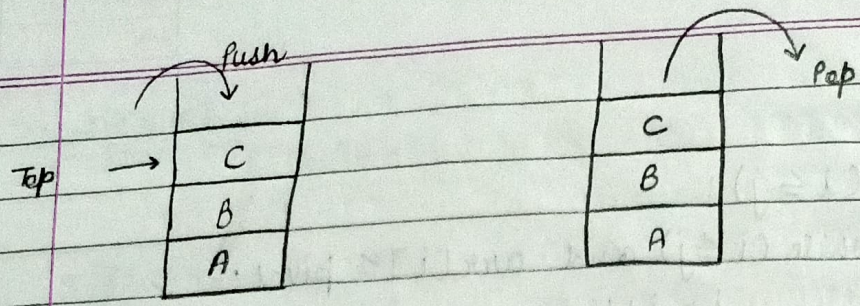
Ques: What do you mean by stack data structure. Write all its operations, algorithm and implementation.

Ans: A stack is a type of data structure that works like a pile of plates. You can only add and remove plates from the top of pile, not from middle and bottom. The order may be LIFO (Last In First Out) or FILO (First in Last Out).




* Key operations on Stack Data Structure:

- **Push()**: Adds an element to the top of the stack. If the stack is full, then it is said to be an overflow condition.

For example: Stack = [ ]
Stack.append(10)
Stack.append(20)
print(Stack)

- **Pop()**: Delete the top most element of the stack.
Time Complexity: O(1)

For example: print(Stack.pop())

- **Peek()**: Returns a reference to the topmost element of stack. Time Complexity O(1). It return element without removing from stack.

- **isEmpty()**: This is method that returns true if the stack is empty else it returns false.

- **Isfull()**: This is a method that returns true if stack is full else it returns false.

Page No.:
Date:

→ Underflow and Overflow are conditions related to stack operations.

- Underflow: It occurs when you try to pop an element from the empty stack.

- Overflow: Overflow occurs when you try to push an element onto full stack.

* Infix, Prefix & Postfix ∫ imp.

a+b ⇒ [ ] - evaluate
- conversion

a + b → Infix
↙ operand ↘ operand
operator

+ab → Prefix / Reverse Polish Notation
ab+ → Postfix / Polish Notation / Suffix Notation.

⇒ Priority:

[ ], { }, ( )

^ R-L

* / L-R

+ - L-R

* Tie breaker: When an operand lies b/w 2 operators that have the same priority.

* Delimeters: Subexpression within delimeters is treated as a single operand, independent from the remainder of the expression.

eg: (a+b) * (c-d) / (e-f)

eg:

| Infix | Postfix | Prefix |
|---|---|---|
| A+B | AB+ | +AB |
| (a+b)*(c-d) | AB+CD-* | +AB-CD |
| a-b/(c*d^e) | Abcde^*/- | -A/B*C^de |

⇨ Algorithm to convert infix to post fix.

1. Initialize

* Stack = [ ] // create an empty stack 'S' to hold operators.
* Postfix list = [ ] // " " " list postfix for the o/p

2. Scan the infix exp. from left to right.

* If the character is a operand, add it to the postfix list.
* If the character is a opening paranthesis '(', push it onto the stack.
* If the character is a closing paranthesis ')' pop from the stack & add to postfix until an opening paranthesis is encountered. Discard both paranthesis.

* If the character is an operator : * while there is an operator at the top of the stack with greater precedence or the same precedence and the operator is left associative, pop it to postfix.
* Push the current operator on the stack.

3. After Scanning the I/P pop all remaining operators from the stack to postfix.

4. End

eg: A+(B-C)

ABC-+

| Input | Stack | Postfix |
|---|---|---|
| A | | A |
| + | + | A |
| ( | +( | A |
| B | +( | AB |
| - | +(- | AB |
| C | +(- | ABC |
| ) | + | ABC- |

| Input | Stack | Past |
|---|---|---|
| A | | A |
| + | + | A |
| ( | +( | AB |
| - | +(- | ABC |
| C | +(- | ABC |
| ) | + | ABC- |

eg Convert the Expression K+L-M*N+(O^P)*W/U/V*T+Q into postfix using Stack.

Sol:

| Input | Stack | Postfix |
|---|---|---|
| K | | K |
| + | + | K |
| L | + | KL |
| - | - | KL+ |
| M | - | KL+M |
| * | -* | KL+M |
| N | -* | KL+MN |
| + | + | KL+MN*- |
| ( | +( | KL+MN*- |
| O | +( | KL+MN*-O |
| ^ | +(^ | KL+MN*-O |
| P | +(^ | KL+MN*-OP |
| ) | + | KL+MN*-OP^ |

| Input | Stack | Postfix |
|---|---|---|
| * | +* | KL+MN*-OP^ |
| W | +* | KL+MN*-OP^W |
| / | +/ | KL+MN*-OP^W* |
| U | +/ | KL+MN*-OP^W*U |
| / | +/ | KL+MN*-OP^W*U/ |
| V | +/ | KL+MN*-OP^W*U/V |
| * | +* | KL+MN*-OP^W*U/V/ |
| T | +* | KL+MN*-OP^W*U/V/T |
| + | + | KL+MN*-OP^W*U/V/T*+ |
| Q | + | KL+MN*-OP^W*U/V/T*+Q |
| | | KL+MN*-OP^W*U/V/T*+Q+ |

Ques K*L+M-N(R*O^P)+W/U-V*T/Q into postfix.

Sol:

| Input | Stack | Postfix |
|---|---|---|
| K | | K |
| * | * | K |
| L | * | KL |
| + | + | KL* |
| M | + | KL*M |
| - | - | KL*M+ |
| N | - | KL*M+N |
| ( | -( | KL*M+N |
| R | -( | KL*M+NR |
| * | -(* | KL*M+NR |
| O | -(* | KL*M+NRO |
| ^ | -(*^ | KL*M+NRO |
| P | -(*^ | KL*M+NRO^P |
| ) | - | KL*M+NRO^P*- |
| + | + | KL*M+NRO^P*-1*- |

| Input | Stack | Postfix |
|---|---|---|
| W | + | KL *M* + NROP* ^ * -W |
| / | +/ | KL*M+ NROP^ *-W |
| U | +/ | KL*M+NROP^ *- UU |
| - | - | KL*M+NROP^ *-W/+ |
| V | - | KL*M+NROP^ *-W/+V |
| * | -* | KL*M+NROP^*-WU/+V |
| T | -* | KL*M+NROP^*-WU/+VT |
| / | -/ | KL*M+NROP^*-WU/+VT* |
| O | -/ | KL*M+NROP^*-WU/+VT*O |

KL*M+NROP^*-WU/+VT*O/-

---

⟹ Algorithm to convert Infix to Prefix using stack:-

1. Reverse the infix expression:
* Start by reversing the I/P infix expression.
* Swap paranthesis: Change each '(' to ')' & vice versa.
2. Convert the reversed infix to postfix:
* Use infix to postfix conversion algo. on the modified infix expression (from Step 1).
3. Reverse the postfix expression.
* Finally reverse the postfix expression obtained in Step 2 to get the prefix expression.

Eg: A/B/C+D(E^F^G)+H-I*J/K-L*M

Reverse: M*L-K/J*I-H+)G^F^E(D+C/B/A

| Input | Stack | Postfix |
|---|---|---|
| M | | M |
| * | * | M |
| L | * | ML |
| - | *- | ML* |
| K | - | ML*K |
| / | -/ | ML*K |
| J | -/ | ML*KJ |
| * | -/* | ML*KJ |
| I | -/* | ML*KJI |
| - | -- | ML*KJI*/ |
| H | -- | ML*KJI*/H |
| + | --+ | ML*KJI*/H |
| ) | --+) | ML*KJI*/H |
| G | --+) | ML*KJI*/HG |
| ^ | --+)^ | ML*KJI*/HG |
| F | --+)^ | ML*KJI*/HGF |
| ^ | --+)^^ | ML*KJI*/HGF |
| E | --+)^^ | ML*KJI*/HGFE |
| ( | --+ | ML*KJI*/HGFE^^ |
| D | --+ | ML*KJI*/HGFE^^D |
| + | --++ | ML*KJI*/HGFE^^D |
| C | --++ | ML*KJI*/HGFE^^DC |
| / | --++/ | ML*KJI*/HGFE^^DC |
| B | --++/ | ML*KJI*/HGFE^^DCB |
| / | --++// | ML*KJI*/HGFE^^DCB |
| A | --++// | ML*KJI*/HGFE^^DCBA |

END: ML*KJI*/HGFE^^DCBA//++--

Reverse: --++//ABCD^^EFGH/*IJK*LM Ans.

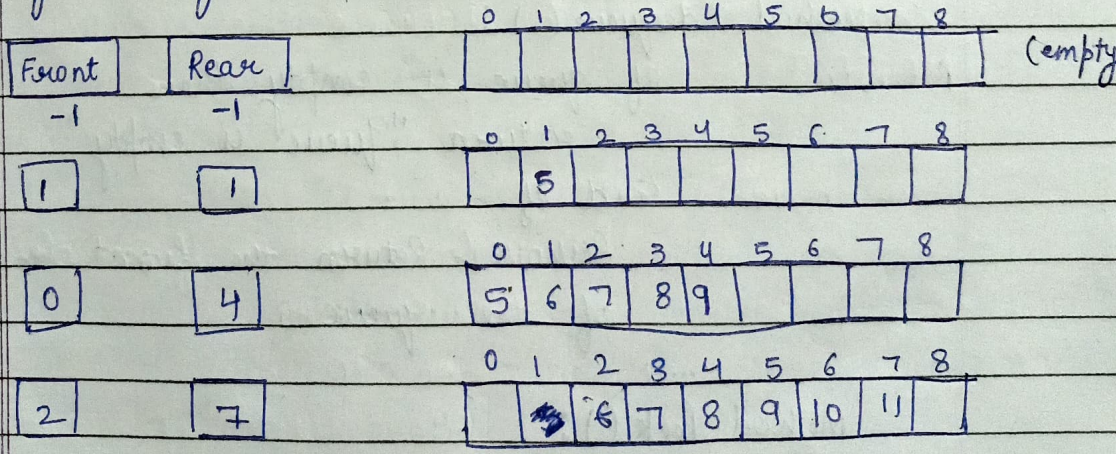
## Queue Data Structure : FIFO Principle

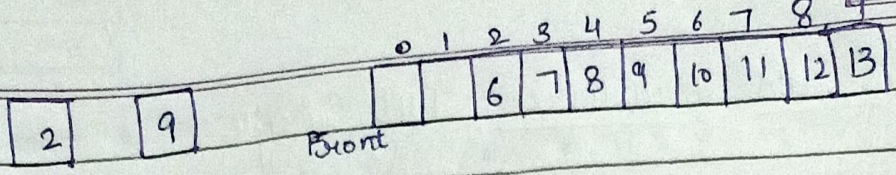
It is a type of linear ds in which the insertion of the elements takes place at one called as 'rear' & the deletion takes place at other end called as 'front' end. eg: waiting for a movie ticket, Rail ticket, Bank etc.

There are different types of queue:
- Linear Queue -> Time Comp of enqueue - O(n)
- Circular Queue → " " " all op - O(1)
- Priority Queue

Operations of Queue:
- enqueue - to insert an element
- dequeue - to delete an element
- Peek - to access top element without deleting it.
- is_empty - check whether queue is empty or not
- is_full - check whether queue is full or no

Representation of Queue
- By using arrays → Best for searching
- By using linked list → Best for insertion (dynamic)






- If it is empty then we can't perform dequeue & this situation is called as underflow. Here we can't perform peek operation as well.

- If queue is full & we still perform enqueue operation it is called as overflow.

# Pseudocode

```
class Queue:                          → constructor
    Method __init__(self):
        // Initialize an empty list to represent
           the queue
        queue = empty list

    Method enque(item):
        // add an element to queue
        append item to queue

    Method dequeue():
        If queue is empty then
            return "queue is empty"
        End If
        Remove & Return the front element
        of the queue.

    Method Peek():
        If queue is empty then return "queue is empty
```

End If
Return the front element from queue.

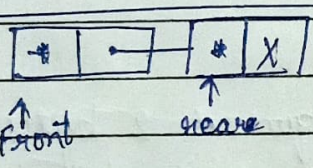
Method Is_empty ():
Return True if the length of queue is 0, otherwise false.

Method is_full ():
Return True if the length of queue is greater than or equal to max_size otherwise false.

Method display ():
If queue is empty then return 'queue is empty'.
End if
Return the current content of queue.

Limitation: Fixed size, Space insufficiency.

linked list queue



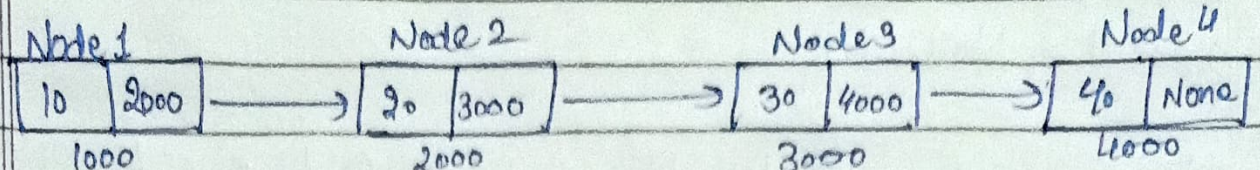

insertion done at rear & deletion done at front.

J sirf ik element a ta chi front te rear nu point karu.

# enqueue adds an element to the rear of the queue. If the queue is empty, the added element becomes the the 1st element of the equeue

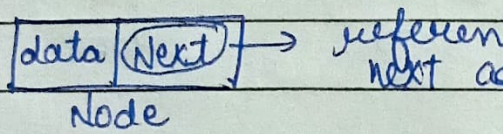
# dequeue removes the element from the front of the queue. If the queue is empthy, this op. cannot be performed.

# Linked list : elements randomly stored

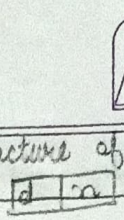



Array: Widely used, store elements in contiguous memory but the problem is when we add elements in between any elements then the time complexity of shifting all the elements is O(n).
And if there is no space in array than array copy its all the elements to other array & then add new elements.
So due to these reasons array is less efficient.

So we use linked list for the insertion & the deletion of elements easily.



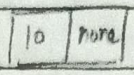
Structure of node



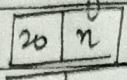

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
```

# Creating Nodes

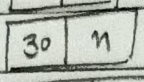
node1 = Node(10) → value for data



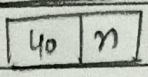

node2 = Node(20)




node3 = Node(30)




node4 = Node(40)

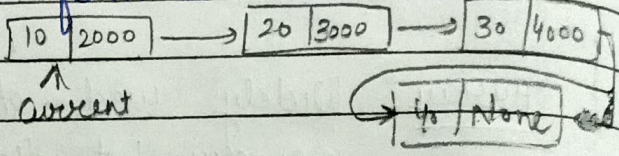



# Connecting nodes to form a linked list

node1.next = node2

node2.next = node3

node3.next = node4

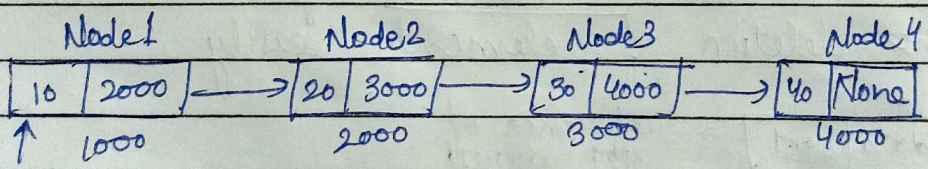



# Printing the linked list

head / current = node1 → 1000 ~~2000 3000 4000 5000~~ none

```
while current is not not:
    print(current.data, end = "->")   10->20->30->40
    current = current.next → 2000 3000 4000 none
print("None")  → none
```

# Traversing in linked list:




current = head (1000)

Same Code upr wala.

ab age → 1 node krde a current = head

# A linked list is a linear data structure that includes a series of connected nodes. Here, each node stores the data and the address of the next node

## Types of linked List.

- Singly Linked List
- Doubly Linked List
- Circular Linked List → circular Singly, circular doubly.

# Difference b/w Linked List & Array.

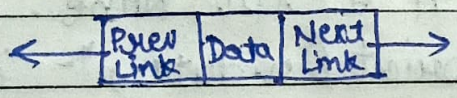
| | Array | Linked List |
|---|---|---|
| Strength | • Random Access (fast search tym)<br>• Less memory needed per element. | • Fast insertion/Deletion time<br>• Dynamic Size<br>• Efficient memory allocation utilization |
| Weakness | • Slow Insertion/Deletion time<br>• Fixed Size<br>• Inefficient memory allocation/utilization. | • Slow Search time<br>• More memory needed per node as additional storage required for pointers. |

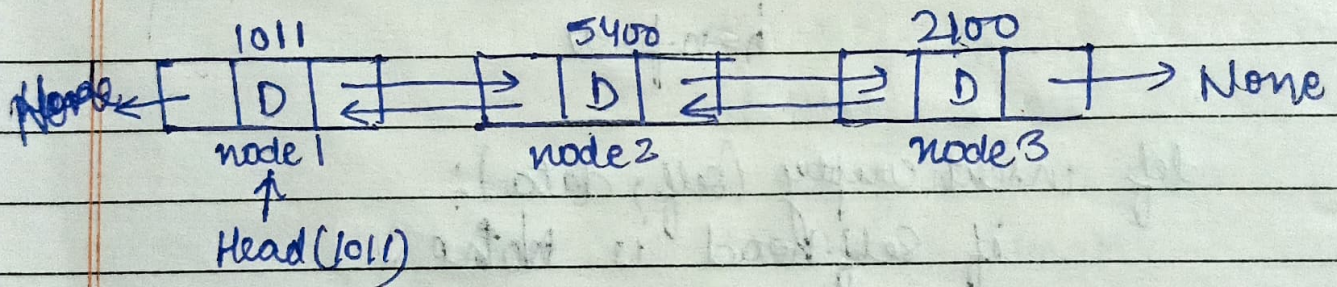
# Singly Linked List:

A singly linked list is a data structure used to organize and store a collection of elements, known as nodes. Each node contain two main components:

# Doubly Linked List

Linked list is the collection of nodes in which each nodes are connected through links & in the singly linked list each node contains the data field and only one link that is the link of next node but in doubly linked list each node contains the data field and two links one is the link of the next node and ~~out~~ another is the link of the previous node. So doubly linked list is the collection of nodes in which every node contains data field and link of the next node as well as the link of the previous node.



# Header Linked List:

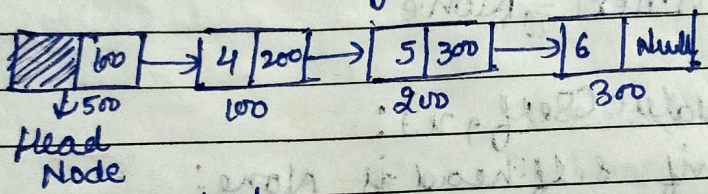
Date
Page

→ Special kind of linked list.
→ Contain extra node at beginning known as header node.
→ Contain important info regarding linked list.

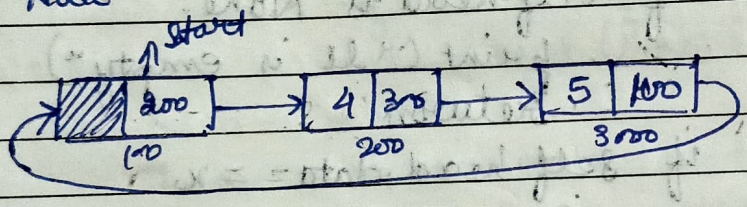
Categories of Header linked list

1 → Grounded Header linked list
2 → Circular Header linked list
3 → Two way Header linked list
4 → Circular two way header linked list.

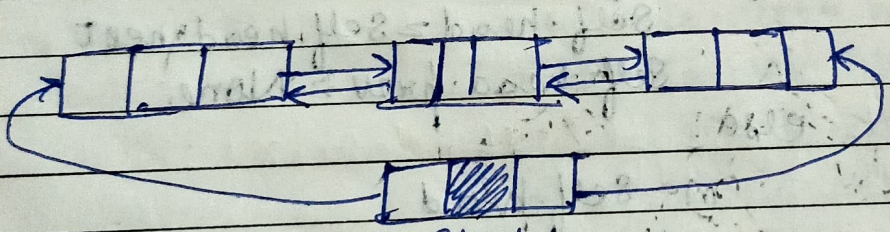
1.




2




3.




4.