LINKED LISTS

Advantages of using arrays in implementing data structures.

→ Faster data access : Specify array name & index.
Eg: 10th element = a[9] → same time taken.

→ Simple to understand & use.

Disadvantages of arrays:

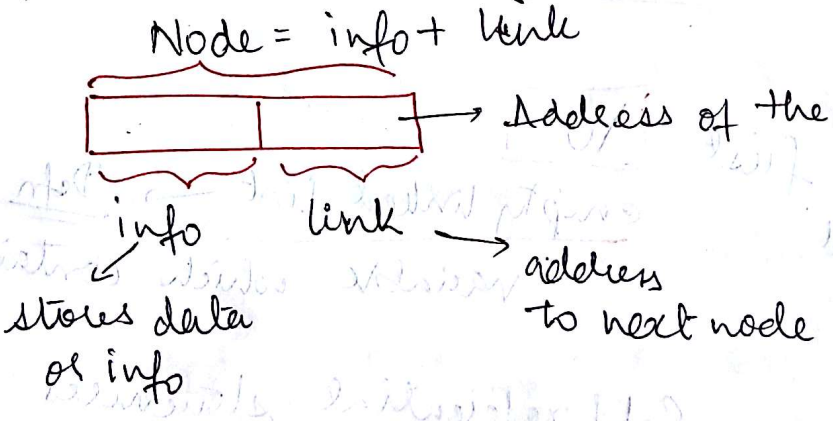
→ Fixed array size (static arrays)

→ Items stored contiguously.

→ Insertion & deletions involving arrays is tedious:
Insertion or deletion @ i^{th} position involves $(i+1)^{th}$ positions movement.

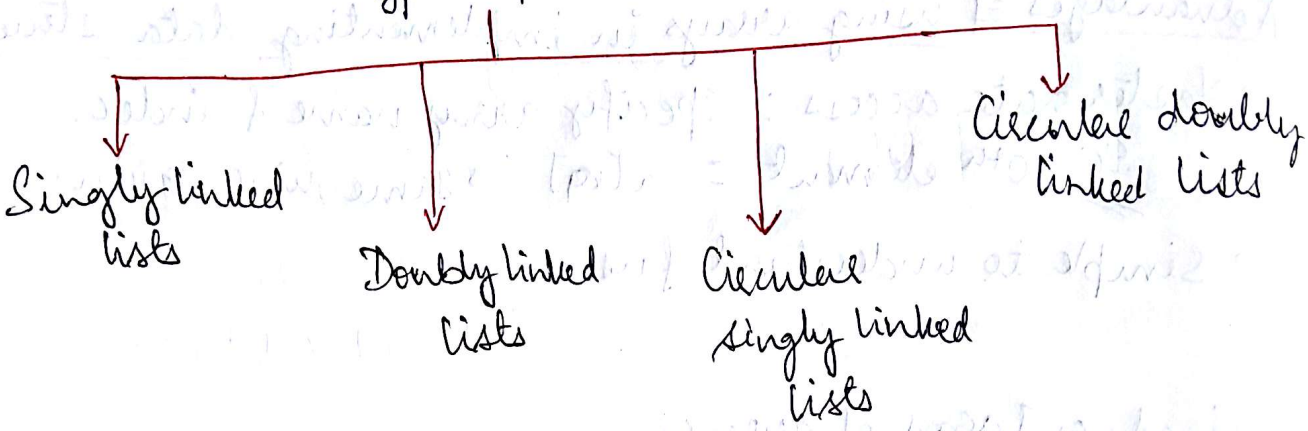
Linked list:

Defn: A linked list is a data structure which is a collection of zero / more nodes where each node has some information.



Note: Given address of one node, we can easily obtain addresses of subsequent nodes using "link" fields.

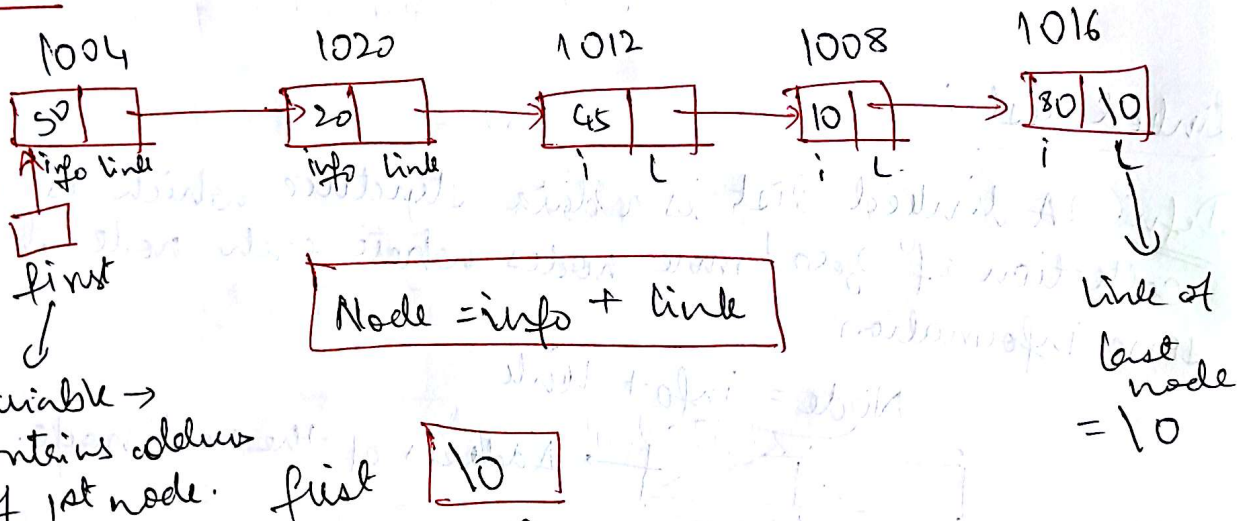
Types of linked lists



Singly linked lists & Chains:

Defn: A SLL is a collection of 0 or more nodes where each node has two or more fields & only one link field which contains address of the next node.

CHAIN: A SLL with 0 or more nodes.



Representation: Self referential structures

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

integer containing info

→ pts to the next node in the list

Structure defn:

```
struct node
{
    int info;
    struct node * link;
};
typedef struct node *NODE;
```

"first" declaration:

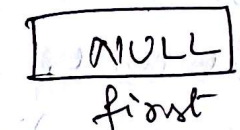
```
NODE first;
```

OR

```
struct node* first;
```

Create an empty list first:

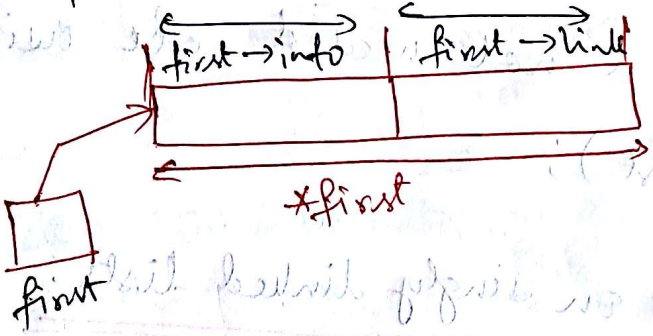
```
first = NULL;
```



Step 1: Create a node:

- using macro MALLOC()

```
MALLOC (first, 1, struct node);
```



Note:

→ first → access address of the node

→ *first → access entire contents of node

→ (*first).info or first->info → access data stored in info field

→ (*first).link or first->link → access link field

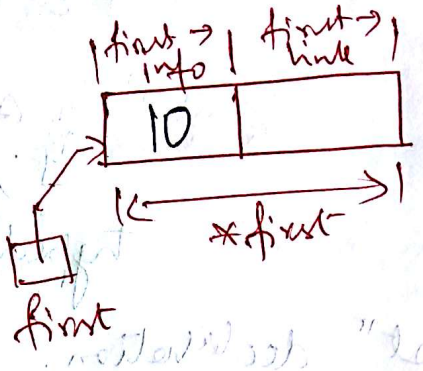
Step 2: Store the data:

→ Store 10 in info field.

first → info = 10;

or

(*first).info = 10;



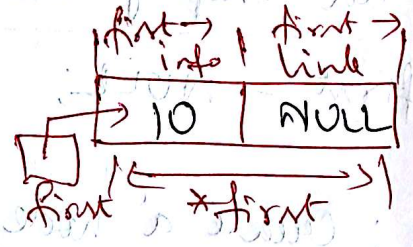
Step 3: Store NULL character:

If we do not want link field to contain address of any other node, store NULL.

first → link = NULL;

or

(*first).link = NULL;



Step 4: Delete a node:

Delete a not-needed node using free()

free(first);

Operations on singly linked lists:

Eg:

```
struct node* head = NULL;
struct node* second = NULL;
struct node* third = NULL;
head = malloc(sizeof(struct node));
second = malloc(sizeof(struct node));
third = malloc(sizeof(struct node));
```

Pointer assignment

```
head → data = 1;
head → next = second;
second → data = 2;
second → link = third;
third → data = 3;
third → link = NULL;
```

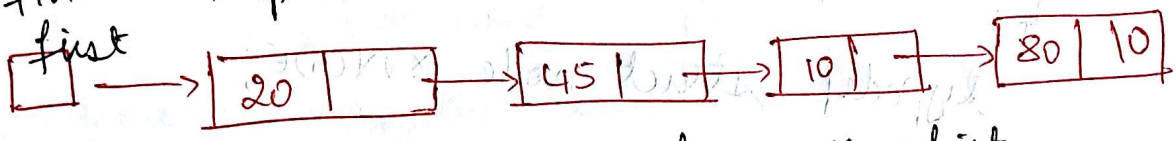
Operations:

1. Insert a node (@ the front end) into the list
2. Delete a node from the list
3. Search in a list
4. Display the contents of a list.

1. Insert a node:

- Insert a node @ the front end of the list.
- Consider a list of 4 nodes.

"first" → ptr to address the first node of the list



- Insert item 50 @ the front end of the list.

Step 1: Allocate memory for new node using malloc()

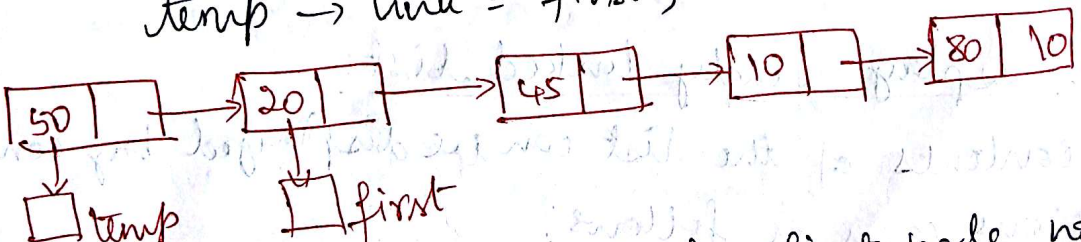
temp = malloc (sizeof (struct node))

Step 2: Copy item 50 into info field.

temp → info = 50; or item;

Step 3: Copy address of the first node of the list stored in pointer variable "first" into link field of temp.

temp → link = first;



Step 4: temp is inserted if it is the first node now.

return temp;

```
NODE insert_front (int item, NODE first)
{
    NODE temp;
    MALLOC (temp, 1, sizeof (struct node));
        temp → info = item;
        temp → link = first;
        return temp;
}

struct node
  { int info;
    struct node *link;
  };
  typedef struct node *NODE;

NODE first;
```

```
first = insert_front (item, first);
```

- If "first" is NULL & the above statement is executed, a linked list consisting only one node is created.
- If it is executed for the second time, a new node is inserted @ the front end & no. of nodes in the list = 2.

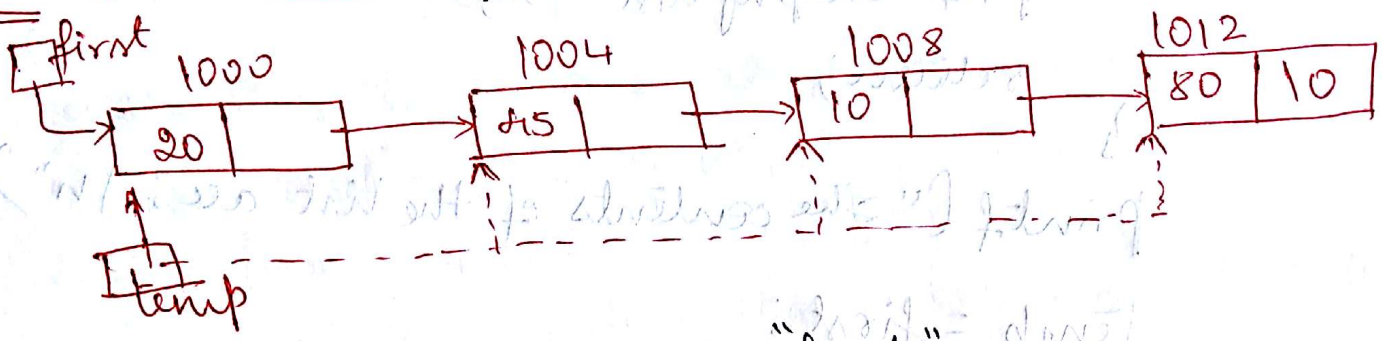
II Display singly linked list:

Contents of the list can be displayed by considering various cases as follows:

Case 1: List is empty: if (list empty) → no display.

```
if (first == NULL)
{
    printf ("List is empty");
    return;
}
```

Case 2: List not empty:



→ Every list has first node = "first"

→ To display, use another variable = "temp"

→ Initialise temp = first to start with.

```
temp = first;
```

→ Update temp to point to next node.

```
temp = temp -> link;
```

Hence, keep updating temp to diff. nodes.

→ Display "item" in every node temp is pointing to:

```
printf ("%d \n", temp -> info);
```

∴

```
temp = first;
while (temp != NULL)
{
    printf ("%d \n", temp -> info);
    temp = temp -> link;
}
```

C Program:

```
void display (NODE first)
{
    NODE temp;
    if (first == NULL)
    {
        printf("Empty list");
        return;
    }
    printf("The contents of the list are: \n");
    temp = first;
    while (temp != NULL)
    {
        printf("%d", temp -> info);
        temp = temp -> link;
    }
    printf("\n");
}
```

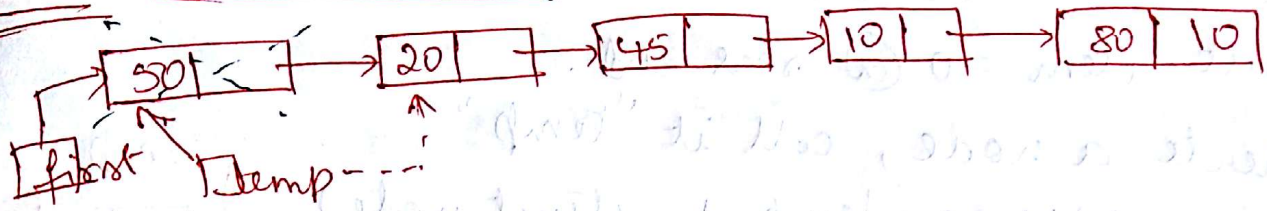
Other operations on linked lists:

① Delete node from front end:

Case 1: Empty list:

```
if (first == NULL)
    {
        printf("Empty list \n");
        return;
    }
```

Case 2: List is existing:



→ Create temp.

Initialise temp to first

temp = first

→ Update temp to contain address of second.

temp = temp → link

Now first = first node;

temp = second node;

To delete first node,

```
free (first);
```

temp, becomes first node, now, return temp.

```
return temp;
```

Code

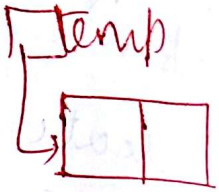
```
NODE delete_front (NODE first)
{
  NODE temp;
  if (first == NULL)
  { printf("List empty, cannot delete \n");
    return first;
  }
  temp = first;
  temp = temp → link;
  printf("item deleted = %d \n", first → info);
  free (first);
  return temp;
}
```

② Insert node @ rear end:

Insert item 50 @ rear end.

→ Create a node, call it "temp"

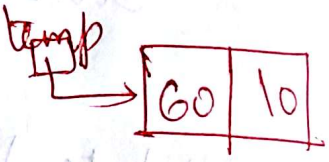
MALLOC(temp, 1, sizeof node);



→ Populate elements:

temp → info = item;

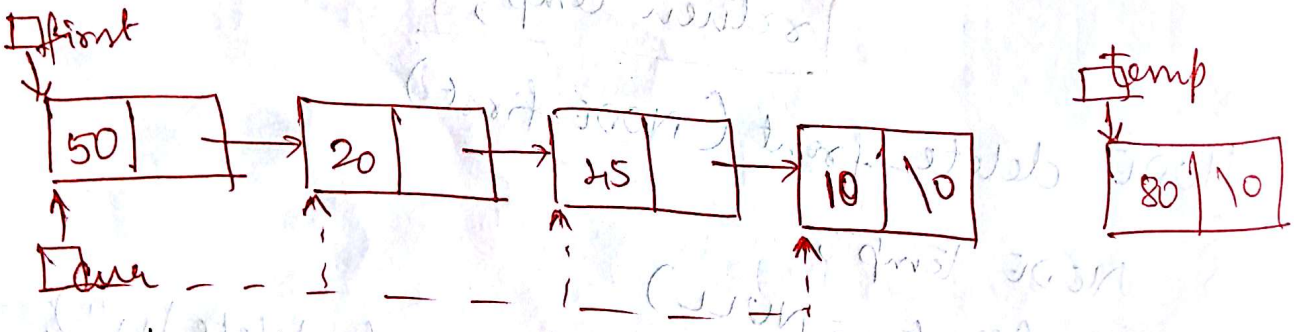
temp → link = NULL;



→ If list is empty, return the above node as the first node.

```
if (first == NULL)
    return temp;
```

→ If list not empty, insert the above node @ the end of the list.



— Create a pointer variable, "cur"

— Make "cur" point to "first":

cur = first;

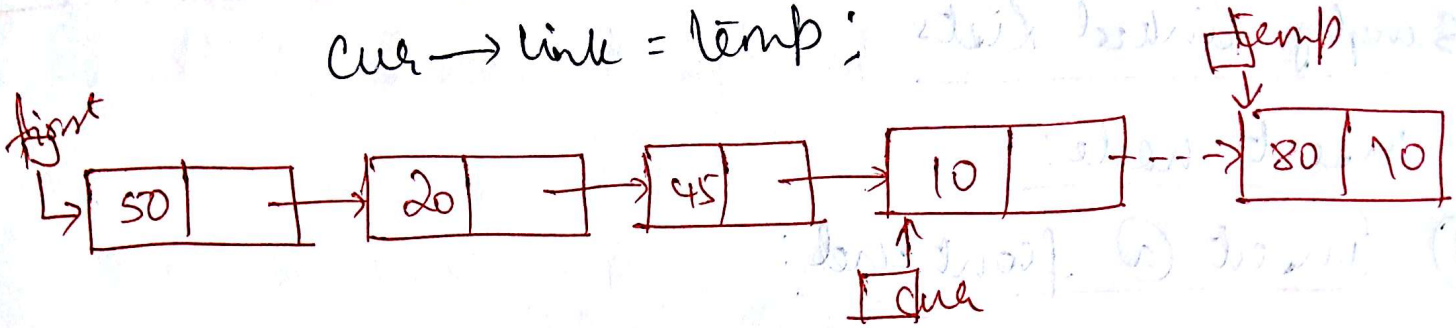
— Obtain address of the last node.

cur = cur → link;

```
while (cur → link != NULL)
    cur = cur → link;
```

→ Once "cur" points to the last node, insert temp address in this last node.

cur → link = temp;



→ After updating, return the address of first node.

return first;

CODE

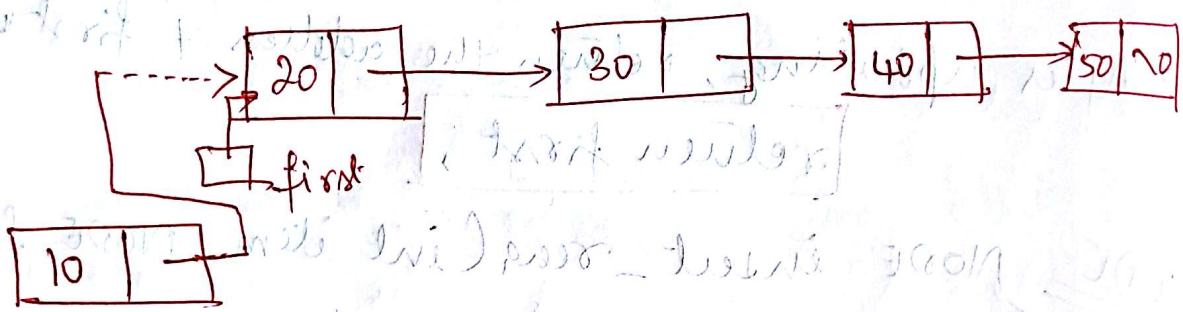
```
NODE insert_rear(int item, NODE first)
{
    NODE temp;
    NODE cur;
    MALLOC(temp, 1, sizeof(struct node));
    temp->info = item;
    temp->link = NULL;
    if (first == NULL)
        return temp;
    cur = first;
    while (cur->link != NULL)
    {
        cur = cur->link;
    }
    cur->link = temp;
    return first;
}
```

③ Delete a node from rear end:

Singly Linked Lists

Insert node:

① Insert @ front end:

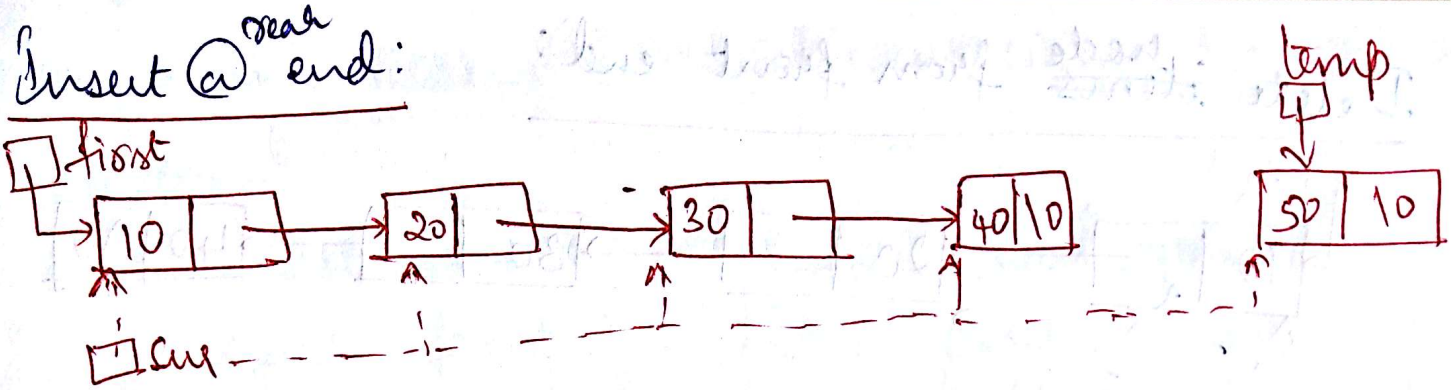


temp, return temp;

```
struct node
{ int info;
  struct node *link;
};
typedef struct node *NODE;

NODE first;
        first = insert_front(item, first);
NODE insert_front(int item, NODE first)
{ NODE temp;
temp= malloc(sizeof(struct node));
        temp → info = item;
        temp → link = first;
        return temp;
}
```

② Insert @ rear end:



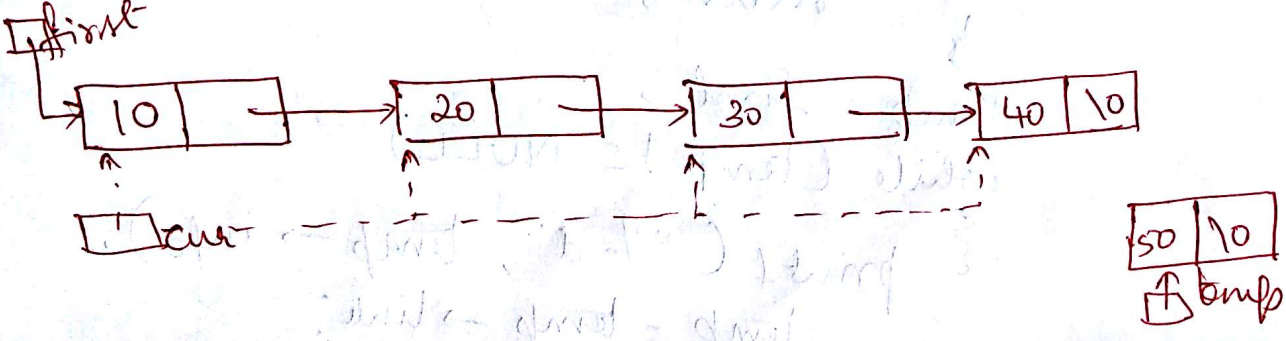

```
NODE temp;
temp = malloc(sizeof(struct node));
    temp → info = item;
    temp → link = NULL;

NODE cur;
    cur = first;
while (cur → link != NULL)
        cur = cur → link;
cur → link = temp;
return first;
```

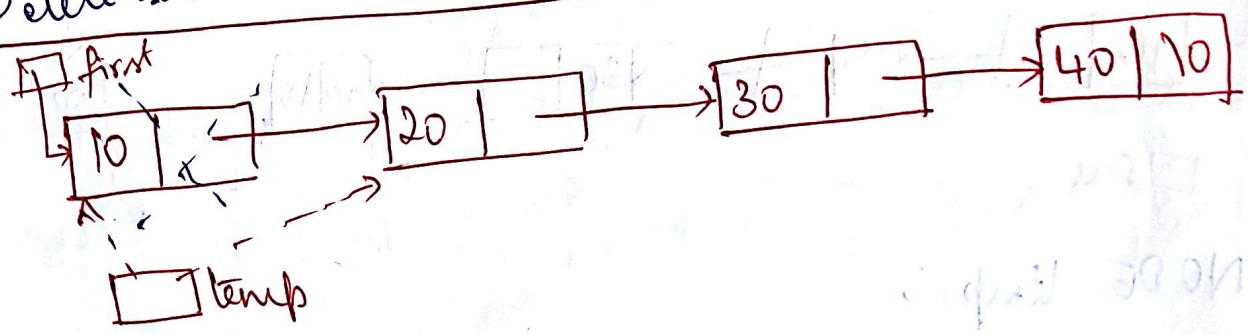
③ Implement Queues using linked list:

Operations: insert_rear()
delete_front()
display()

Insert item @ rear end:

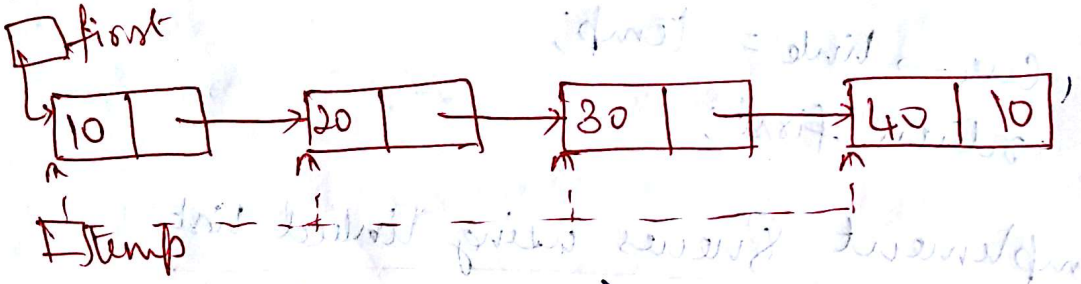


Delete ~~item~~ node from front end:




```
NODE temp;
    temp = first;
    temp = temp → link;
free(first);
return temp;
```

Display contents of list:




```
Void display (NODE first)
{ NODE temp;
    if (first == NULL)
    { printf ("List empty");
        return;
    }
    temp = first;
    while (temp != NULL)
    { printf ("%d", temp → info);
        temp = temp → link;
    }
}
```

(4) Implement Stacks using linked lists:

Operations:

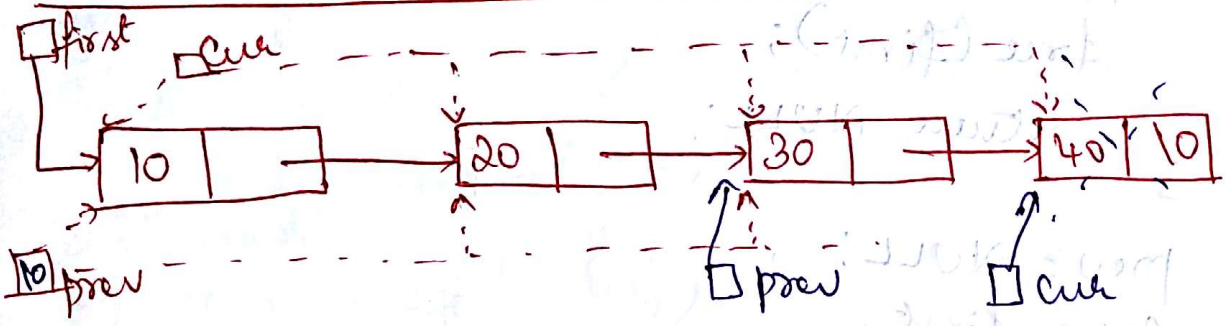
insert_front()
delete_front()
display()

OR

insert_rear()
delete_rear()
display()

insert_front, insert_rear, delete_front → done
display

Delete a node from rear end:



Initialise,

```
prev = NULL;
cur = first;
```

Update both cur & prev such that "cur" contains the last node & "prev" contains the last but one node.

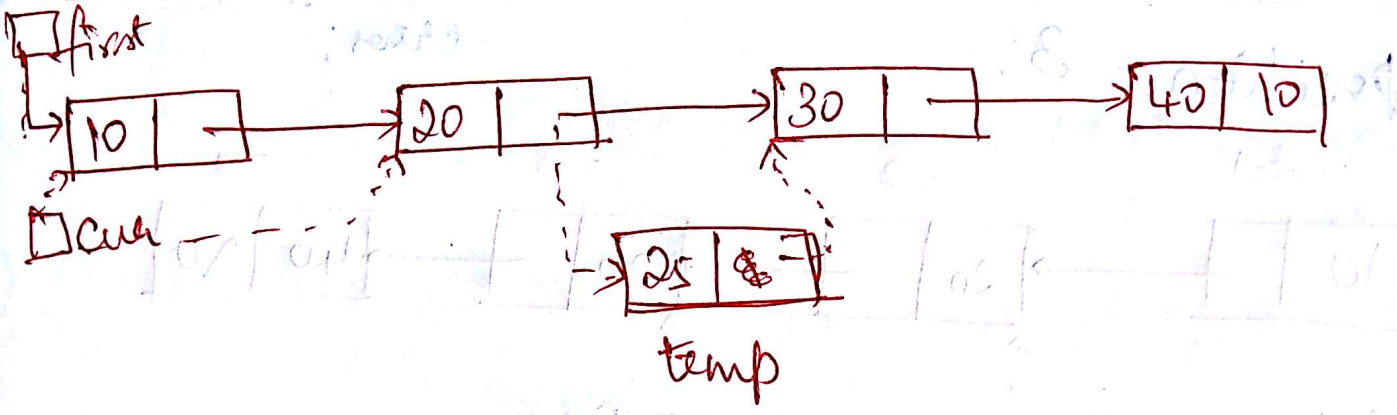
```
while (cur→link != NULL)
{ prev = cur;
  cur = cur→link;
}
```

Now, cur = last node & prev = last but one node.

```
printf("item deleted is %d \n", cur→info);
free(cur);
prev→link = NULL;
```

```
NODE delete_rear (NODE first)
{
    NODE cur, prev;
    if (first == NULL)
    {
        pf("list empty");
        return first;
    }
    if (first -> link = NULL)
    {
        pf("item deleted is %d", first -> info);
        free (first);
        return NULL;
    }
    prev = NULL;
    cur = first;
    while (cur -> link != NULL)
    {
        prev = cur;
        cur = cur -> link;
    }
    pf("item deleted is: %d", cur -> info);
    free (cur);
    prev -> link = NULL;
    return first;
}
```

(5) Insert by value/number:

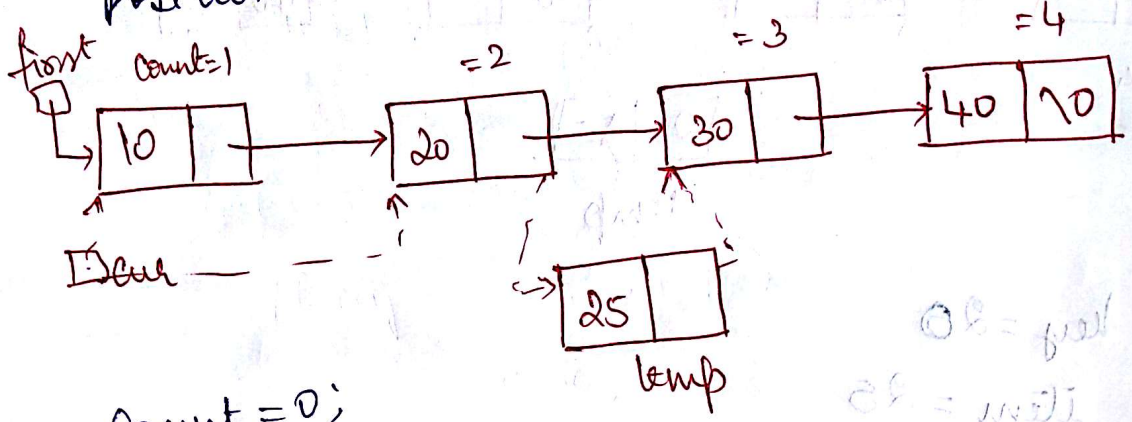


Here key = 20
item = 25

```
NODE temp;
temp = malloc (sizeof (struct node));
temp → info = item;
cur = head;
do
{if (cur → info == key)
  {  temp → link = cur → link;
     cur → link = temp;
     return;
  }
else
   cur = cur → link;
}
```

⑥ Insert by position: if(position > no of nodes) error;

position = 3;




```
count = 0;
cur = first;
while (cur != NULL)
{
    count++;
    if (count < position)
        cur = cur -> link;
    else
        break;
}
temp -> link = cur -> link;
cur -> link = temp;
return;
```

⑦ Search for a key in the linked list:

```
NODE search_key (int key, NODE first)
{
    NODE cur;
    cur = first;
    while (cur != NULL)
    {
        if (key == cur -> info)
            return cur;
        cur = cur -> link;
    }
    return NULL;
}
```

(8) Delete from front end:

DONE

(9) Delete from rear end:

DONE

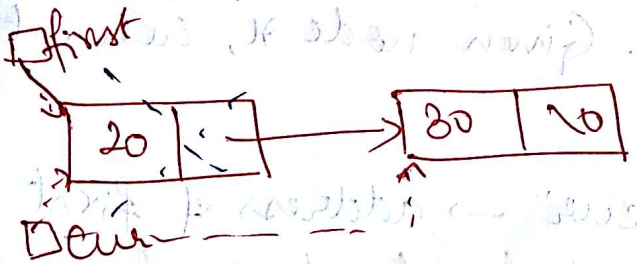
(10) Delete by key/value/number:

key = 20;

Case 1: List empty

```
if(first == NULL)
{
    pf("List empty");
    return NULL;
}
```

Case 2: Key present in first node:




```
cur = first
if (cur->key = cur->info)
        first = first->link;
free(cur);
return first;
```

Case3: If key is present in the list (not in first)

```
NODE prev, cur;
prev = NULL;
cur = first;
while (cur != NULL)
{   if(key == cur -> info)
        break;
    prev = cur;
    cur = cur -> link;
}
prev -> link = cur -> link;
free(cur);
return(first);
}
```

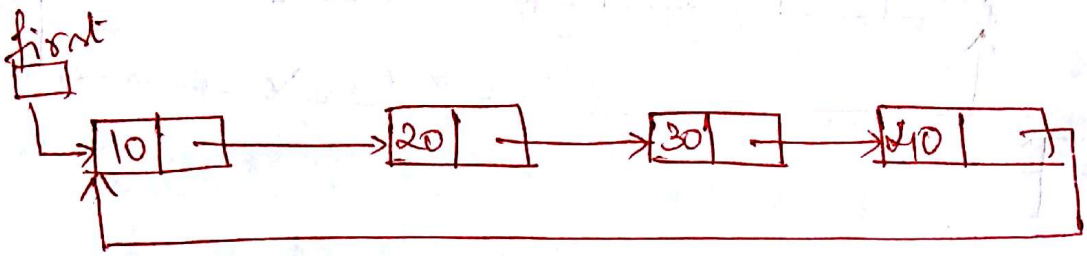
Circular Singly Linked List

Disadvantages of SLL:

(1) In SLL, only one link, hence traversing is done in only one direction. Given node x, cannot access the previous node.

(2) Deletion of node "cur" → address of first node is needed, hence search has to begin from first node of list.

Defn: A circular SLL is a variation of an ordinary linked list in which the "link" field of the last node contains the address of the first node.



Advantages of C-SLL:

(1) Every node is accessible from a given node by traversing successively using "link" field.

(2) To delete a node "cur", need not start from "first" node. The predecessor can be found out using "cur" itself.

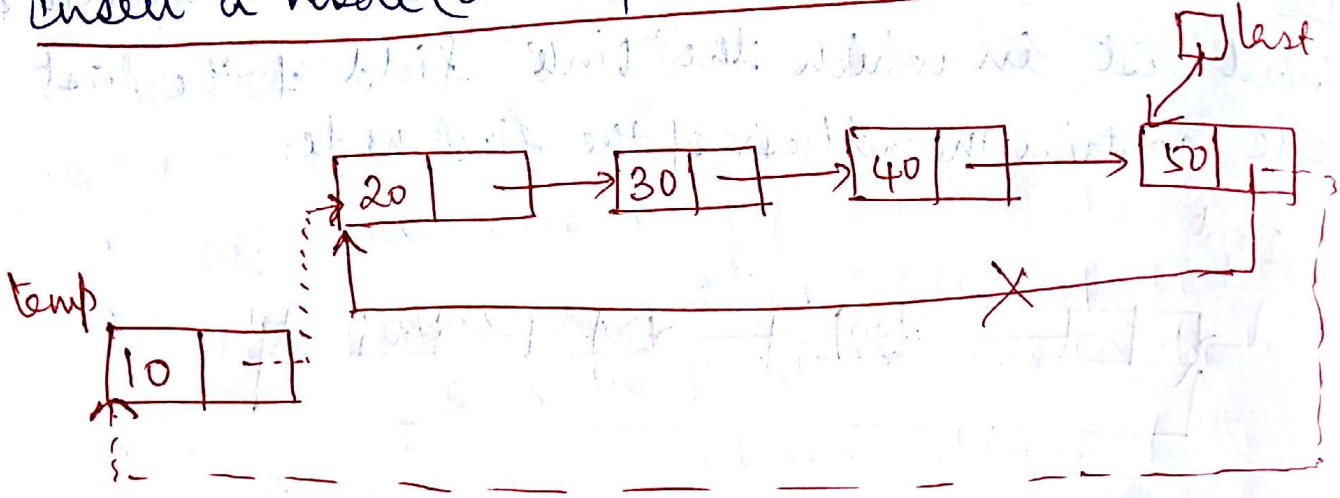
(3) Certain operations such as concatenation & splitting of lists is more efficient here.

2 conventions to identify first & last nodes:

Approach 1: Use ptr. variable "first" to designate the starting pt of the list.
Address of "last" can be obtained by traversing the entire list.

Approach 2: Use ptr. variable "last" to designate the last node. Node after "last" becomes the "first" node.

(1) Insert a node @ the front end:



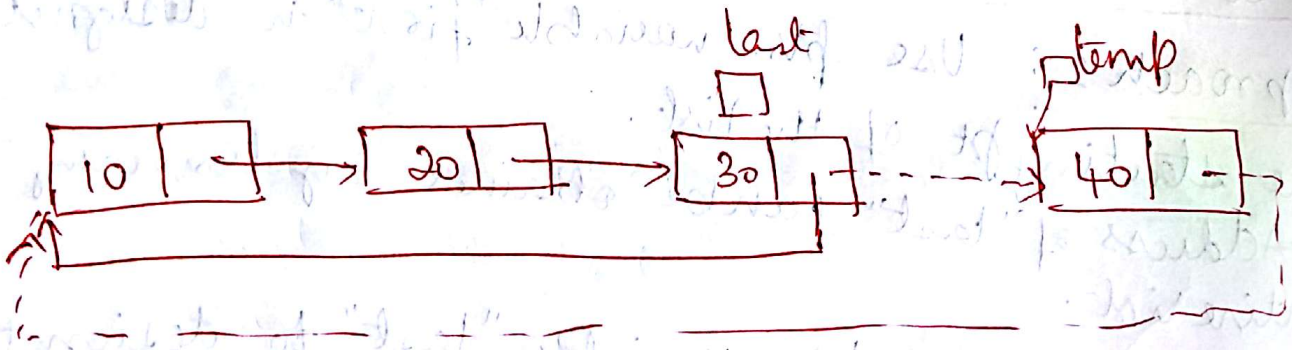
(1) Create temp:

```
temp = malloc(sizeof(struct node));
temp → info = item;
```

(2). Insert temp b/w last & first.

```
temp → link = last → link;
last → link = temp;
return last;
```

(2) Insert node @ rear end:

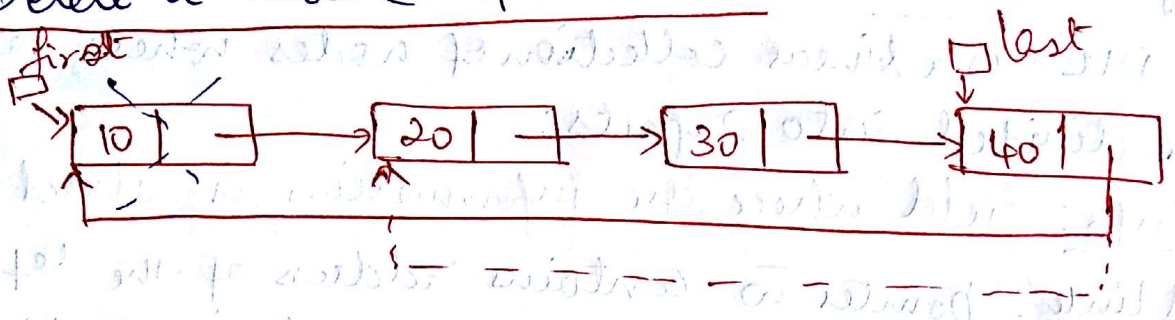


(1) create temp;

(2) temp → link = last → link;
last → link = temp;

return temp;

(3) Delete a node @ front end:



(1) Obtain first

```
first = last → link;
```

(2) Make last point to second (after first)

```
last → link = first → link;
```

(3) Delete first

```
free (first);
```

(4) Display the contents of a C-SLL:

```
Void display (NODE last)
{
  NODE temp;
  if (last == NULL)
  { printf ("Empty list");
    return;
  }
  printf ("contents: ");
  temp = last → link;
  while (temp != last)
  { printf ("%d", temp → info);
    temp = temp → link;
  }
  printf ("%d \n", temp → info);
}
```

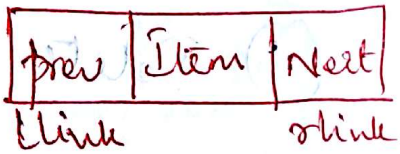
Doubly linked list:

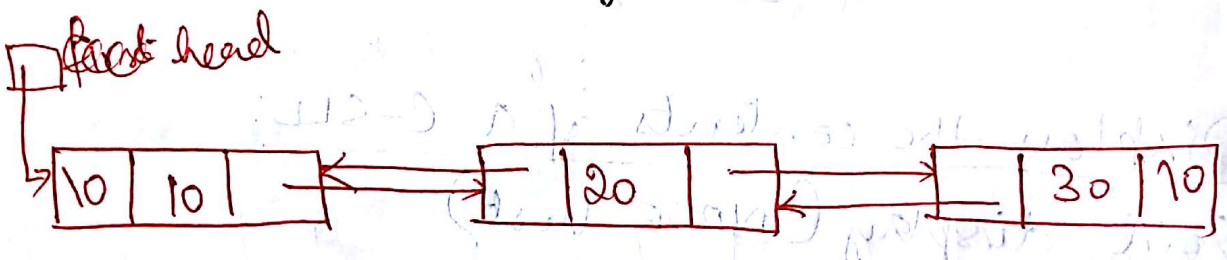
Defn: A DLL is a linear collection of nodes where each node is divided into 3 parts:

- → **info:** field where the information is stored
- → **llink/prev:** pointer to contains address of the left node of the previous node in the list.
- → **rlink/next:** pointer to contains address of the right node of the next node in the list.

→ Possible to traverse the list in forward & backward directions.

→ Also called a two-way list.



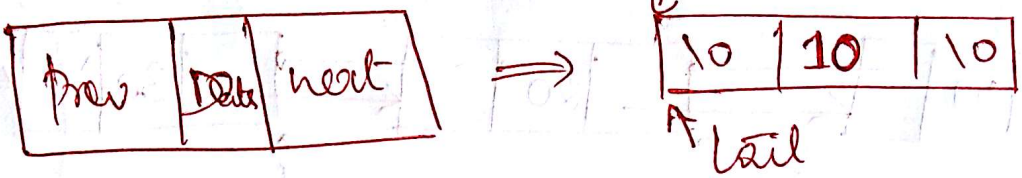


Structure Representation:

```
typedef struct dnode
{
    int data;
    struct node *prev;
    struct node *next;
}DNODE;
```

Operations on DLL:

① Creation of DLL:



Case 1:

Create node.

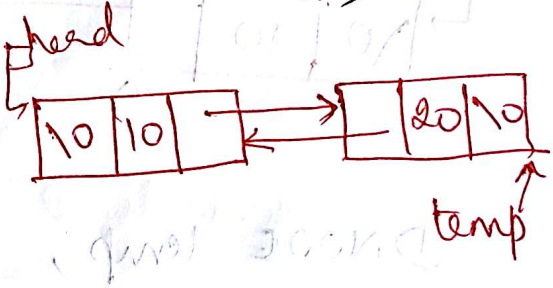
```
DNODE head;
    head = malloc (sizeof (struct dnode));
    head → prev = NULL;
    head → data = item;
    head → next = NULL;
```

```
DNODE tail;
tail = head;
```

Case 2: Create 2nd node.

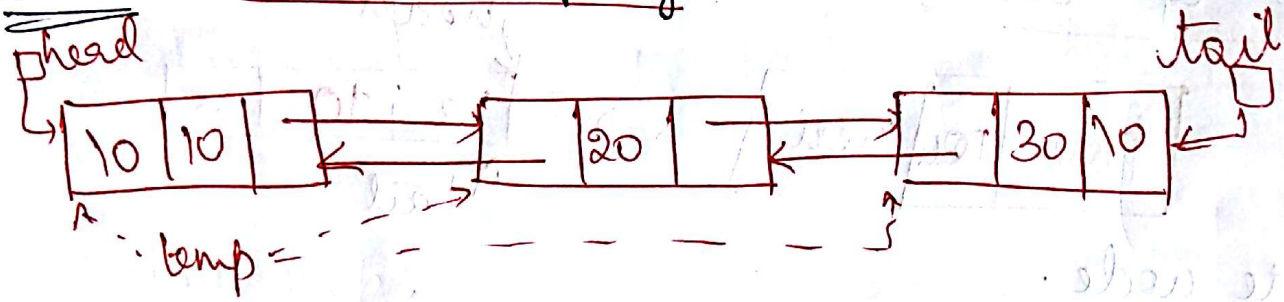
create 2nd node - temp.

```
DNODE temp;
    temp = malloc (sizeof (struct dnode));
    temp → data = item
    temp → next = NULL
    head → next = temp;
    temp → prev = head;
if (temp → next == NULL)
    tail = temp;
```



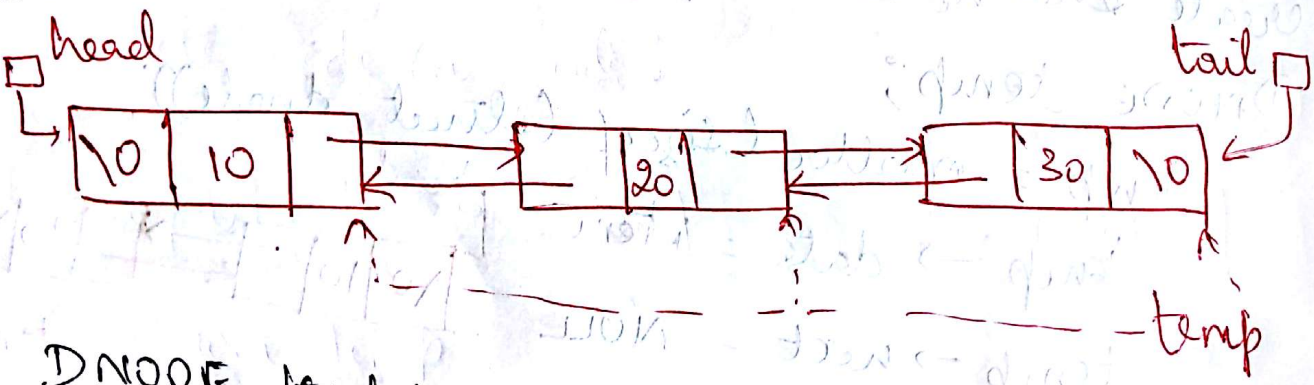
② Display of DLL:

Case 1: Forward display:



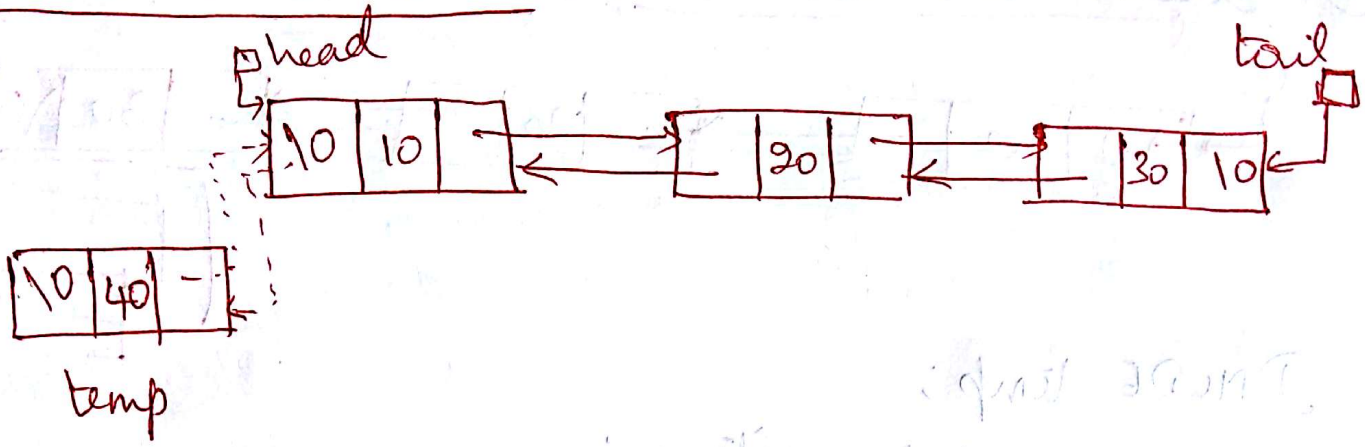

```
DNODE temp;
    temp = head;
while(temp != NULL)
{ printf("%d \n", temp -> data);
    temp = temp -> next;
}
```

Case 2: Reverse display:



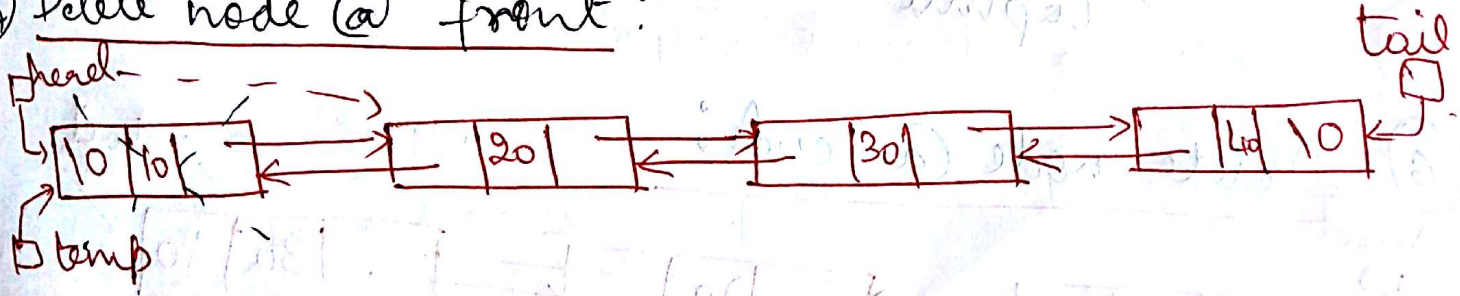

```
DNODE temp;
    temp = tail
while (temp != NULL)
{
    printf("%d \n", temp -> data);
    temp = temp -> prev;
}
```

③ Insert a node @ front:



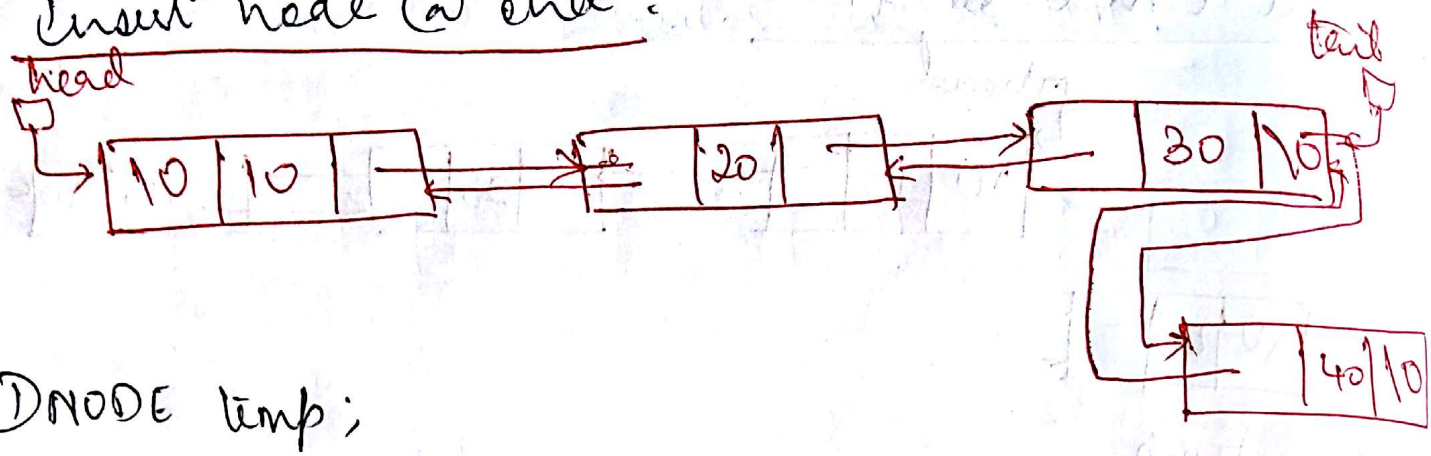

```
DNODE temp;
temp → prev = NULL;
temp → data = item;
temp → next = head;
head → prev = temp;
return temp; / head = temp;
```

④ Delete node @ front:




```
DNODE temp;
temp = head;
head = head → next;
free (temp);
head → prev = NULL;
```

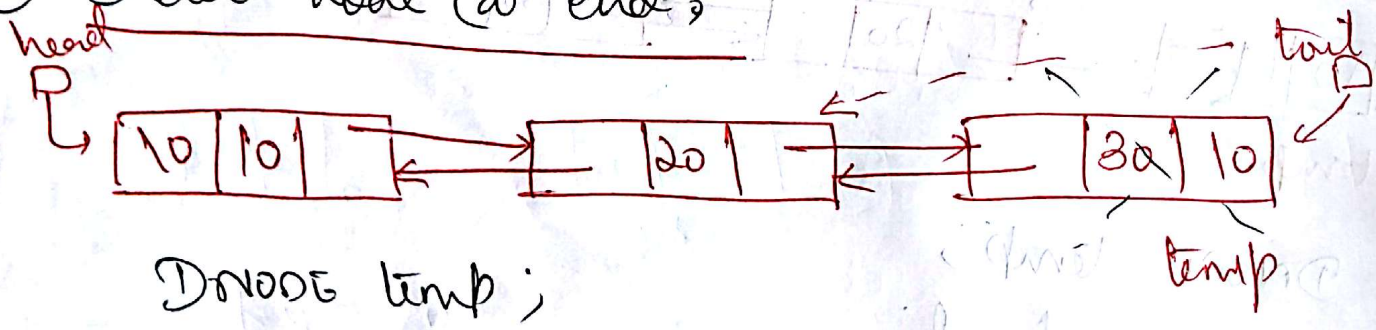
⑤ Insert node @ end :




```
DNODE temp;
    temp → data = item;
    temp → next = NULL;
    temp → prev = tail;
    tail → next = temp;
    return temp;
```

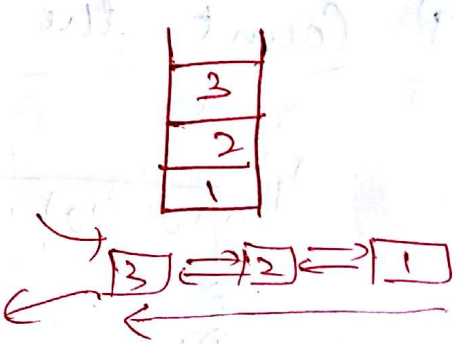
↳ Capture this temp as tail

⑥ Delete node @ end;




```
DNODE temp;
    temp = tail;
tail = tail → prev;
    free(temp);
tail → next = NULL;
```

(7) Implement Stacks using DLL:

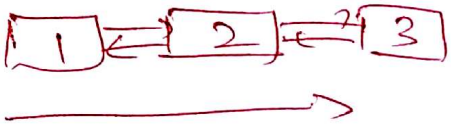


insert_front();

delete_front();

reverse_display();

OR

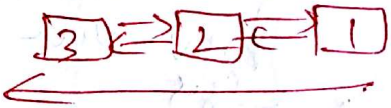


insert_rear();

delete_rear();

forward_display();

(8) Implement Queues using DLL:



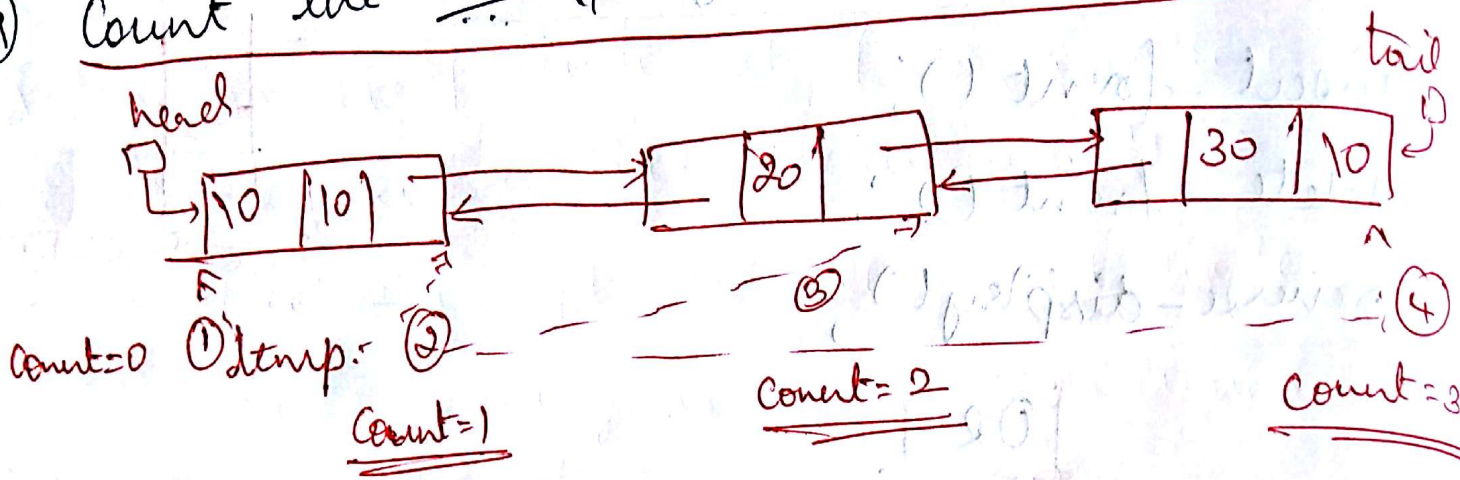
insert_rear();

delete_front();

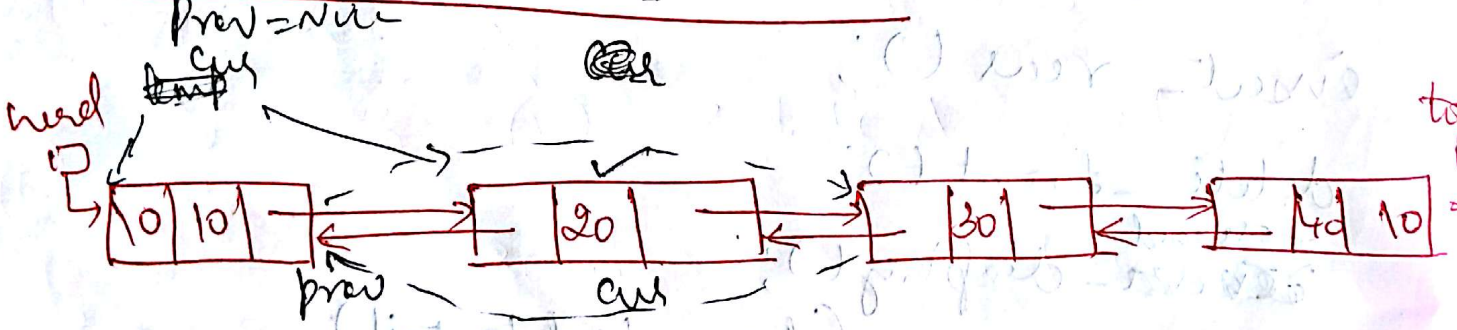
forward_display();

(from head to tail)

(9) Count the no of nodes in the list:



(10) Search an element & delete:

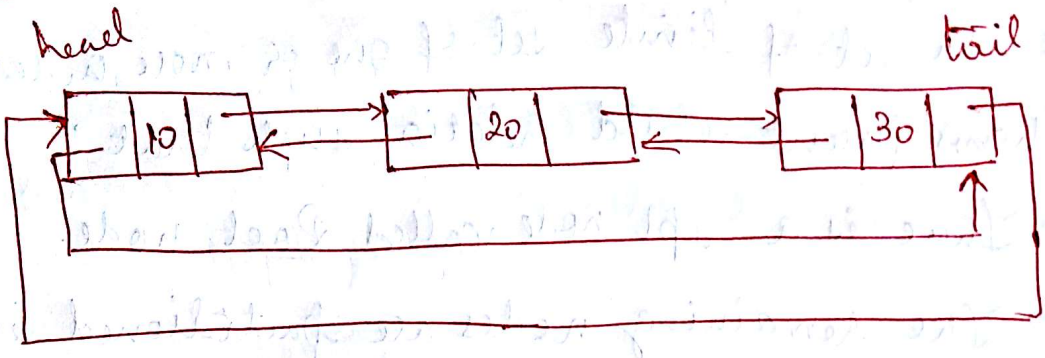


Key = 20

~~temp = head~~

prev = NULL
cur = head } Increment both

Circular Doubly Linked List:



Operations:

1. Insert node @ front / rear end
2. Delete node @ front / rear end
3. Display
   - Forward display (head to tail)
   - Reverse display (tail to head)

Applications of linked list:

1. Viewing images
2. Phones contacts list.
3. Stacks, Queues, graphs etc.
4. Linux process management
5. Cache in your browser that lets you hit the BACK button (linked list of URLs)