LINKED LIST:-

A linked list consists of nodes where each node contains a data element and the reference (link) to the next node in the list. First node of Linked list is called "head". Last node of linked list is called "tail".

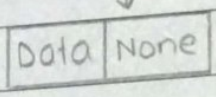
Scanned with CamScanner

=> Implementing Stacks Through Linked List

First making a class node

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

→ In order to create nodes in a linked list we first create a class node with attribute data



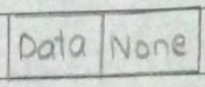
Now making a class stack

```
class linkStack( ):
    def __init__(self):
        self.head = None
        self.tail = None
```

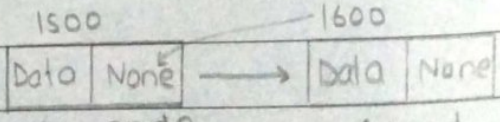
→ Now in order to implement stacks in linked list we now creating a class stack and intializing head and tail to None

Now writing a method of push

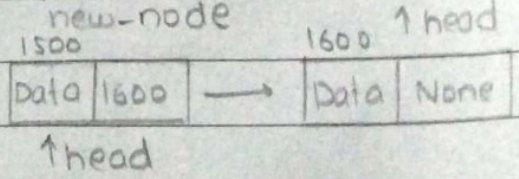
```
def push(self, data):
    new_node = node(data)
```




```
new_node.next = self.head
```




```
self.head = new_node
```



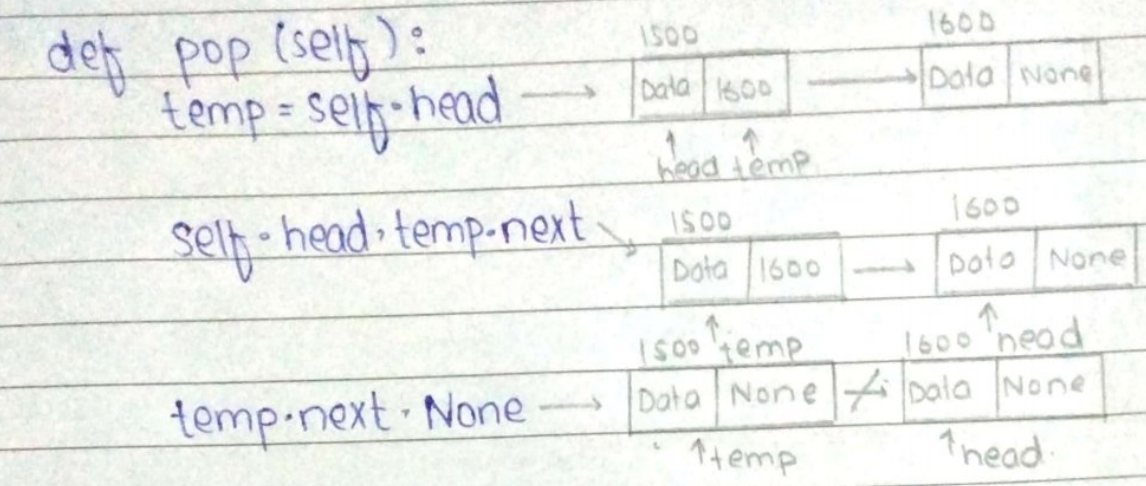
* head is always the first node of linked list
* In push function we create nodes at head

Scanned with CamScanner

Date:

Now writing a function of pop

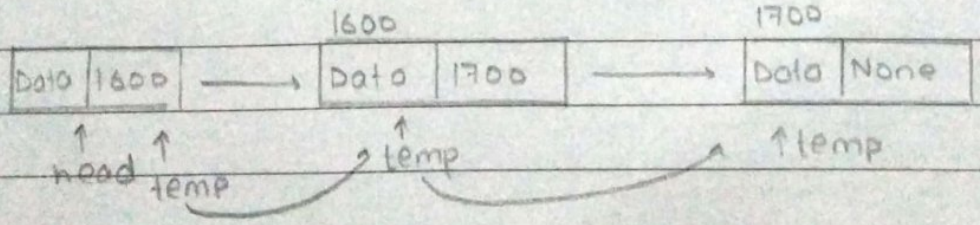
```
def pop(self):
    temp = self.head
    self.head = temp.next
    temp.next = None
```



⇒ In pop function we delete the first node by disconnecting its link to second node.

Now writing a function of show-all

```
def show-all(self):
    temp = self.head
    If self.head is None:
        Print ("List is Empty")
        return

    While temp:
        Print (temp.data)
        temp = temp.next
```



In order to print our linked list we first make a temporary variable equal to head and then traverse it to last.

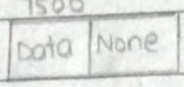
Scanned with CamScanner

=> Implementing Queues Through Linked List :-

First making a class node

```
class node( ):
    def --init-- (self, data):
        self.data = data
        self.next = None
```

→ First we make a class node in order to create nodes in a linked list with attribute data i.e:



Now making a class queue

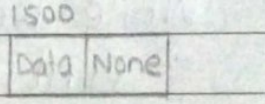
```
class Queue ( ):
    def --init-- (self, dat):
        self.head = None
        self.tail = None
```

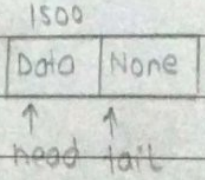
→ In order to implement queue through Linked list we now creating a class queue and initializing head & tail to None

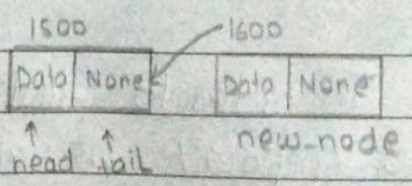
Now writing a method of enque

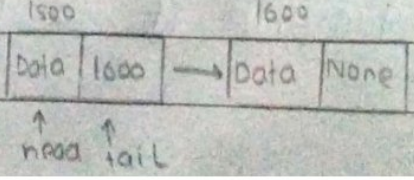
```
def enqueue(self, data):
    If self.head is None:
        new_node = node(data)
        self.head = new_node
        self.tail = new_node
    else:
        new_node = node(data)
        self.tail.next = new_node
```

for first node

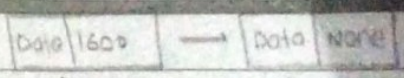








Date: 1600

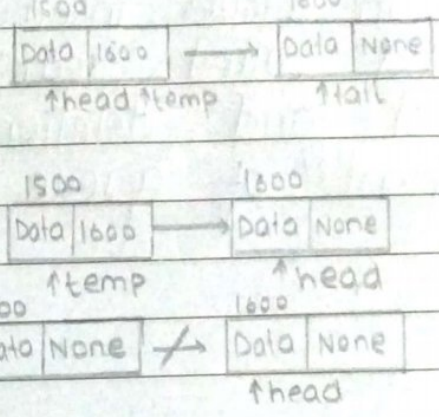
self.tail = self.tail.next → 

* Last node of linked list is "tail"

* In enqueue function we add nodes at tail.

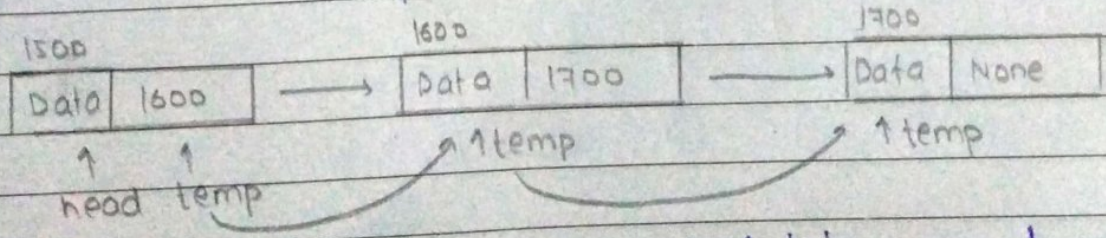
Now writing a method of dequeue

```
def dequeue (self):
    temp = self.head
    self.head = temp.next
    temp.next = None
```



Now writing a method of show-all

```
def show_all (self):
    temp = self.head
    If self.head is None:
        Print ("List is Empty")
        Return

    While temp:
        Print (temp.data)
        temp = temp.temp.next
```



⇒ In order to print whole linked list we need to traverse whole linked list.

Scanned with CamScanner

Date: _____

CONCLUSION:-

=> In stacks both insertion and deletion can be performed at one end called "top" of the stack and "head" of the linked list.

=> In Queues insertion can be performed at one end called "rear" / "tail" of the linked list and deletion can be performed at "front" of queue / "head" of linked list.

LAB: 10

Using node class develop stack, queues.

singly connected linked list with following features

a. add nodes
b. Traverse all node starting from top node
c. search any key value in all node

d. Insert node between any two nodes.

STACK:

A list or collection with restriction that insertion can be performed at one end (head) and deletion can be performed also at same end (head).

Scanned with CamScanner

• STACKS :-

• STACK IMPLEMENTATION OF LINKED LIST :-

⇒ we will select the beginning of the linked list as the top of the stack.

↓

mtlb stack mai node hamesha beginning mai add hogi.

⇒ For push operation, a node will be inserted at the beginning / head of the linked list. bcuz we consider the beginning of the linked list as the top of the stack.

⇒ For pop operation, everytime the first node will be deleted.

=> Time complexity of adding a node at beginning is O(1)

=> Time complexity of removing a node at beginning is also O(1).

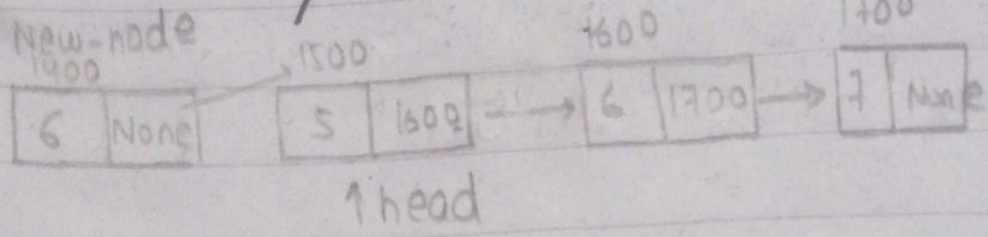
• NOTE:

- The code of the push( ) function must be similar to the code of inserting the node at beginning of singly connected linked list.

- The code of the pop( ) operation must be similar to the removing the node at the beginning of the singly connected linked list.

CLASS STACK :-

PUSH( ) / INSERTION_AT_BEGINNING

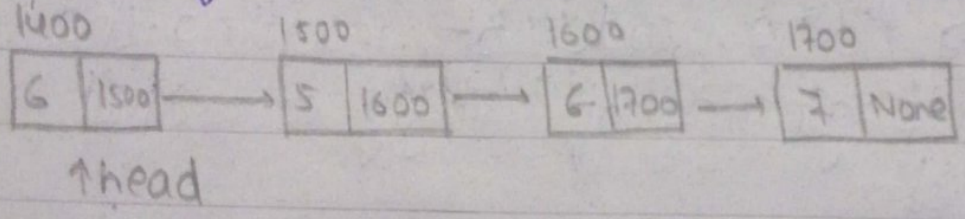



```
def push(self,data):
```

new_node = node(data) → 6 | None

new_node.next = self.head → linking

self.head = new_node → head updated



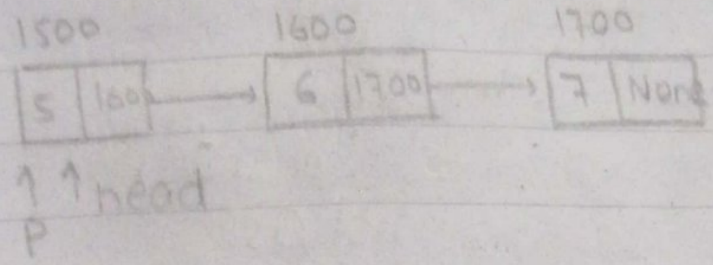
* head is always first node

* In stack, push( ) operation is similar to insertion at beginning.

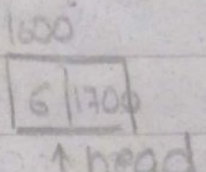
* Time complexity of push operation is O(1)

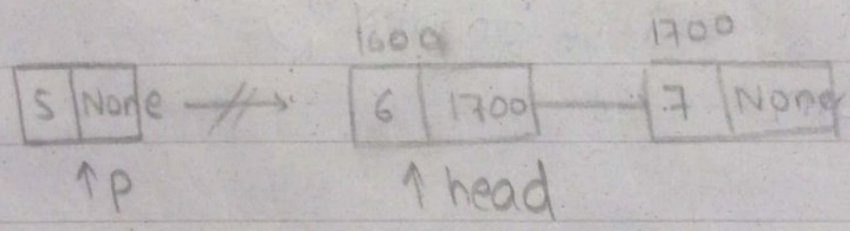
Scanned with CamScanner

POP / DELETION AT BEGGINING:-




```
def pop(self):

    P = self.head
    self.head = P.next
    P.next = None
```



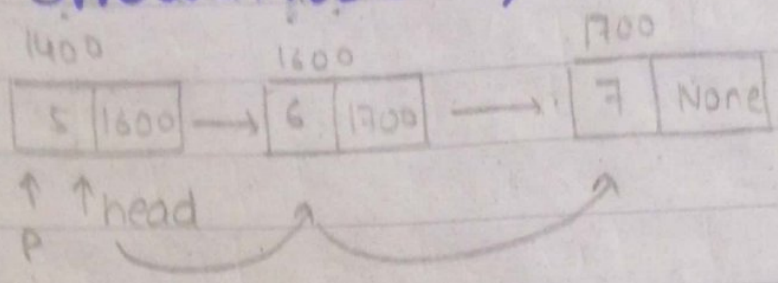


* deletion at beginning is just deleting the connection of first and second node.

* We first update our head.
* Then disconnecting links.
* Time complexity of pop operation is O(1)

Scanned with CamScanner

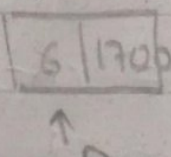
SHOW-ALL( ) / TRAVERSAL:-




```
def show_all(self):

    P = self.head

    While self.head is None:
        Print("List is Empty")
        return

    While P:
        Print(P.data)
        P = P.next
```

→ If list is empty self.head must be None

P = 1600 →



* Time-complexity of show_all function is O(n)

* We must have to traverse all nodes of linked list from head to last node.

Scanned with CamScanner

• QUEUES:-

QUEUE IMPLEMENTATION OF LINKED LIST:-

• **Queue:**

A list or collection with restriction that insertion can be performed at one end (tail) and deletion can be performed at other hand (head)

Operations:

1. enqueue()] → O(1)
2. dequeue()]
3. show_all() → O(n)

⇒ we will select the tail of the linked list as the top of the queue.

↓

mtlb queue mai node hamesha tail pr ke insert hogi

Scanned with CamScanner

=> For enque operation, a node will be inserted at tail of the linked list, because we consider the tail of the linked list as the top of the queue.

deque

=> For pop operation, first node will be deleted.

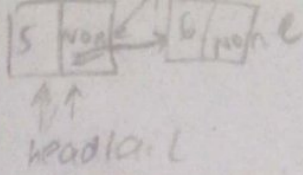
=> Time complexity of enque is O(1)

=> Time complexity of deque is O(1)

NOTE:-

- The code of enqueue operation must be similar to the insertion of node at tail.

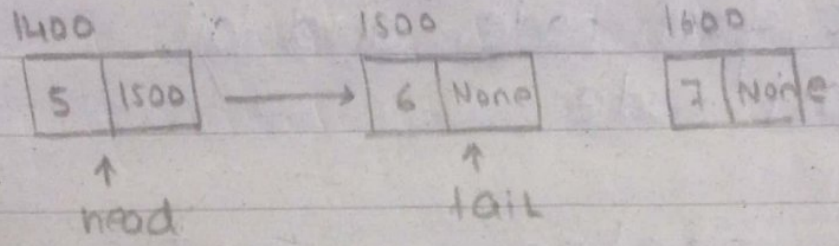
Scanned with CamScanner



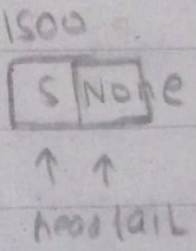
* The code of deque operation must be similar to the deletion of node at beginning.

CLASS QUEUE:-

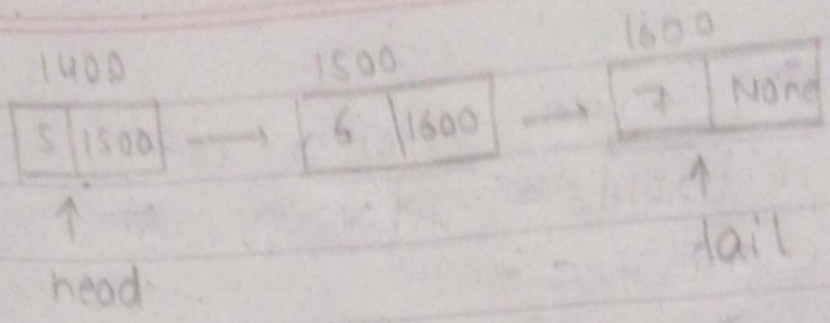
ENQUEUE / INSERTION AT TAIL :-




```
def enque(self,data):

    If self.tail is None:
        new_node = node(data)
        self.head = new_node
        self.tail = self.head

    else:
        new_node = node(data)
        self.tail.next = new_node
        self.tail = self.tail.next
```



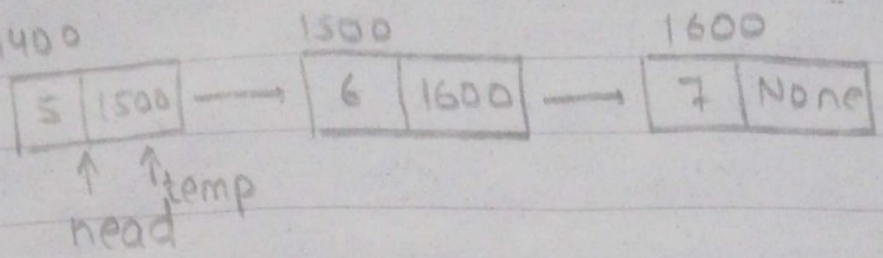
Scanned with CamScanner



- Time complexity of enqueue operation is O(1)
- head is always first node and tail is always last node
- In equeue operation, we insert nodes at tail.

Scanned with CamScanner

DEQUEUE:/DELETION AT BEGINNING:



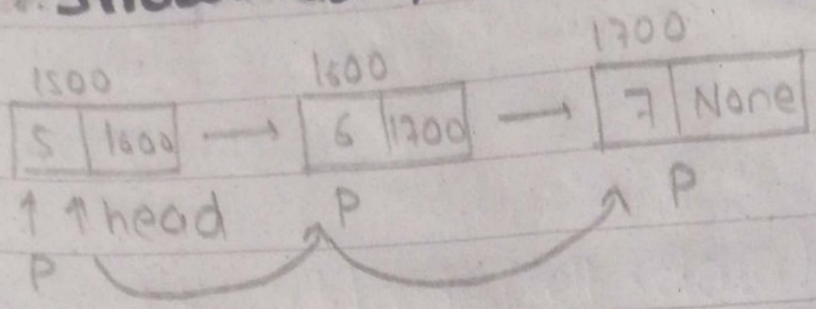

```
def deque(self):
```

temp - self.head → first make temporary variable
self.head = temp.next → update head
temp.next = None → disconnect 1st node and second node

=> Time complexity of deque operation is O(1)

=> We first make temp variable.
=> Then update our head
=> Then disconnecting first and second node links.

SHOW-ALL / TRAVERSAL:




```
def show_all(self):
    P = self.head
    If self.head is None:
        Print("List is Empty")

    While P:
    Print(P.data)
    P = P.next
```

If list is empty

* Time complexity of show_all function is O(n)

* We must have to traverse all nodes from head to tail.

Scanned with CamScanner

DINKED DIST WITH FEATURES:-

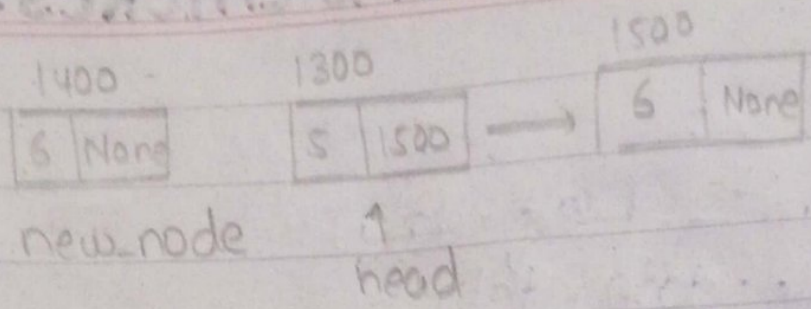
```
class node( ):
    def __init__(self, data):
        self.data = data
        self.next = None

class linked_list( ):
    def __init__(self):
        self.head = None
```

(a) add node

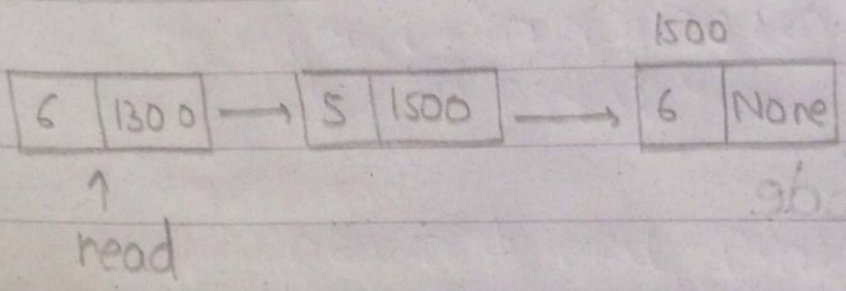
⇒ hum addnode head pai bhi kiasktay and tail pai bhi

⇒ function of add_node is same as insertion at beginning or insertion at end.

Scanned with CamScanner



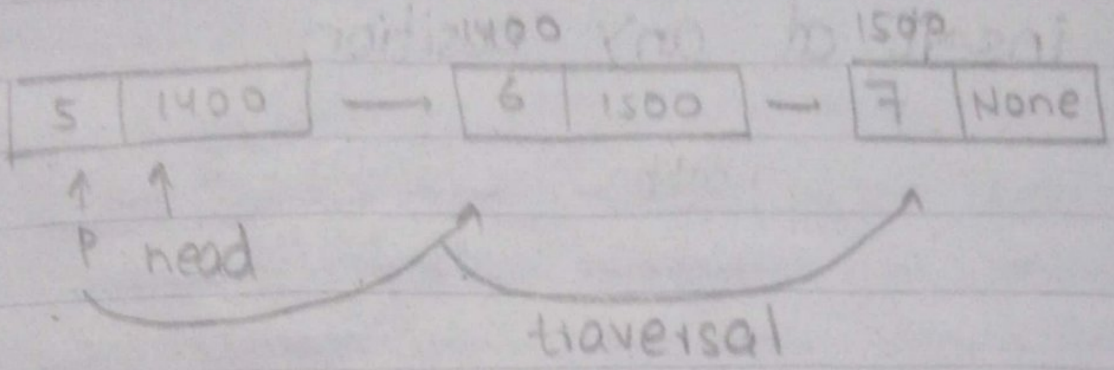

```
def addnode(self, data):
    new_node = node(data)
    new_node.next = self.head
    self.head = new_node
```



- head is always first node
- Time complexity of new_node is O(1)

Scanned with CamScanner

(b) Traversal

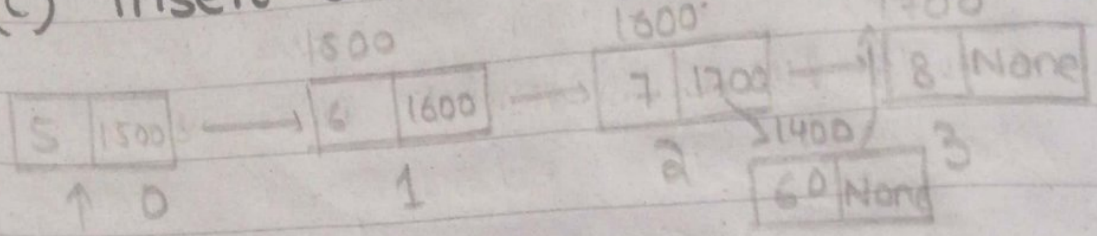



```
def traversal(self):
    P = self.head

    If self.head is None:
        print("List is Empty")

    While P:
        Print(P.data)
        P = P.next
```

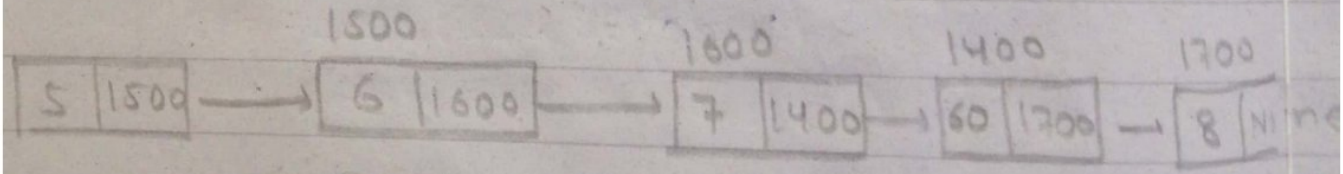
=> Time complexity of traversal is O(n).

Scanned with CamScanner

(c) insert at any position

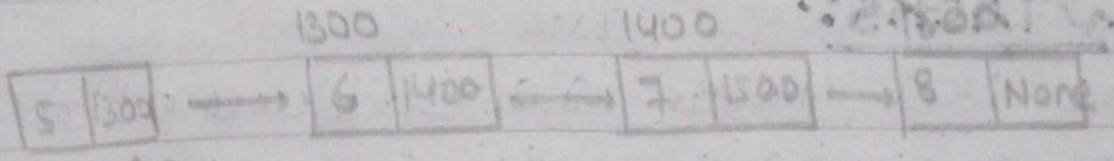



```
def insert_at_index(self, data, index):
    new_node = node(data)
    P = self.head
    for i in range(index-1):
        P = P.next

    new_node.next = P.next
    P.next = new_node
```



Scanned with CamScanner

(a) Search any key value in all node




```
def search(self,key):-
    P=self.head
    i=0
    While P:
        If key==P.data:
            print(f"{key} is present at
                  index {i}")
            i=i+1
            return
        else:
            P=P.next
    Print("key is absent").
        ↳ If key not found.
```


Scanned with CamScanner

LAB: 09

STACKS:

Stacks is a linear data structure which follows a particular order in which the operations are performed. The order may be LIFO (last in first out).

- The operations of stacks are:
- push operation
- pop operation.

```
Class stack( ):
    def __init__(self):
        self.container = list( )

    def push(self, data):
        self.container.insert(0, data)

    def pop(self, da):
        self.container.remove(self.containe
```

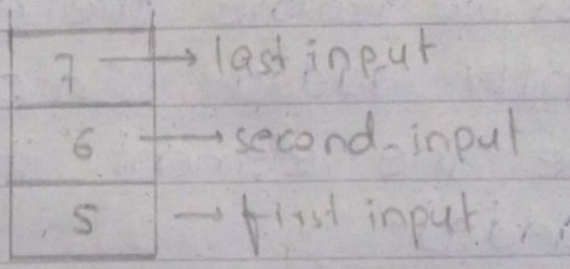

Scanned with CamScanner

```
def show_stack(self):
    return self.container
```

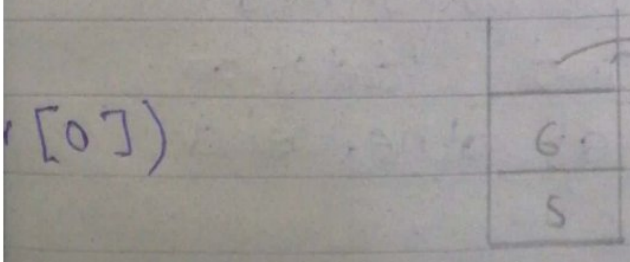
* stack is a linear data structure in which the insertion and deletion are allowed only at end, called the top of the stack.

OPERATIONS OF STACK:-

- **PUSH(DATA):** Insert data onto stack.



- **POP( ):-** Deletes the last inserted element from the stack.



Scanned with CamScanner

• QUEUE:

Queue is a linear data structure which follows a particular order in which the operations are performed. The order may be FIFO. (First in first out). A good example of queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In stacks, we remove the element the most recently added. In queues, we remove the element the least recently added.

Or

A list or collection with restriction that insertion can be performed at one end (tail) and deletion can be performed at other end (head).

Scanned with CamScanner

0
n=4 4 3 2 (1) <n-1
n-1 0 1 2 3

operations of queue:

* enque
* deque

* Insertion at one end and deletion at other end.

```
class queue():
    def __init__(self):
        self.container = list()

    def enqueue(self, data):
        self.container.insert(0, data)

    def deque(self):
        self.container.remov
        n = len(self.container)
        self.container.remove(self.container[n-1])
```

```
def show_queue(self):
    return self.container
```

• DYNAMIC SET:-

A set that supports insertion or deletion of elements. This data structure is frequently used in database access.

```
class dynamic_set():
    def __init__(self):
        self.set = list()

    def insert(self, data):
        self.set.append(data)

    def search(self, key):
        n = len(self.set)
        for i in range(n):
            If self.set[i] == key:
                Print("KEY IS PRESENT")
                return
```


Scanned with CamScanner

```
Print("KEY IS NOT PRESENT")

def delete(self):
    self.set.remove(self.set[0])

def maximum(self):
    n=len(self.set)
    for i in range(n):
        for j in range(0,n-i-1,1):
            if self.set[j]>self.set[j+1]:
                self.set[j],self.set[j+1]=
                    self.set[j+1],self.set[j]
    return self.set[n-1]

def minimum(self):
    n=len(self.set)
    for i in range(n):
        for j in range(n-i-1):
            if self.set[j]>self.set[j+1]:
                self.set[j],self.set[j+1]=
                    self.set[j+1],self.set[j]
```


Scanned with CamScanner

return self.set[0]

Scanned with CamScanner