postfix
↳ reverse the exp.

(D) Prefix → Infix

↳ scan from right to left $
do same as postfix → infix.

Queue



FIFO (first in first out).

enqueue → insertion
dequeue → deletion

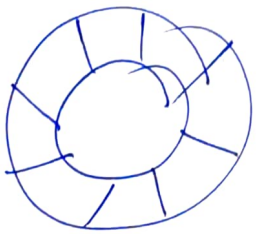
two variables
front;
rear;

front = rear = -1 → empty.
front = 0, rear = 0 → 1 element

→ a[rear++]; → insertion
→ front++; → deletion

(10)

Circular queue

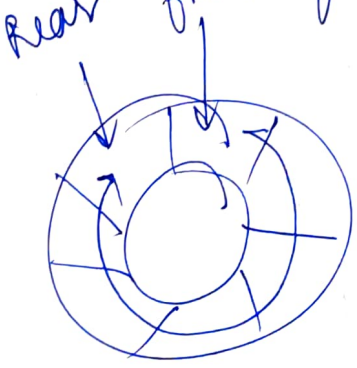


benefit over regular queue : empty slots are reused thus saving memory.

Clockwise operation :

max queue : ~~Rear = front~~.

front = rear + 1



Queue full.

rear = rear + 1 → insertion

front = front + 1 → deletion.

Dequeue (double ended queue)

→ Insertion, deletion from both sides

two types

- **Input restricted queue**
  (insertion can be done from rear end but deletion can be done from both ends)
- **Output restricted queue**
  (deletion can be done from front only but insertion from both ends)

→ **PRIORITY Queue**

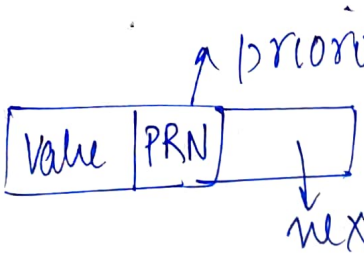
Deletion/Insertion based on priority (VIP)

→ Higher priority > Lower priority

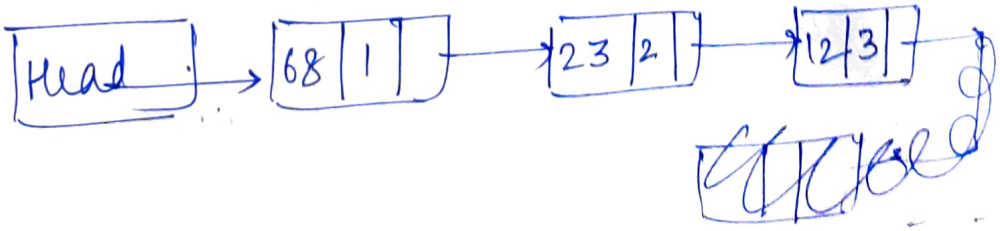
→ Same priority → First come first serve.

types of priority queue

- Ascending order priority queue.
  (Lower priority no. is given to high priority elements)
- Descending order priority queue.
  (higher priority no. is given to high priority elements)



Info.	priority
23	2
42	4
51	4
68	1
17	5
12	3.



Linked list

→ Dynamic data structure
(size can grow & shrink)

→ Each element of linked list is called a node

node has 2 → [info] → ptr to next node.

→ Self referential data type

```
struct node
{
    int info;
    node *next;
};
```

→ points to itself in declaration.

→ Insertion / deletion at any node is allowed

→ Random Access is not allowed (traverse from start to the node we want)

Advantages:

→ dynamic
→ memory efficient (random chunks)
→ easy insertion

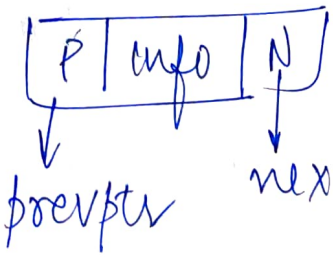
Disadvantages

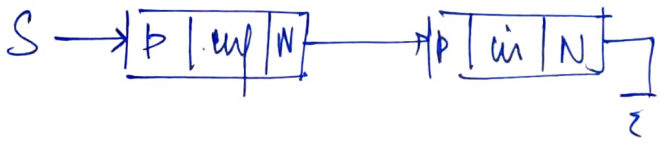
→ more memory
→ Random access is not allowed.
(more time consumption).

Insertion: begining, end, middle.
deletion: beginning, end, middle.

Doubly linked list

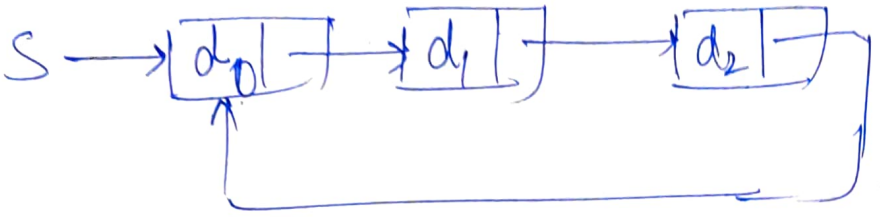
nodes



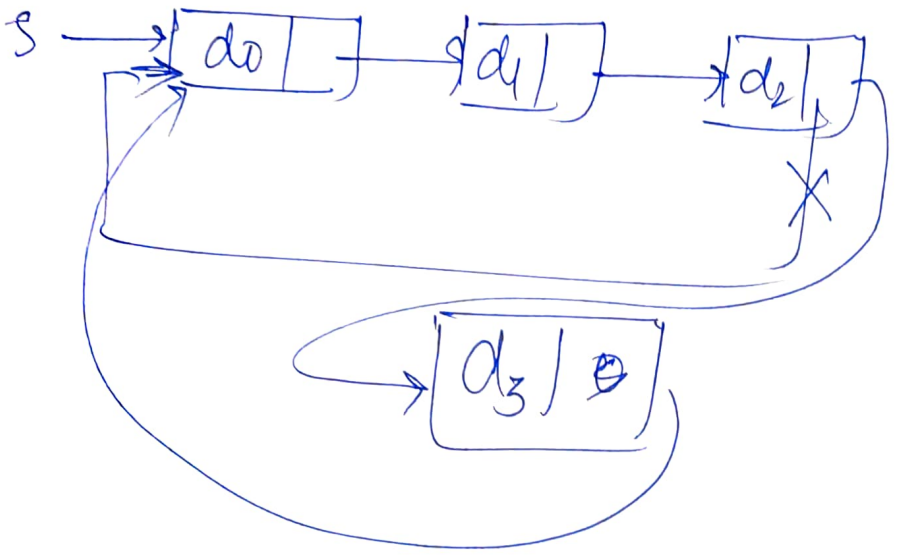


traversing in both direction can be possible.

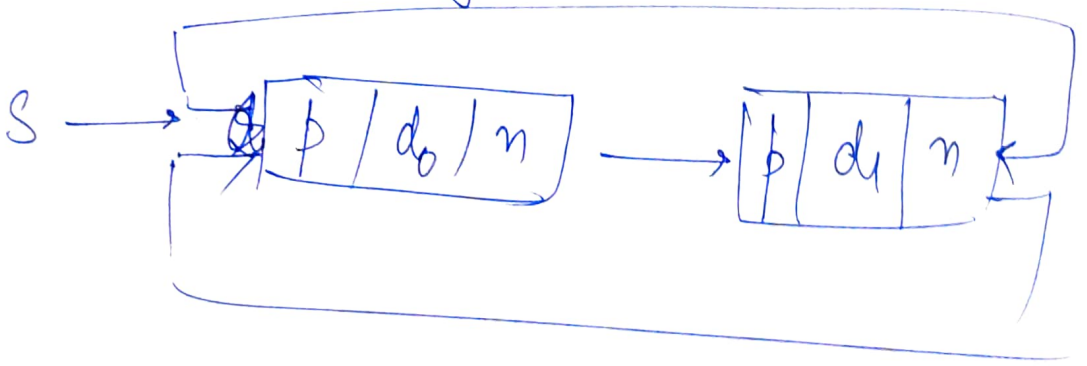
Circular linked list



addition in list

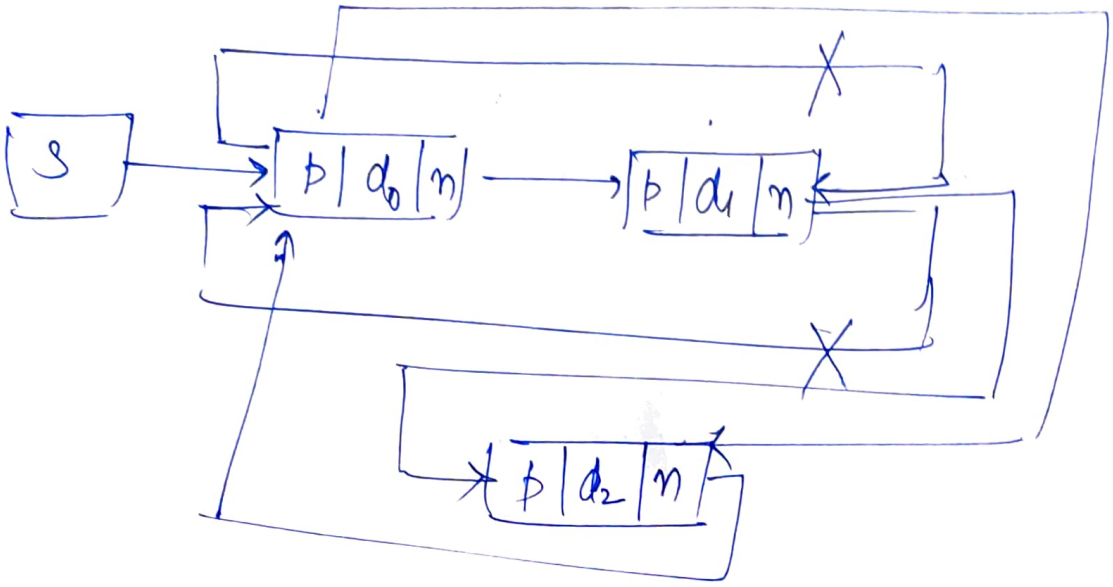


Circular doubly linked list



16.

addition in circular doubly linked list



Insertion in array:

1. Start
2. In array 'a' of n elements.
3. Input element from user 'ITEM' and position 'K'.
4. Set J = n
   [while] J ≥ K
   Set a[J+1] = a[J]
   Set J = J - 1.
   [end while]
5. Set a[K] = ITEM.
6. Set N = N+1.
7. Exit.

Deletion in an array.

1. Start.
2. Let array 'a' of n elements.
3. Input position at which the element is to be deleted 'k'.
4. Set J = K.
   [while] J < N
   Set a[J] = a[J+1]
   Set J = J+1.
   [end while]
5. Set N = N-1
6. Exit.

Traversing in array.

1. Start-
2. Let- array 'a' of n elements.
3. [While] J = 0

   [While] J < N

   Print- a [J]

   Set J = J + 1

   [end while]
4. Exit-

Stack

→ Push – Insertion.

1. Start-
2. Let stack 'a' of n elements.
3. Input element 'ITEM'.
4. if top = max,
   print- Overflow and Exit.
5. Set top = top + 1.
6. Set Stack [top] = ITEM.
7. Exit.

→ Pop – deletion.

1. Start-
2. let stack 'a' of n elements.
3. if top = -1,
   print- underflow. and exit.
4. Set top = top - 1.
5. Exit.

Queue

Insertion (enqueue)

1. Start.
2. Let there be a queue 'a' of n elements.
3. Input the element to be inserted 'ITEM'.
4. if ~~F=0~~. R = MAX,
   print overflow and exit.

   else if F = R = -1,
   Set F = R = 0,

   else.
   set Rear = Rear + 1.
5. Set a[Rear] = ITEM.
6. Exit.

Deletion (dequeue)

1. Start.
2. Let there be queue 'a' of n elements
3. If F = -1
   print underflow and exit
   else if F = R = 0
   Set ~~front~~ F = R = -1
   else
   Set F = F + 1
4. Exit.

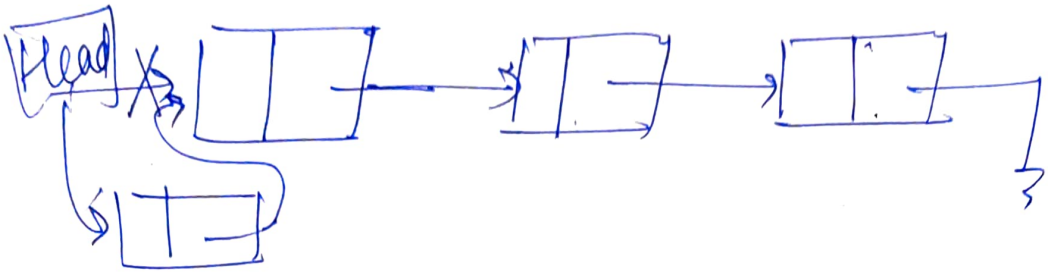
Circular Queue

Insertion

1. Start.
2. Let there be a circular queue 'a' of n elements
3. Input the element - 'ITEM' to be inserted.
4. if F = R+1 or F = 0 & R = size - 1.
   print queue is full & exit.
5. else if
   Rear = MAX.
   Set Rear = 0;
   else
   Rear = Rear + 1.
6. Queue [Rear] = ITEM.
7. Exit

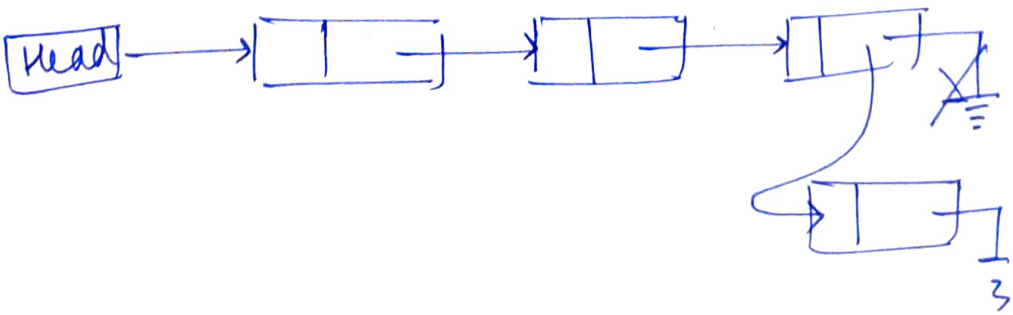
Linked list → Insertion

at beginning



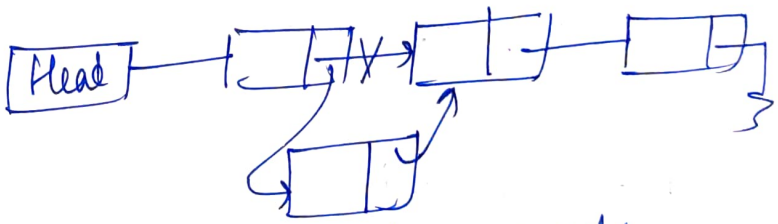
1. Start-
2. Let there be a linked list containing n nodes.
3. ~~input~~ value.
4. Create a new node 'temp'
5. Set 'temp → info = VALUE
6. Set temp → next = head.
7. Set head = temp.
8. Exit-

at end.



1. Start.
2. Let there be a linked list containing n nodes.
3. Input value to be inserted 'VALUE'.
4. Create a new node 'temp'.
5. Set temp → info = VALUE.
6. Traverse the linked list till next-ptr = NULL, ~~Set node~~
7. Set this node as 'prevnode'.
7. Set prevnode → next = temp;
8. Set temp → next = NULL.
9. Exit.

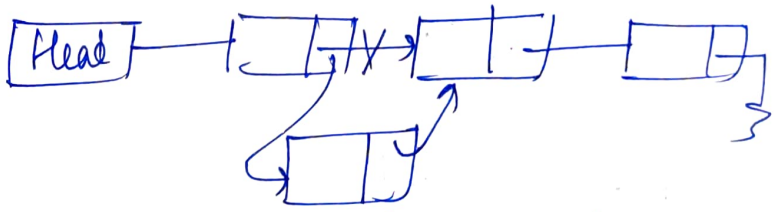
in middle



1. Start
2. Let there be a linked list of n nodes.
3. Input value to be inserted 'VALUE'
5. Create a new node temp.
6. Set temp → info = VALUE.
7. Traverse to the position the node is to be inserted. Set this node as 'prev_node'.
8. ~~Set next_node = prev_node → next;~~

1. Start.
2. Let there be a linked list containing n nodes.
3. Input value to be inserted 'VALUE'.
4. Create a new node 'temp'.
5. Set temp → info = VALUE.
6. Traverse the linked list till next ptr = NULL, Set this node as 'prevnode'.
7. Set prevnode → next = temp;
8. Set temp → next = NULL.
9. Exit.

in middle



1. Start
2. Let there be a linked list of n nodes.
3. Input value to be inserted 'VALUE'
5. Create a new node temp.
6. Set temp → info = VALUE.
7. Traverse to the position the node is to be inserted. Set this node as 'prev_node'.
8. Set next_node = prev_node → next;

~~8. Set prev_node → next = temp;~~

9. Set temp → next = ~~next node~~ prev_node → next;

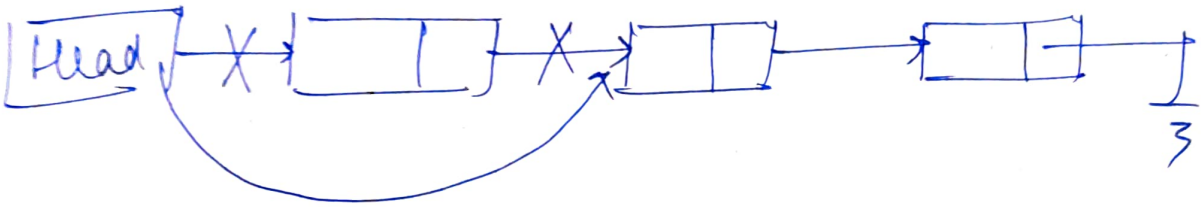
~~10. Exit.~~

10 Set prev_node → next = temp;

11 Exit.

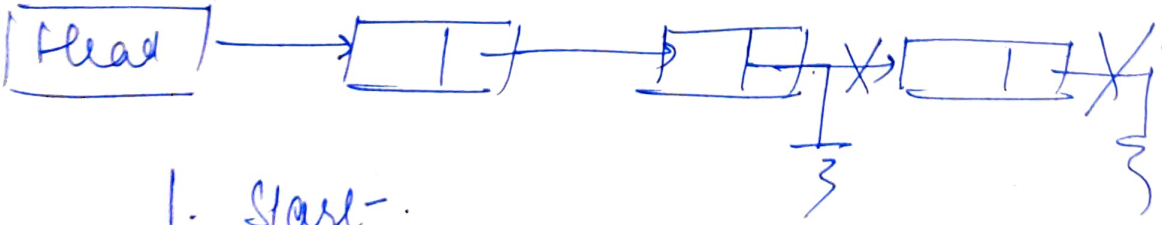
Deletion

at the beginning.



1. Start.
2. Let there be a linked list containing n nodes.
3. Set head = head → next;
4. Exit.

at the end

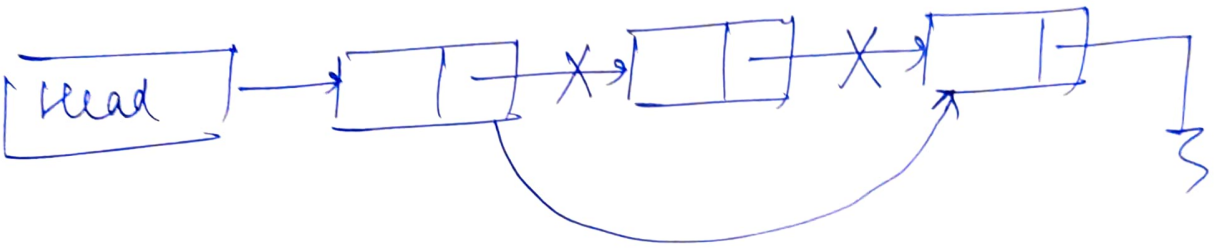


1. Start.
2. Let there be a linked list containing n nodes.
3. Traverse the linked list ~~one~~ till

One node before the last. Set this node as prev_node.

4. Set prev_node → next = NULL;
5. Exit.

at the middle



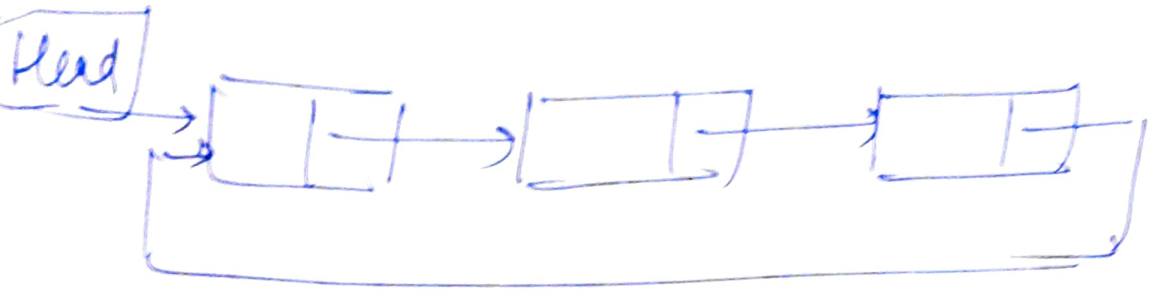
1. Start.
2. Let there be a linked list. Containing n nodes.
3. traverse the ~~array~~ linked list one before the element to be deleted. ~~Set~~
4. Set this node as prev_node.
5. Set next_node = prev_node → next → next;
6. Set prevnode → next = nextnode;
7. Exit.

5 Set prevnode → next =
current = node

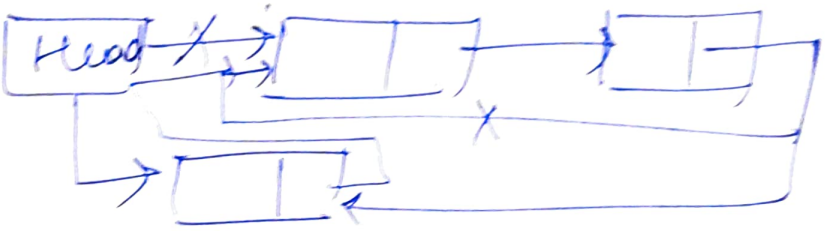
6. Set prevnode → next
= current_node →
next;

7 - Exit.

Circular linked list

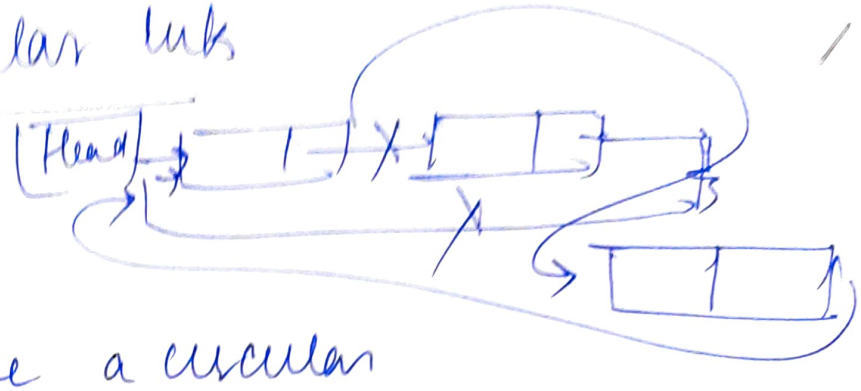


Starting of cir.



1. Start
2. Let there be a linked list of n nodes.
3. Create a new node 'temp'
4. Set temp → info = value;
5. Traverse to last of the linked list and name this node as last_node.
6. Set last_node → next = temp;
7. Set temp → next = head;
8. Set head = temp;
9. Exit.

end of circular link



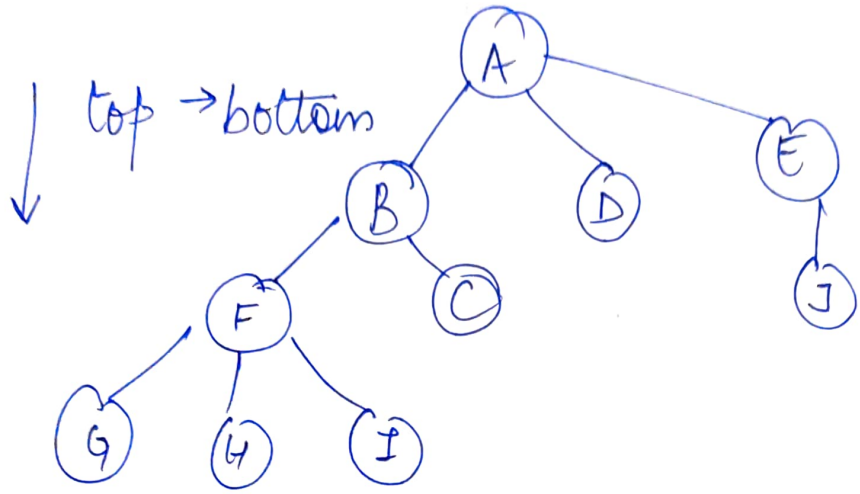
1. Start
2. Let there be a circular linked list of nodes n.
3. ~~Set~~ Create a new node 'temp'
4. Set temp → info = value;
5. traverse to the one before the last node & ~~save~~ name it as prev_node.
6. Set prev_node → next = temp.
7. Set temp → next = head;
8. Exit

middle of circular link

→ Same as normal

Trees

→ non-linear data structure.



node : each element in a tree

top node : → root node.

last level nodes → leaf nodes

parent node → node one above current node.

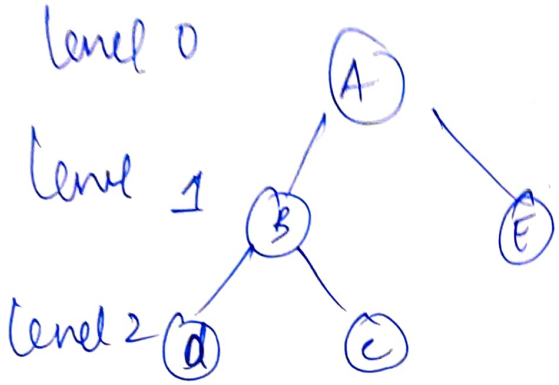
Child node → node one below current node.

siblings → nodes of same parent

degree of node → no. of child nodes of node.

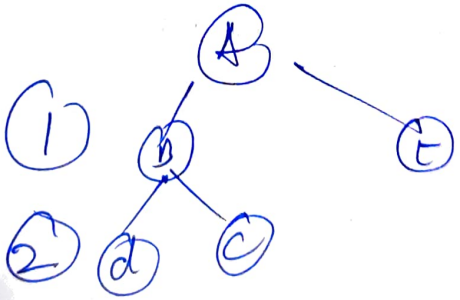
degree of tree → max degree of the node.

depth of a node → length of path from root to that node.



depth of d = level = 2.

height of node → no. of levels below the node

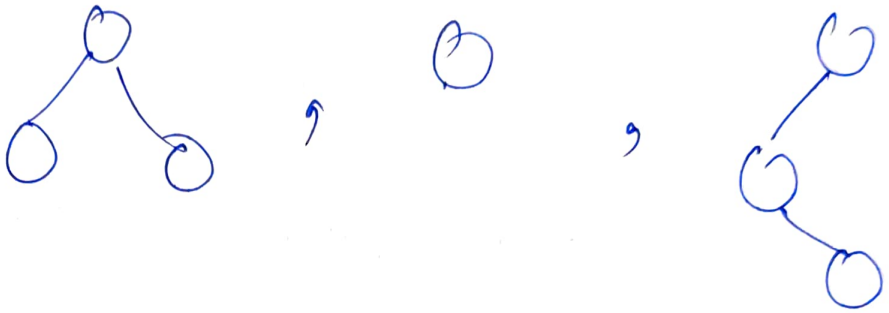


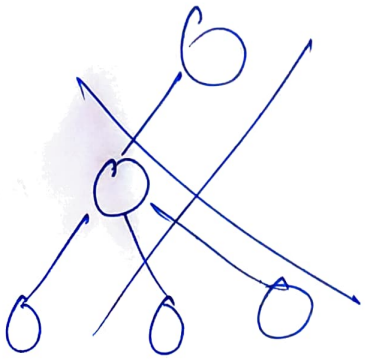
height of A = 2

Binary tree

→ Can have atmost 2 children

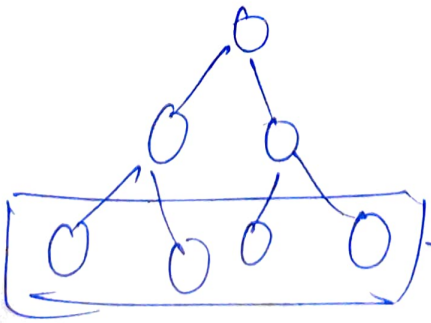
0, 1, 2





max nodes at level $l = 2^l$

Level 2 = 4



max no. of nodes of height h.

$= 2^0 + 2^1 + 2^2 \ldots 2^h$.

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

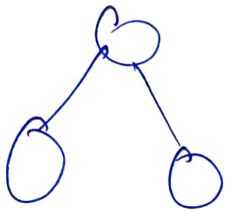
or

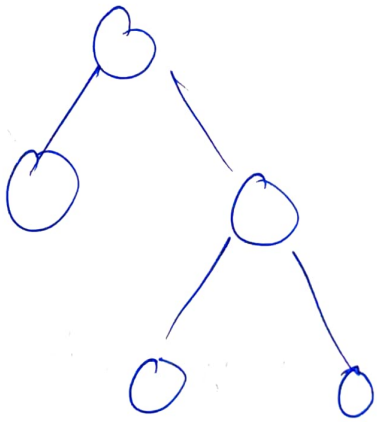
$2^{l+1} - 1$

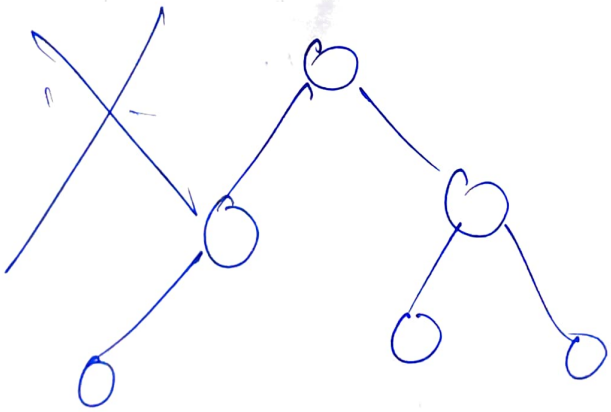
Types of binary tree

→ Strict / full binary tree

→ 0 or 2 children





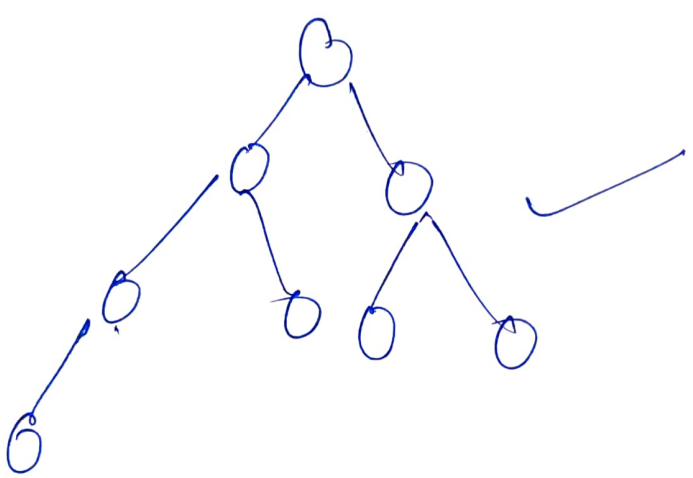


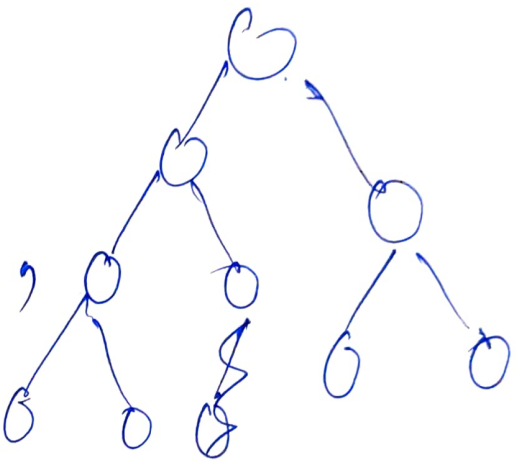
1 children

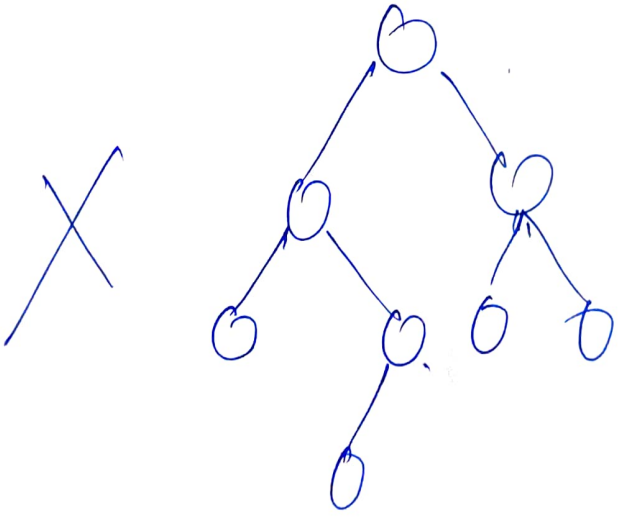
Complete binary tree

→ all levels are completely filled except last level

→ nodes should be as left as possible







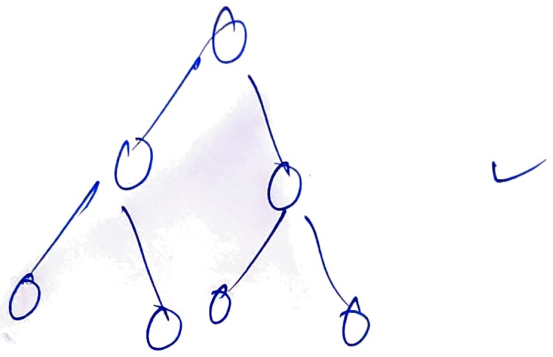
left side empty

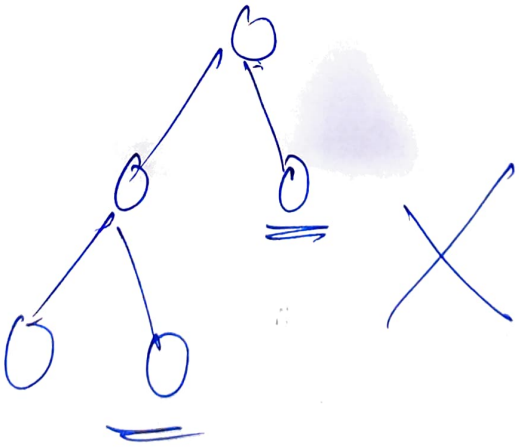
Perfect binary tree

→ all leaf nodes at same level

→ and completely filled.

↳ each node should have 2 child





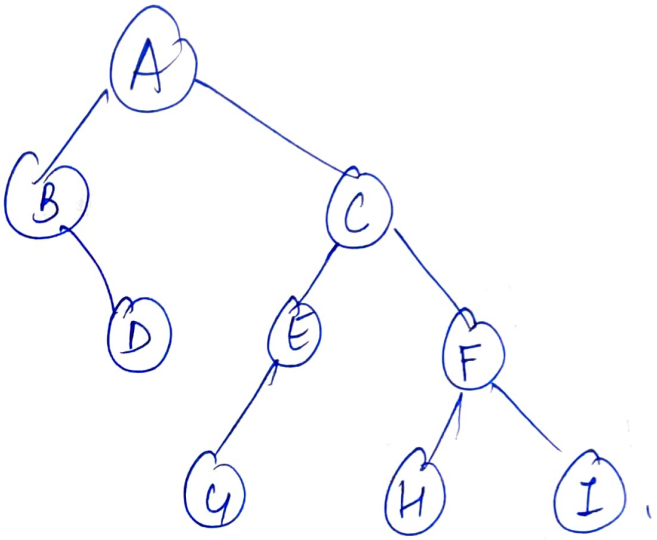
Traversal Binary tree

Inorder ⟶ L D R

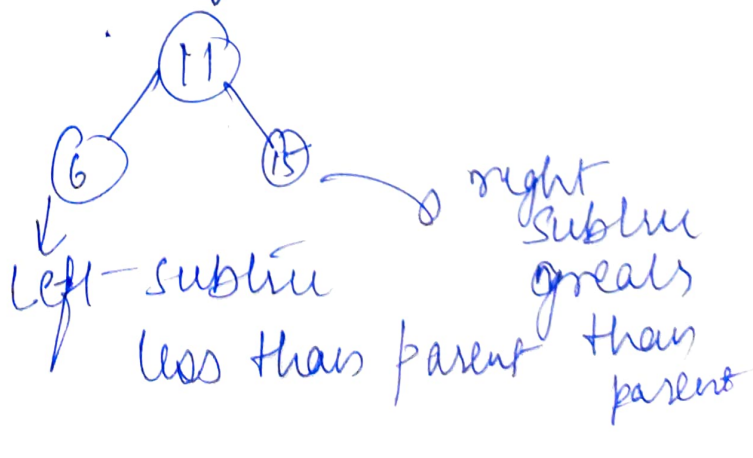
Preorder ⟶ D L R

Postorder ⟶ L R D.

Ques



Insertion in Binary search tree



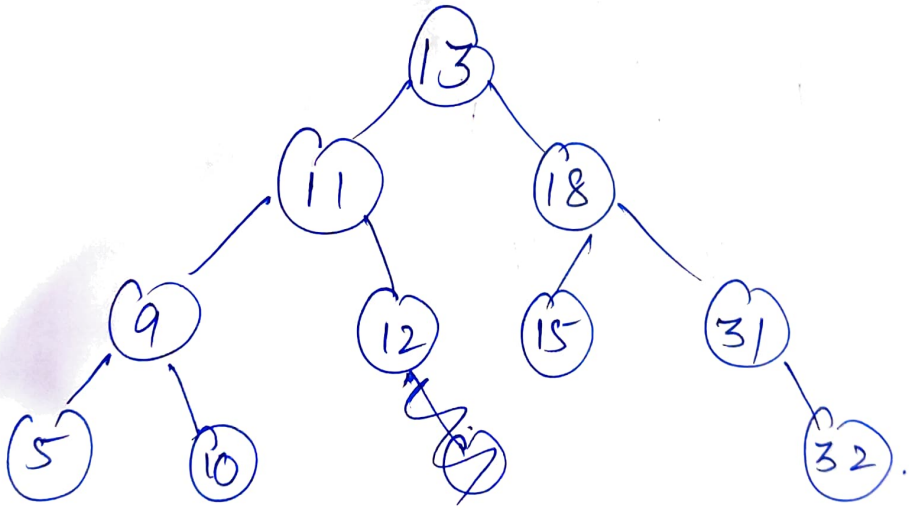
Ques 11, 6, 8, 19, 4, 10, 5, 17, 43, 49, 3)

Deletion in B.ST

Case 01 → no child (leaf node)

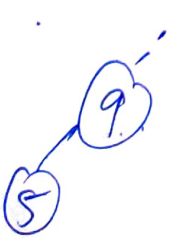
Case 2 → 1 child.

Case 3 → 2 child.

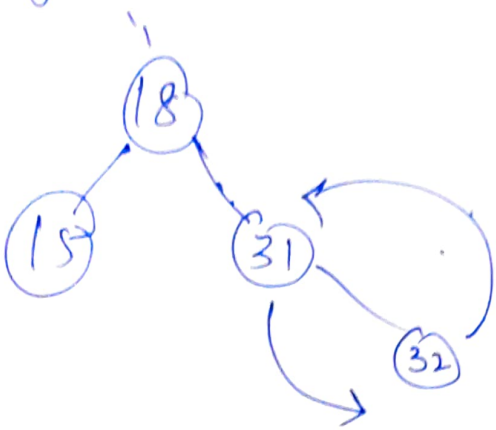


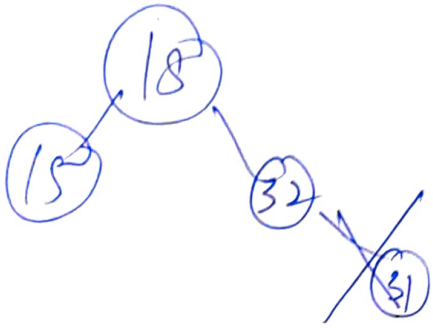
→ deleting 10.

normally remove the link



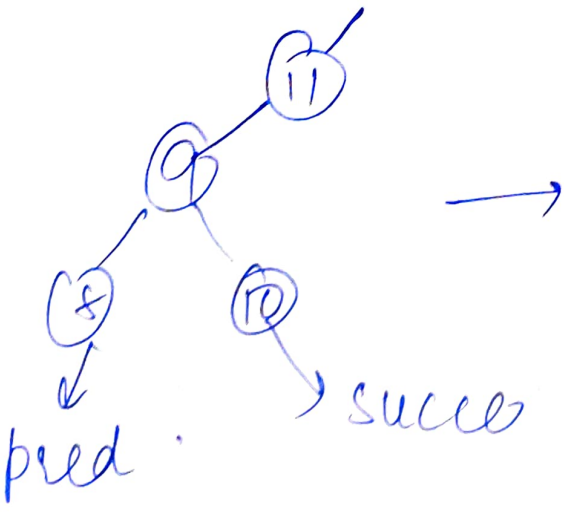
deleting (31)

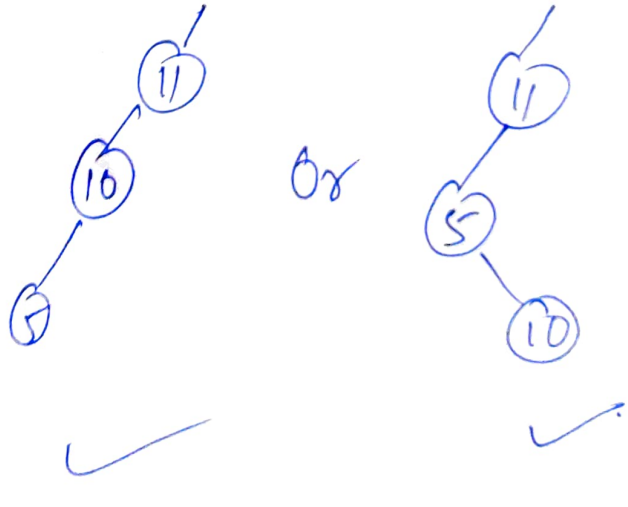




→ deleting (9) (case 3)

(inorder successor
or
inorder predecessor.





AVL tree

→ It is a B.S.T

→ $H_L - H_R = 0, 1, -1$ only.
for each node

$H_L - H_R$ = balance factor.

Rotations

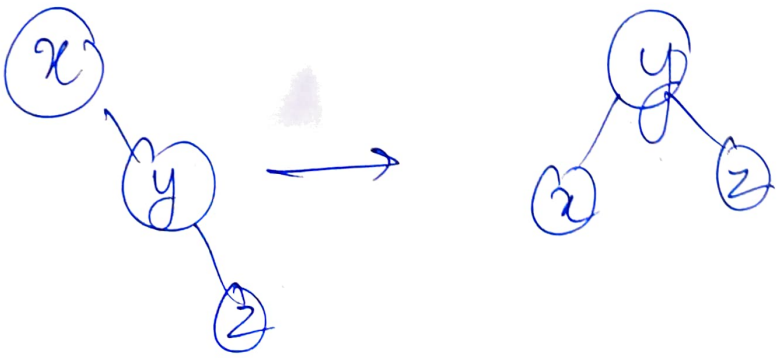
→ left rotation

→ right rotation

→ left ~~rotate~~ right rotation

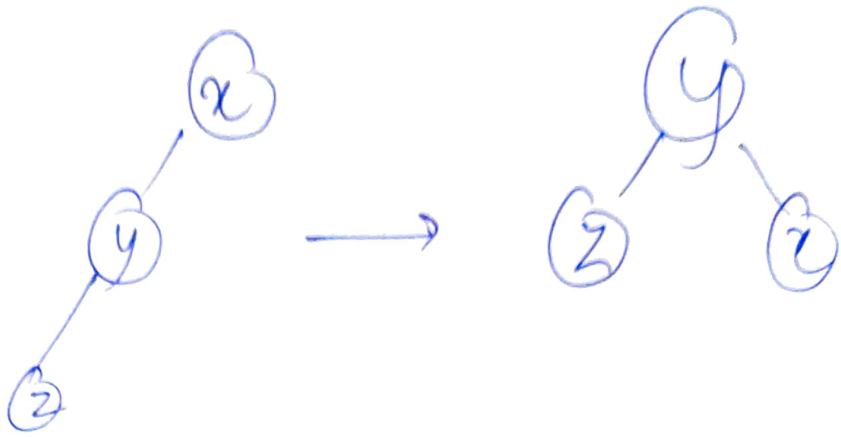
→ right left rotation

L.

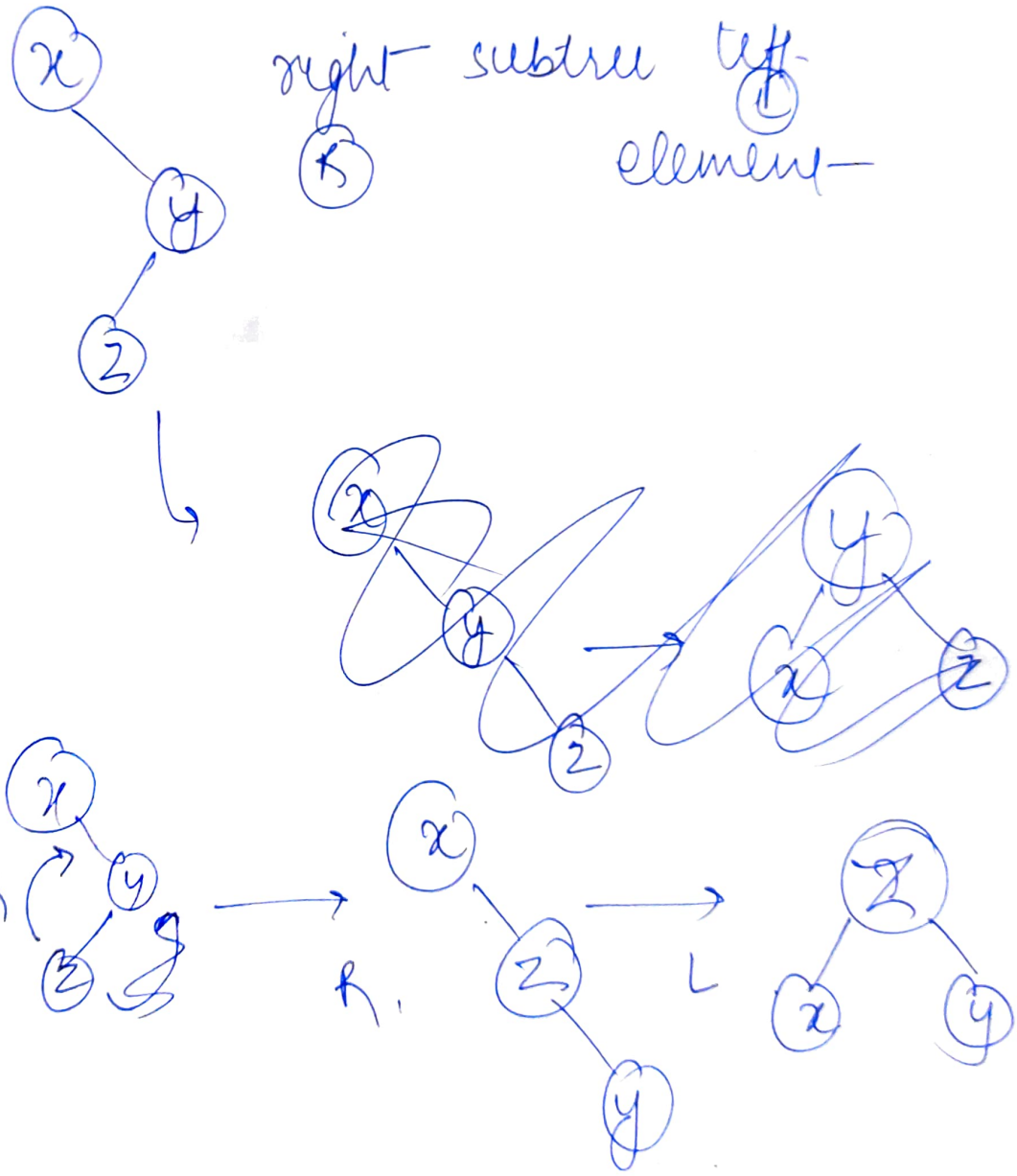


median element will go up

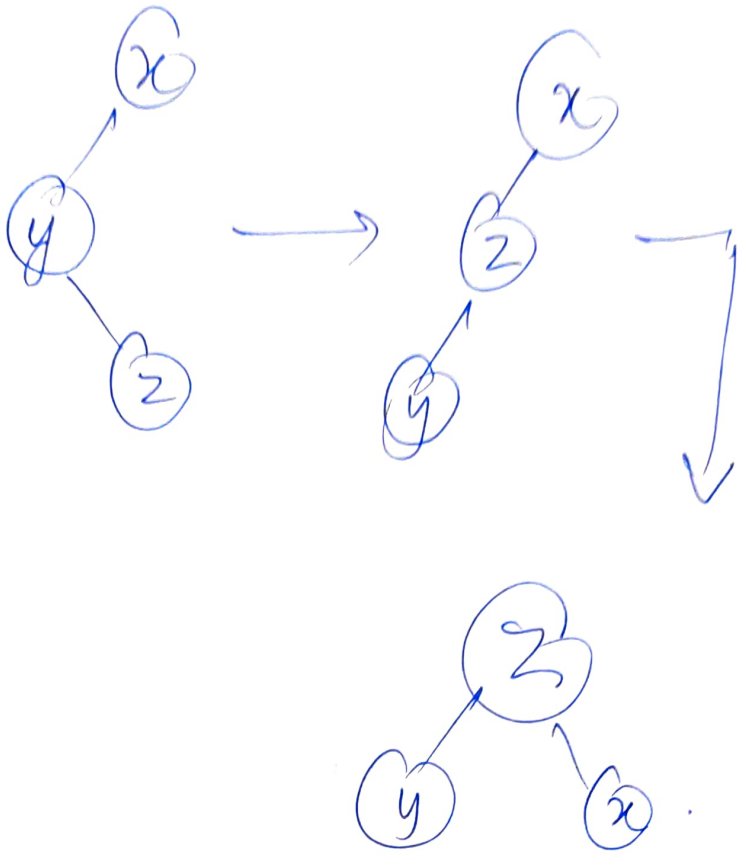
R


x
y
z
y
z
x

RL


x
y
z
right subtree left-
element-
R
1
x
y
z
y
x
z
x
R
y
z
R.
x
z
y
L
z
x
y

LR



Q. 14, 17, 11, 7, 53, 4, 13, 12, 8, 60, 19, 16, 20.

Heaps → max heap

parent node is greater than child node

types min heap, max heap

~~max heap~~ min heap

parent node is smaller than child node

Binary Search

1. Start.
2. Let there be a sorted array 'a'
3. Set beg = 0, end = n-1.
4. Set mid = (beg + end) / 2.
5. [loop] while (beg <= end)
6. if item < mid
   Set end = mid.
   else if item > mid
   Set beg = mid.
   else (item = mid)
   print item found "& exit
7. mid = (beg + end) / 2.
8. [end loop]
9. Exit.

Linear Search

1. Start.
2. Input the item to be searched 'ITEM'.
3. [Loop], I = 0, while I < n.
4. if a[I] = ITEM.
   print item found at 'I'.
   and Exit
5. If.
   print not found
6. Exit

Sorting.

Bubble Sort-

1. Start
2. Let there be an array 'a' with 'n' elements
3. Set- I = J = 0.
4. [Outer loop] while I <= n-1. / J < n
5. [Inner loop] while J <= n-1. / J < n
6. [if] a [J+1] < a [J]
7. swap a [J+1] , a [J]
8. [end if]
9. Set- J = J+1.
10. [exit- inner loop]
11. Set I = I+1
12. [exit- outer loop]
13. Exit.

Selection Sort

1. Start.
2. Let there be an array 'a' with n elements
3. Set I = 0, MIN = 0.
4. [outer loop] while I < n.
5. Set J = I.
6. [inner loop] while J < n.
7. [If] (a [J] < a [MIN])
   Set MIN = J + 1.
6
7. [End If]
   J = J + 1
8. [exit inner loop]
9. Swap a [I] with a [MIN]
10. Set I = I + 1.
11. [Exit outer loop]
12. Exit.

Insertion Sort

1. Start
2. Let there be an array 'a' with 'n' elements.
3. Set I = 0.
4. [outer loop] while I < n.
5. Set J = I.
6. [inner loop] while J >= 0.
7. [if] (a [J+1] < a [J])
   swap a [J+1], a [J]
8. [End if]
9. Set J = J - 1
10. [exit inner loop]
11. Set I = I + 1
12. [exit outer loop
13. Exit.