# Data structures :-

Date
Page

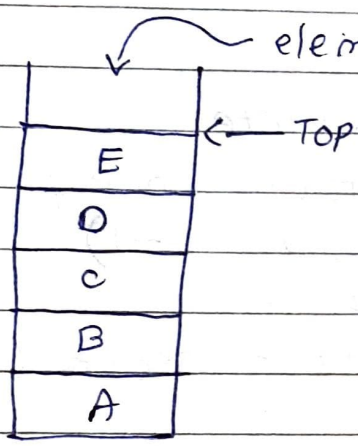
## stack

↳ stack is ordered collection of items

↳ new items inserted or deleted only at one end.

~~called top of stack~~

called top of stack.




→ stack is different from array

↳ it has the provision for the insertion and deletion of items.

↳ a stack is dynamic, constantly changing object.

→ The last element inserted into stack is first element deleted. → last in first out (LIFO).

→ operation on stack

→ push → item is added to stack

→ pop → item is removed from stack

→ given a stack s and item i,

-> push operation ⟶ push (s, i)

-> pop operation ⟶ i = pop (s).

→ empty(s) → determine whether or not a stack s is empty.

→ stack top (s) → return top element of stack without poping it.

| Push | pop. |
| --- | --- |
| → insert | → delete |
| → stack full (top == size - 1) | → stack empty (top == -1) |
| → top ++ | top -- |
| → stack (top) = ele | → ele = stack (top) |

→ Infix, postfix and ~~pre~~ prefix

→ Infix
↳ if the operator is situated in between the operands then the representation is called infix.

→ prefix
↳ if the operator is preceding the operands representation is called prefix.

→ postfix
↳ operator is following the operands....

A + B → infix
+ AB → prefix
AB + → postfix

Date
Page

## Infix to postfix :-

→ A + (B * C - (D / E ↑ F) * G) * H

| symbol | stack | Expression |
|---|---|---|
| A | | A |
| + | + | A |
| ( | +( | A |
| B | +( | AB |
| * | +(* | AB |
| C | +(* | ABC |
| - | +(- | ABC* |
| ( | +(-( | ABC* |
| D | +(-( | ABC*D |
| / | +(-(/ | ABC*D |
| E | +(-(/ | ABC*DE |
| ↑ | +(-(/↑ | ABC*DE |
| F | +(-(/↑ | ABC*DEF |
| ) | +(-(/↑) | ABC*DEF↑/ |
| * | +(-* | ABC*DEF↑/ |
| G | +(-* | ABC*DEF↑/G |
| ) | +(-*) | ABC*DEF↑/G*- |
| * | +* | ABC*DEF↑/G*- |
| H | +* | ABC*DEF↑/G*-H*+ |

Date
Page

## Evaluation of postfix Expression

→ ABC+* DE/-

| Symbol | stack | |
|---|---|---|
| A | A | |
| B | AB | |
| C | ABC | |
| + | A(B+C) | |
| * | A x y | y = (B+C) |
| D | ZD | Z = A x y |
| E | ZDE | |
| / | Z(D/E) | |
| - | Z - W | W = D/E |

Answer = Z - W

Date
Page

## Infix to prefix :

→ A + ( B * C - ( D / E ↑ F ) * G ) * H

| symbol | stack | Expression |
|---|---|---|
| H | | H |
| * | * | H |
| ( | *( | H |
| G | *( | HG |
| * | *(* | HG |
| ( | *(*( | HG |
| F | *(*( | HGF |
| ↑ | *(*(↑ | HGF |
| E | *(*(↑ | HGFE |
| / | *(*(/ | HGFE↑ |
| D | *(*(/ | HGFE↑D |
| ) | *(*(/) | HGFE↑D/ |
| - | *(- | HGFE↑D/* |
| C | *(- | HGFE↑D/*C |
| * | *(-* | HGFE↑D/*C |
| B | *(-* | HGFE↑D/*CB |
| ) | *(-*) | HGFE↑D/*CB*- |
| + | + | HGFE↑D/*CB*-* |
| A | + | HGFE↑D/*CB*-*A+ |

Answer = +A*-*BC*/D↑EFGH.

## ☆ Queue :-

↳ an ordered collection of items from which items may be deleted at one end called the front of the queue. and may be inserted at the other end called rear of the queue.

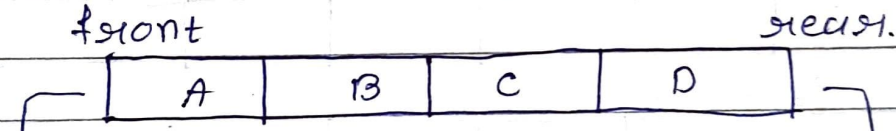
↳ first element inserted into a queue into a queue is first element to be removed. (FIFO) list.

→ operations on queue :-

→ insert → inserting element in queue.

↳ before insertion quefull condition must be checked

→ deletion → deleting element from the queue




element can deleted only at front of queue.

↓

no element in queue

front == -1

element can inserted only at rear of queue

↓

no element in queue

rear == -1

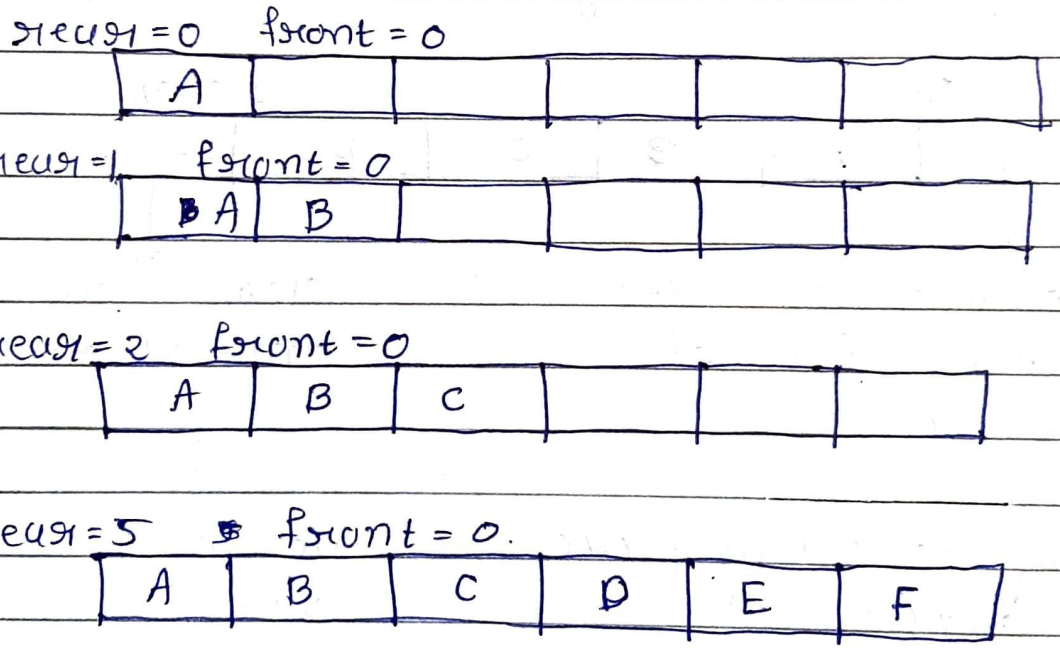
Date
Page

## Insertion algorithm

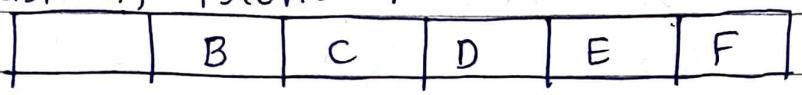
→ check stuck is full or not (rear = maxsize - 1)
→ if it is not full increament rear by only one position (rear ++)
→ insert element in queue → queue(rear) = element.

## Deletion algorithm :-

→ check queue is empty or not (front == -1)
→ if it is not empty then....
delete element from queue (element = queue(front))
→ increament front by one position
(front++)

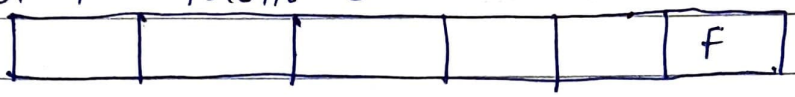


rear = 4, front = 1




rear = 4 front = 2


rear = 4 front = 5




## circuler queue :-

→

Insert :-

→ if front == (rear + 1) % maxsize ⟶ queue is full

→ else...

rear = (rear + 1) % maxsize

→ queue(rear) = element

→ if front = -1 then front = 0 and rear = 0.

→ delete :-

→ if front == -1 ⟶ queue is underflow.

→ else.

element = queue(front)

→ if

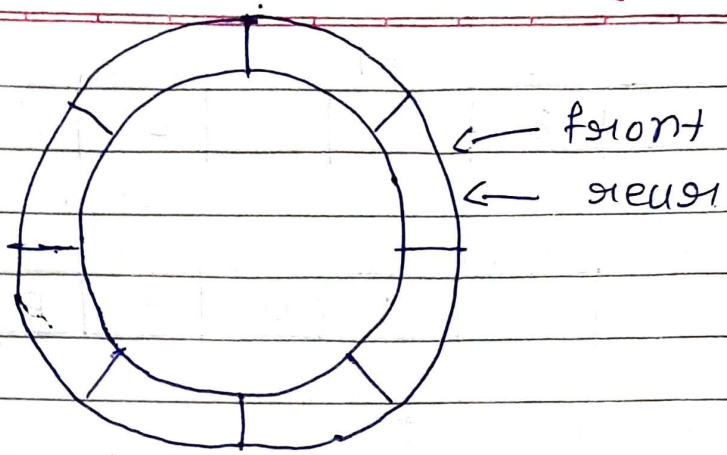
front = rear

front = -1

rear = -1

→ else.

front = (front + 1) % maxsize




## ★ Priority queue :-

→ Priority queue is a data structure in which the intrinsic ordering of the element determine type of priority queue.

↳ an ascending priority queue is a collection of items into which items can be inserted arbitarily and from which only the smallest item can be removed.

↳ a descending priority queue is collection of items into which can be inserted arbitarily and it allows deletion of only the largest item.

→ ~~for~~ priority queue has following rules :-

↳ an element of highest priority is processed before any element of lower priority

↳

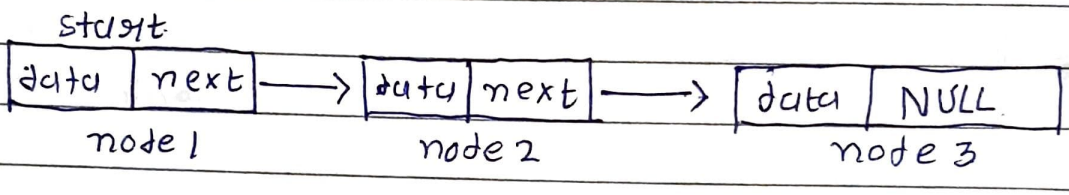
Date
Page

Two elements with the same priority are processed according to the order in which they were added to the queue.

~~linked liste~~

## linked lists :-

→ Suppose that the items of a stack or a queue where explicitly ordered i.e. each item contained within itself the address of the next item .... such un explicit ordering give rise on a data structure known as a linear linked list.




→ each item in the list is called a node and it contains two fields.
  - ↳ information field -> holds actual element on list
  - ↳ next address field -> it contaimes address of next node.

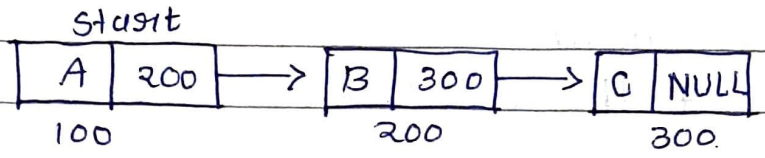
→ the entire linked list is accessed from an external pointer list that points to first node in the list.

→ the next address field of the last node in the list contains a special value called as NULL is represents end of the list.

→ Declaration →
```
struct node {
    int data;
    struct node *next;
};
```

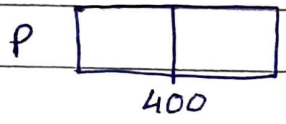
## ★ Insertion

↳ create a node
↳ assign the data
↳ adjust the pointers



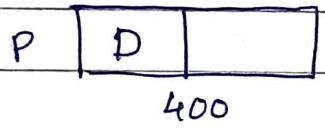

→ create node :-

```
p = malloc(sizeof(struct node))
```




→ assign data :-

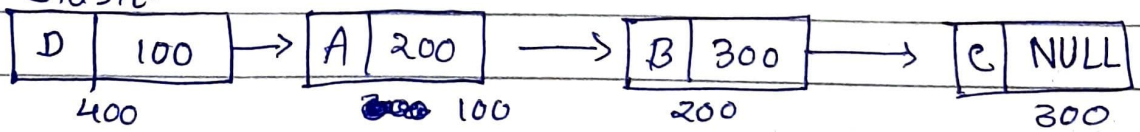
```
p -> data = D
```




→ adjust pointers :-

```
p -> next = start
start = p.
```

start.



Date
Page

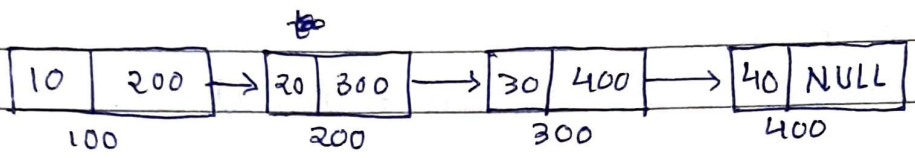
## Insert at specific position :-

↳ traverse a linked list upto the specific position.

↳ create a new node.

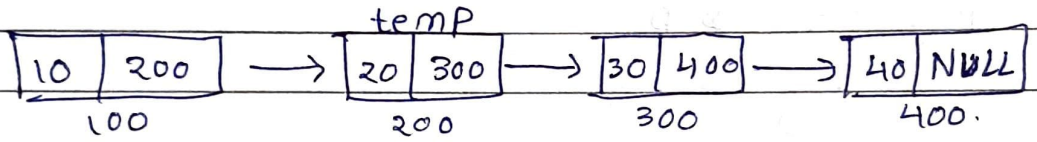
↳ insert data.

↳ adjust the pointers



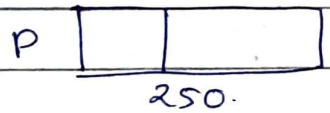

→ traverse linked list.

→ temp = temp → next.




→ create node.

p = malloc (sizeof (struct node))



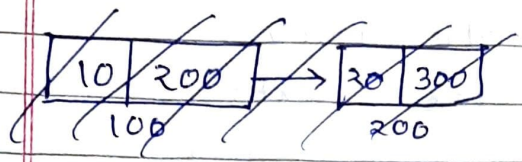

→ Insert data.

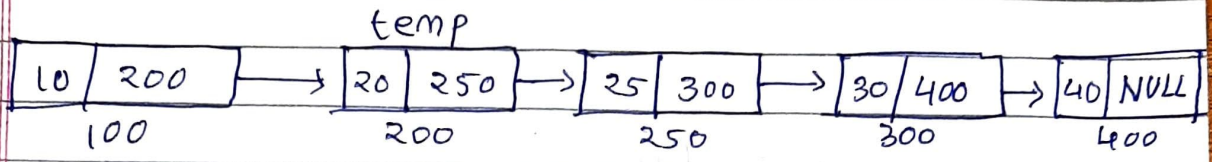
p → data = 25

→ adjust the pointers.

↳ p → next = temp → next

temp → next = p




## ✱ Doubly linked list :-

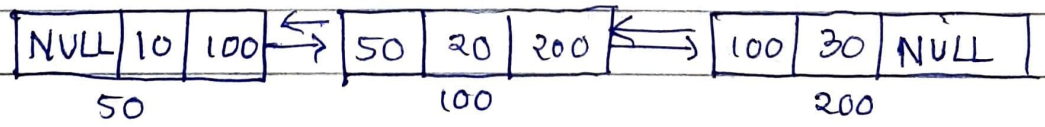
↳ in doubly linked list each nodes containes two pointers.

↳ node on doubly linked list containes three fields

↳ one of them stores actual data

↳ one pointer stores ~~pre~~ previous node address.

↳ another pointer stores next node address.




→ Previous part of node is NULL if it is starting, first node of list.

→ next part of node is NULL that shows that the last node of list.

→ Insertion

→ declaration

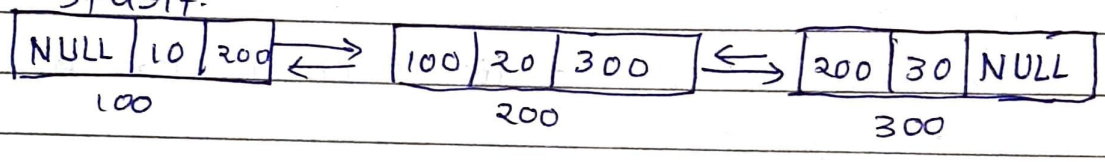
```
struct node {
    int data;
    struct node * prev;
    struct node *next;
    };
```

→ Insertion :- (at first)

↳ create a new node

↳ insert data

↳ adjust the pointer.

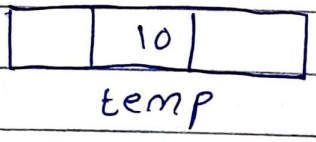



→ create a new node.

```
temp = malloc(sizeof(struct node))
```

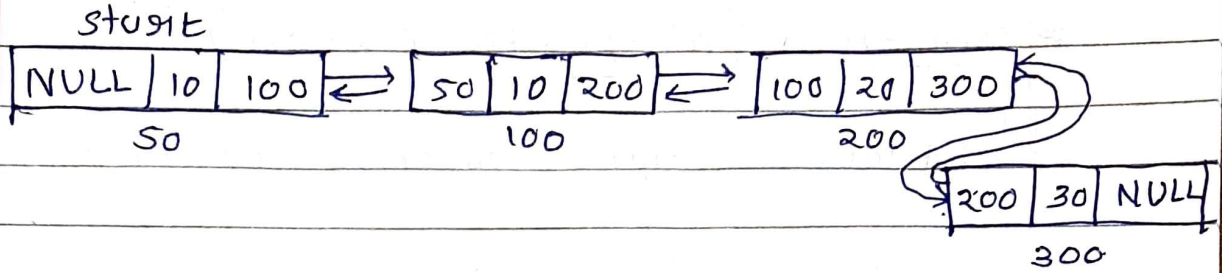
→ insert data

```
temp -> data = 10.
```




→ adjust the pointer :-

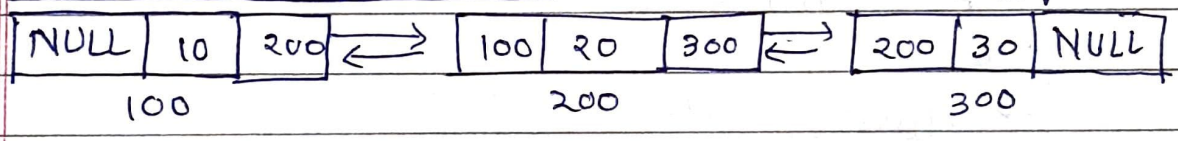
```
temp -> next = start
temp -> prev = NULL
start -> prev = temp
start = temp
```




## ★ Insertion (at last) :-

↳ traverse linked list upto NULL.
↳ create new node
↳ insert data
↳ adjust the pointer

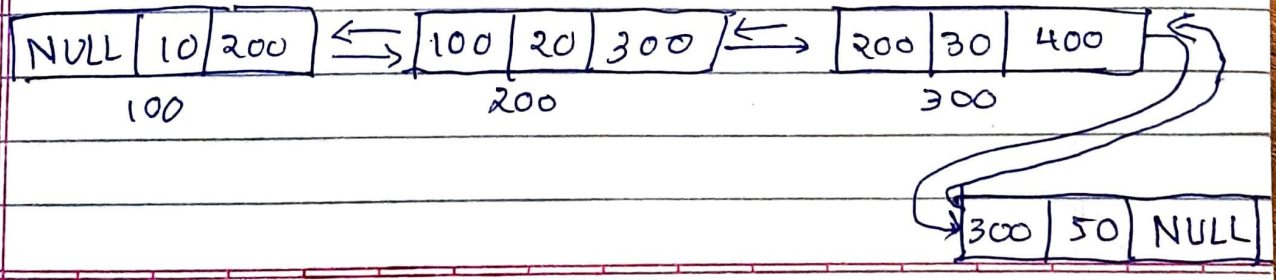
### ① temp → next != NULL :-




→ create node
→ p = malloc (sizeof (struct node))

→ insert data
p → data = 50.

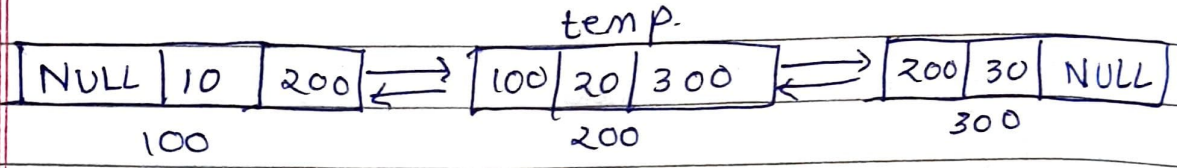
→ adjust pointers
temp → next = p
p → next = NULL
p → prev = temp.



## Insertion (at specific position)

↳ traverse upto specific position.

```
temp = temp -> next;
```




↳ create a node

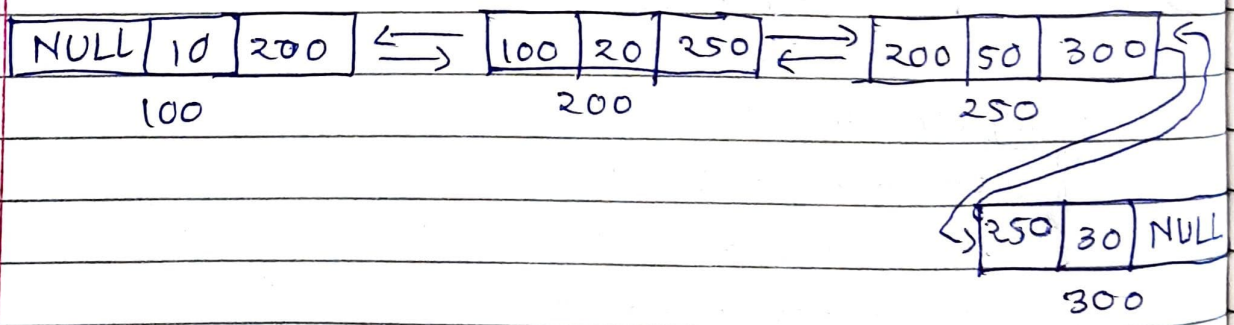
```
p = malloc (sizeof (struct node))
```

↳ insert data

```
p -> data = 50;
```

↳ adjust pointers :-

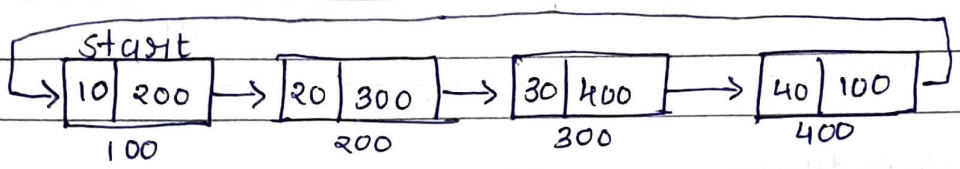
```
p -> prev = temp
p -> next = temp -> next;
temp -> next = p
p -> next -> prev = p;
```



## ✱ Circular linked lists :-

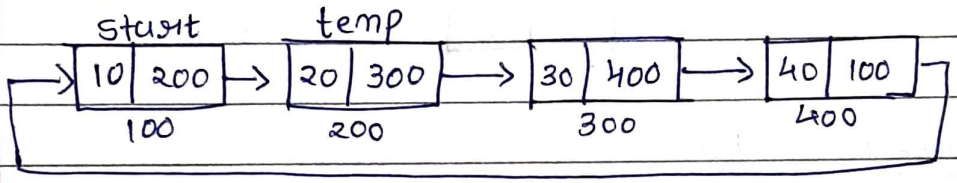
### ✱ Singly linked list

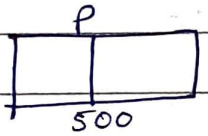
→




→ last node stores address of first node.

→ Insertion :-




→ create node

→ p = malloc (sizeof (struct node))

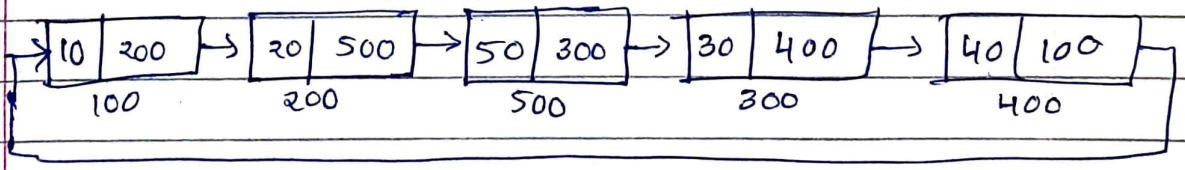
→ ● insert data

p → data = 50

→ Pointers adjust.

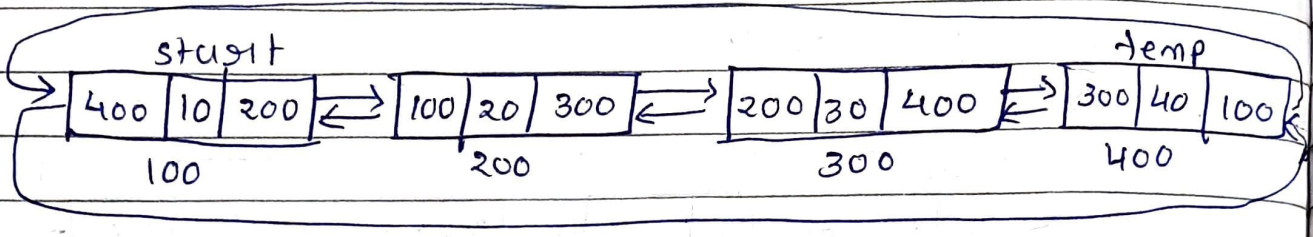
p → next = temp → next

temp → next = p



Date
Page

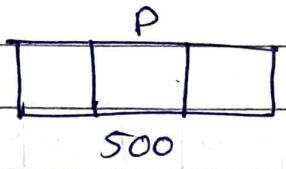
## doubly linked list :-




→ Insertion :-

→ create a node :-

p = malloc(sizeof(struct node))




→ insert data :-

p → data = 50.

→ adjust pointers :-

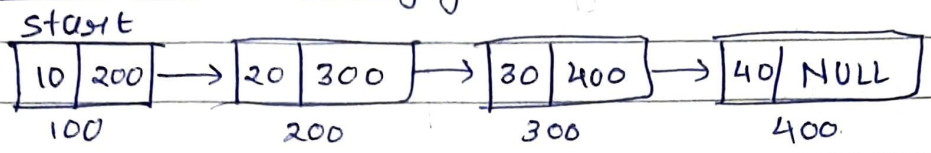
~~p → prev →~~ p → prev = temp

p → next = temp → next.

temp → next = p

start → prev = p

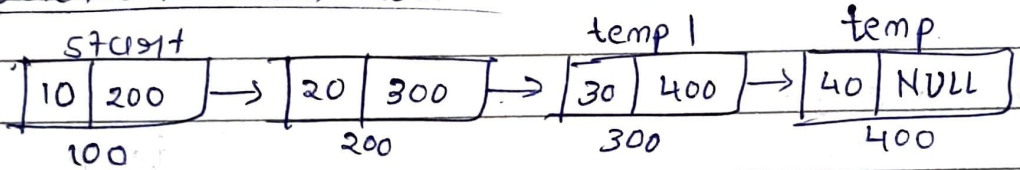
## Deletion from singly linked list :-




→ deletion from ~~front~~ start :-

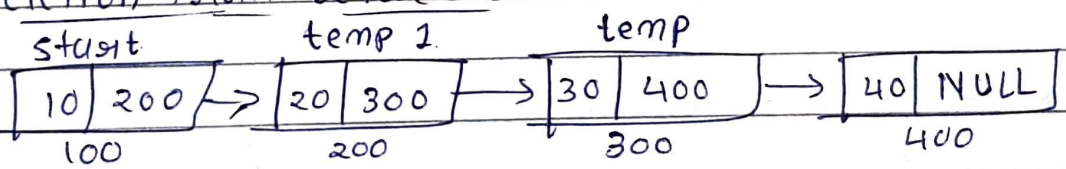
start1 = start → next
free (start)

→ deletion from last :-




→ temp 1 → next = NULL
free (temp)

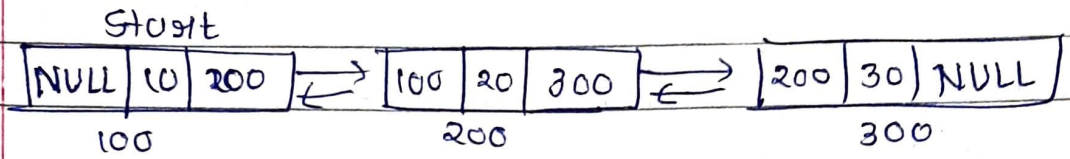
→ deletion from between :-




→ temp1 → next = temp → next
free (temp)

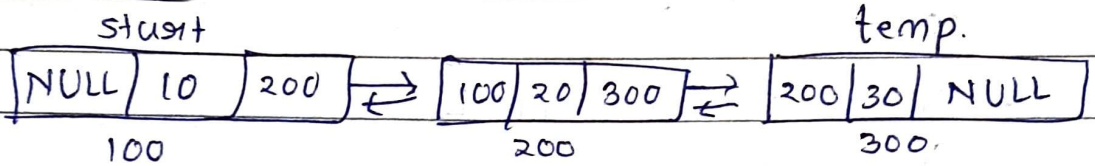
## ★ deletion in doubly linked list:-

### ✱ from start :-



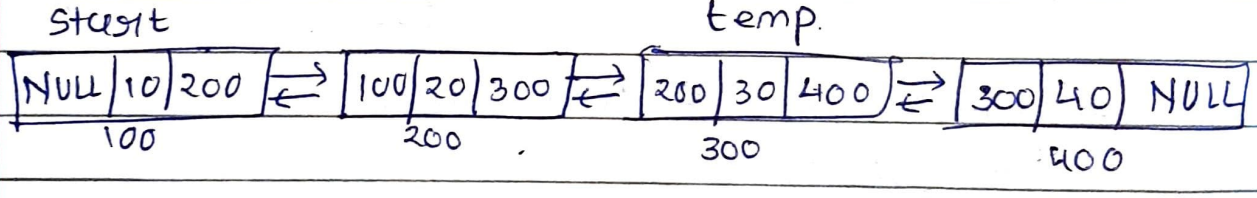

→ (start → next) → prev = NULL
start1 = start → next
free (start)

### ✱ from end :-




(temp → prev) → next = NULL
free (temp)

### ✱ from in between :-




(temp → prev) → next = temp → next
(temp → next) → prev = temp → prev.
free(temp).