LINKED LIST

→ ~~Linear~~ Linked list are special list of some data elements linked to one another.

→ Disadvantage of arrays

1. We cannot increase the size of array when additional space is required.
2. It is necessary to declare in advance the amount of memory to be utilized

Advantage of linked list

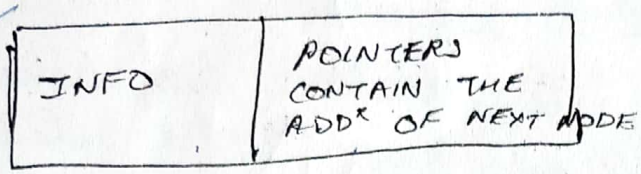
1. Linked list are dynamic data structure
2. Efficient memory utilisation.
3. Insertion & deletion are easier & efficient i.e flexibility in inserting in data item at a specified position & deletion of a data item from the given position.
4. Many complex application can be easily carried out with linked list.

Disadvantage of linked list

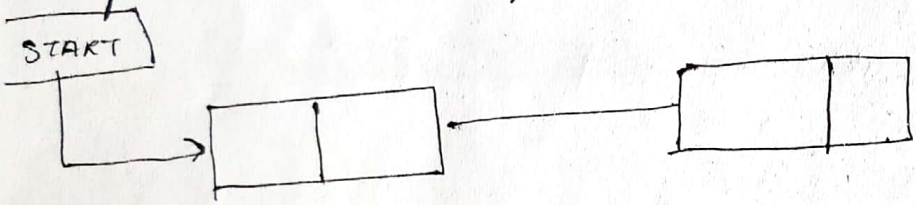
1. More Memory - i.e. If the no. of fields are more then more memory space is needed
2. Access to an arbitary data item is little bit difficult & time consuming.

Linked list →

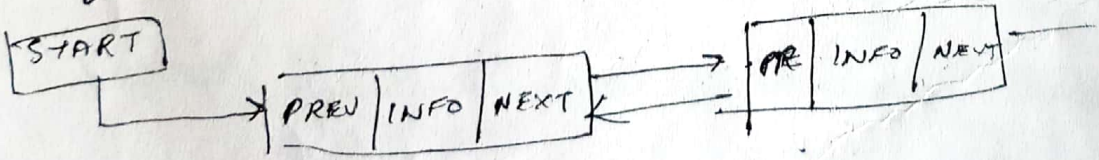
- A linked list or one way list is a linear collection of data elements, called nodes where the linear order is given by means of pointers.



→ In singly linked list nodes have one pointer (Next) pointing to the next node.



→ In doubly linked list have two pointers (prev & next). Prev points to the previous node & next points to the next node in the list



Operations on linked list

(1) Creation → It create linked list i.e. node is created.

(2) Insertion → It insert new node in the link list.

(3) Deletion → It delete node from the linked list.

(4) Traversing → process of going through all nodes of a linked list from one end to the other end.

(5) Concatenation
It is the process of combining two list.

(6) Display
This operation is used to print each & every node information.

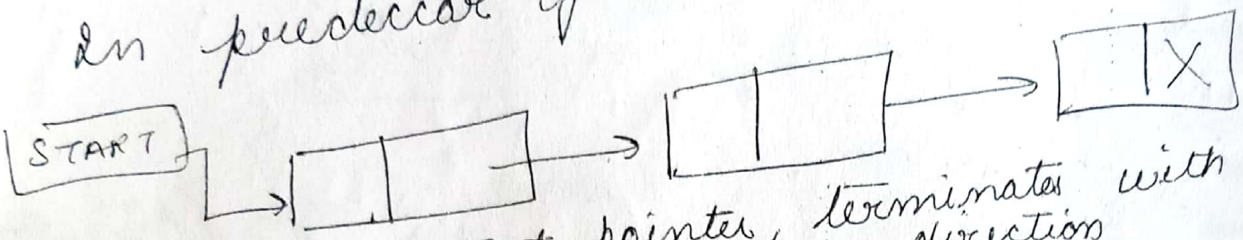
(7) Searching

Types of linked list

(1) Singly - linked list / linear linked list.

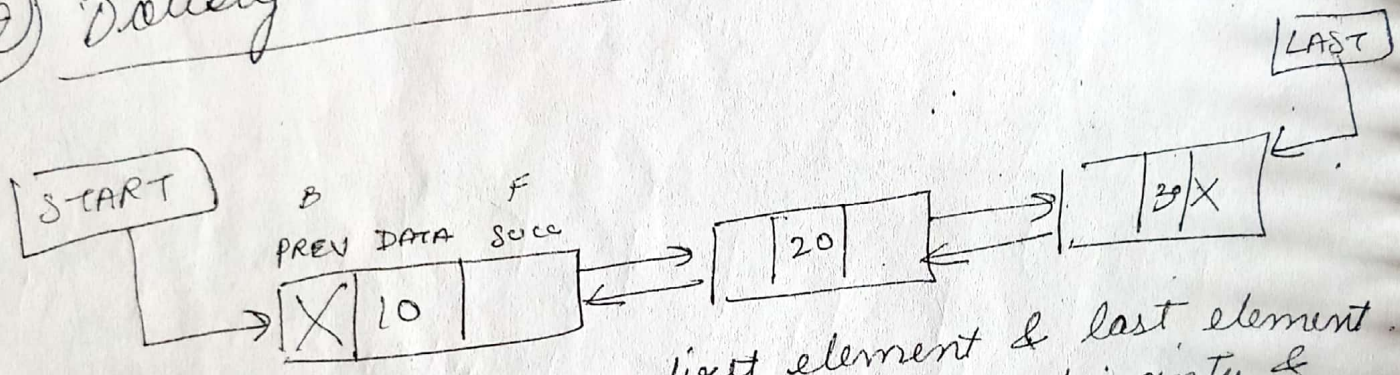
→ In this all nodes are linked together in some sequential manner.

→ Problem:
In predeccar of node cannot be accessed.



→ Begin with start pointer, terminates with a null pointer, only traversed in one direction

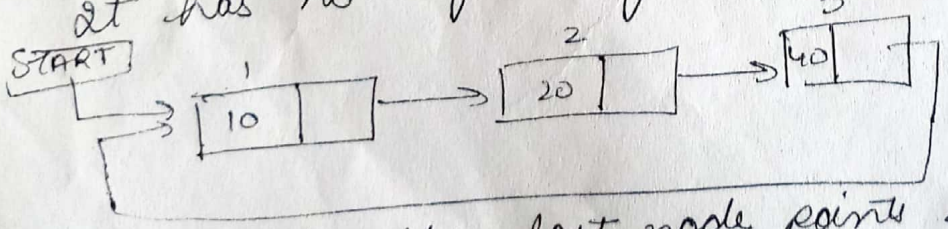
(2) Doubly - linked list



→ Two start pointers - first element & last element.
Each node has a forward & backward pointer & all traversal in both direction.

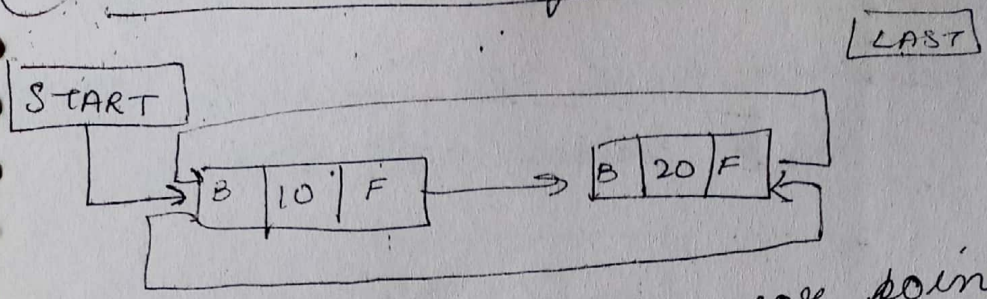
(3) CIRCULAR LINKED LIST

It has no beginning & no end.



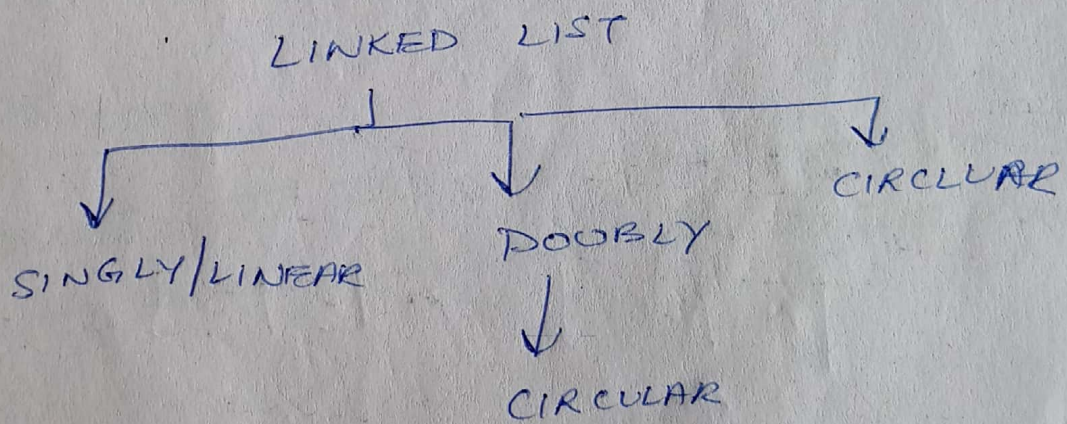
pointer in the last node points back to the first node.

(4) Circular doubly linked list



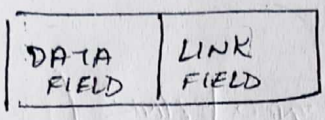
→ In this both the successor pointer & predecessor pointer in circular manner.

→ Forward pointer of the last node points to the first node & backward pointer of the first node points to the last node.



Def^n

Node: Linked list is a non-sequential collection of data item called nodes.



DATA FIELD → It contain an actual value to be stored & processed.

LINK FIELD → It contain address of next ~~node~~ data item in the linked list.

NULL POINTER → The link field of the last node contain NULL rather than valid address

EXTERNAL POINTER →

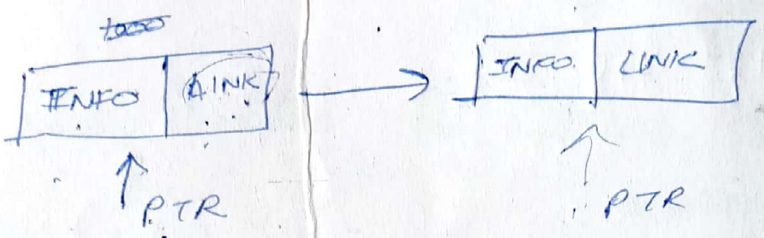
It is a pointer to the very first node in the linked list, it enables to access the entire linked list.

EMPTY LIST

If the node are not present in a linked list, then it is a empty linked list or also known as null list.

∴ Start = NULL.

→ Node (p) → a node pointed to by the pointer p
Data (p) → data of the node pointed to by p
Link (p) → address of the next node that follows the node pointed to by the pointer p.



PTR → LINK ✓

LINK[PTR] ✓

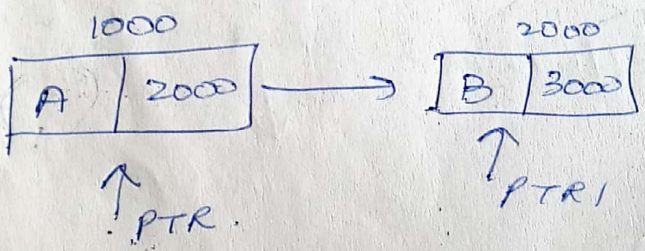
PTR[INFO] = A ✓

INFO[PTR]

PTR = PTR[LINK] } [INCREMENT]

= PTR → LINK

e.g.



printf("%d", INFO[PTR]) → O/P → A

or

PTR → INFO

PTR → 1000

PTR[LINK]

LINK[PTR] } → 2000

or

PTR → LINK

INCREMENT

PTR1 = PTR → LINK

PTR1 = 2000

NOTATIONS

START → it contain the address of the first node in the list.

START=5

EMPTY LIST OR NULL LIST → when there is no element.
∴ START = NULL

INFO[K] → It is linear array, it contain information part.

LINK[K] → It is linear array, it contain next pointer field of a node of LIST.

NULL → Indicating the end of LIST.

Traversing a linked list

PTR → pointer variable, which points to the node that is currently being processed

LINK[PTR] → points to the next node to be processed.
OR
PTR → LINK

PTR = LINK[PTR] [It moves the pointer to the next node in the list]
OR
~~PTR → LINK = PTR~~
PTR = PTR → LINK
OR update PTR so that it points to second node]

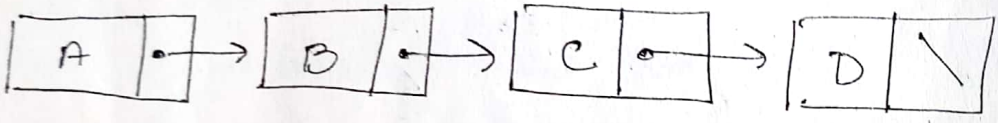
INFO[PTR] → information in the node.

Algo

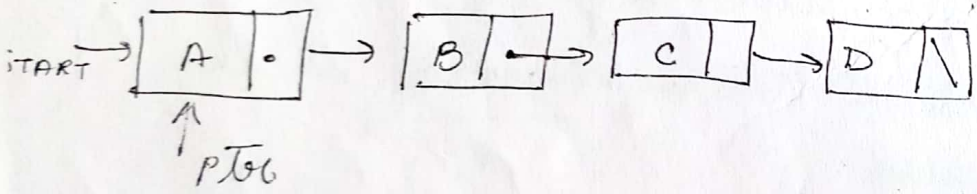
1. Set PTR = START
2. Repeat steps 3 & 4 while PTR ≠ NULL
3. Apply PROCESS to INFO[PTR]
4. Set PTR = LINK[PTR] // PTR now points to next node. PTR = PTR → Link
   [End of Step 2 loop]
5. Exit

E.G

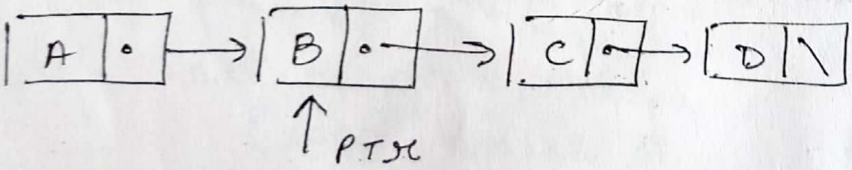
1. Initial list

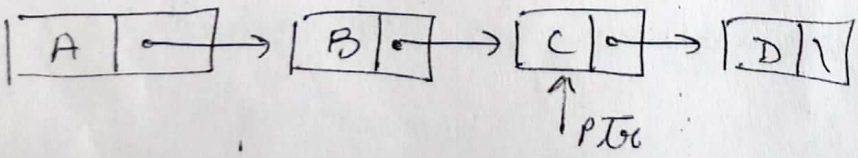


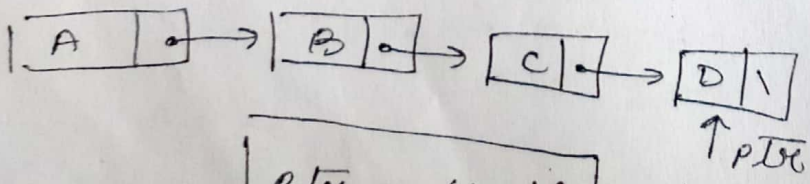
2. Set Ptr = Start



Repeat while Ptr ≠ Null







Ptr = Null

SEARCHING A LINKED LIST

SEARCHING →

LIST IS UNSORTED

9) Searching in a sorted list

1) SEARCHING IN A UNSORTED LIST

→ This algo searches ITEM in list by traversing through the list using a pointer variable ptr & comparing ITEM with the content of Info[PTR] of each node, one by one.

→ We continue this procedure till we reach the end of list or the ITEM is found.

LIST → it is a linked list in memory.

LOC → find the location of the node where ITEM first appears in LIST or

sets LOC = NULL

1) Set PTR = START
2) Repeat step 3 while PTR ≠ Null
3) If item = Info [PTR] then
   set LOC = PTR
   Exit.
   else
   PTR = link [PTR]
4) Set LOC = NULL (if search is unsuccessfull)
5) Exit

(2) SEARCHING IN ~~UNSORTED~~ SORTED LIST

→ We search for items in list by using the ~~list using the~~ ptr variable & comparing ITEM with the contents Info[Ptr] each node one by one. Hence we stop once item exceeds Info[Ptr].

(i) Set PTR = start
(ii) Repeat step 3 while PTR ≠ Null
(iii) if item < info [Ptr] then
set Ptr = Link [Ptr]
else if Item = Info [Ptr]
LOC = Ptr
exit
else LOC = NULL
exit.

EXAMPLE :—

AVAILABILITY LIST (GARBAGE COLLECTION)

→ It is also called as Availability stack.
→ It is pool or list of free nodes maintained with linked allocation are known as the availability list.
→ Whenever a node is to be inserted in a list a free node is taken from the availability list & linked to the former as required.
→ Similarly for the deletion of the node from the list causes it, return to the availability list.
→ It is further assumed that an available area of storage for this node structure consist of a linked stack of available nodes, where the pointer variable AVAIL contains the address of the top node in the stack.
→ In order to implement the procedure for inserting a node in an already existing singly linked list, it is necessary to have a procedure that supplies an unused node from the pool of available unused memory.

OVERFLOW UNDERFLOW

OVERFLOW CONDITION

→ When new data are to be inserted into a data structure but there is no available space i.e. the free-storage list is empty. This situation is called as overflow

Overflow occurs when AVAIL = NULL

→ Underflow refers to the situation where one wants to delete data from a data structure that is empty.

Underflow will occur with linked list when START = NULL & there is a deletion.

(1) Header node

→ It is a special node which lie at the front of a list. This node desnot represent an item in the list and also known as list header.

→ The info field of such a node could be used to keep global information about the entire list. For example no. of nodes can be store except header node in a list.

→ ADVANTAGE

The no. of item in the list may be obtained directly from header node without traversing the entire list.

DISADVANTAGE

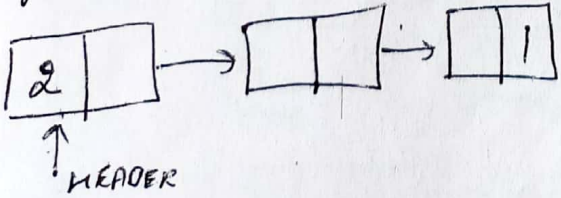
→ In this type of data structure more work is needed to add or delete an item from the list, since the count in the header node must be adjusted.

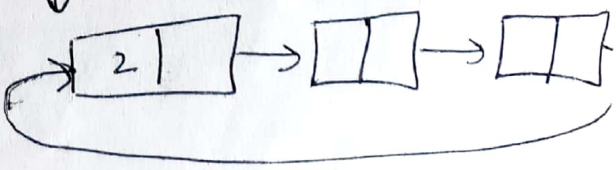
TYPES

GROUNDED HEADER LIST

CIRCULAR HEADER LIST

e.g





(2) TRAILER NODE → It is same as header node except.. that this node is placed at the end of

SINGLY LINKED LIST

INSERTION

ALGORITHM

INSERTION A NODE AT THE BEGINNING

1. [check for overflow]
   PTR = (Node *) malloc (sizeof(node))
   if (PTR == NULL)
   then: Print "overflow"
   exit

2. else:
   set PTR → INFO = item

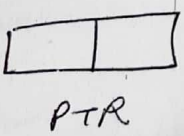
3. [check empty list]
   Ptr → link = NULL
   if (start == NULL) then start = Ptr return;

4. else: set ptr → Link = start

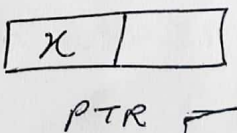
5. Set start = Ptr.

EXAMPLE

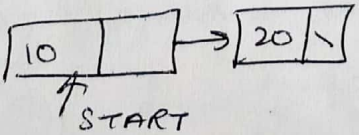
(i)



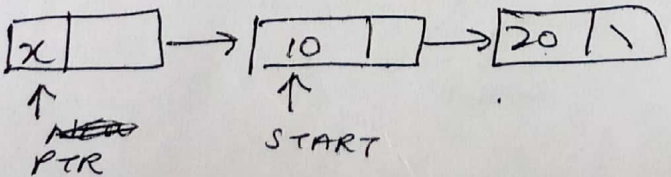
(ii)



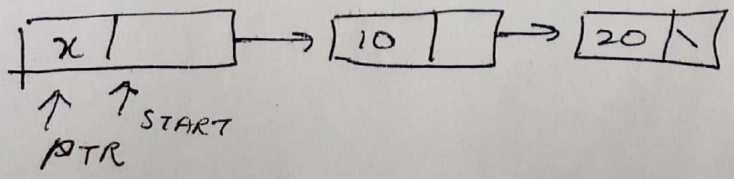
List is



(iii)



(iv)



CODE IN C

```
void insert (int x
{
Node *ptr;
Ptr = (Node*) malloc
        (sizeof(Node
ptr → info = item
if (start == NULL)
{
Ptr → link = NULL
}
else
Ptr → link = start
start = Ptr
}
```

(2) INSERTION OF NODE AT THE END

(1) [check for overflow]
PTR = (Node *) malloc (size of (node));
if (PTR == NULL)
then: Print "Overflow", exit.

(2) else:
set ~~ADD~~ PTR → INFO = ITEM

(3) set PTR → LINK = NULL

(4) [check for empty list]
if (START == NULL)
Then START = PTR
return

(5) [LOCATE THE LAST NODE]
else:
set TEMP = START

(6) Repeat Step 7 while TEMP → LINK != NULL
{

(7) set TEMP = TEMP → LINK [INCREMENT]
}

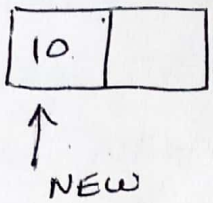
(8) set TEMP → LINK = ~~ADD~~ PTR
exit.

EXAMPLE:

NEW → INFO = ITEM

12

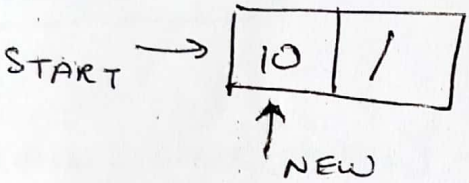
1)



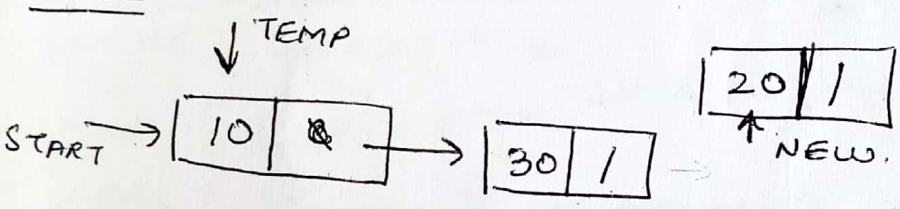
2) ~~PTT~~ NEW → LINK = NULL

3) IF

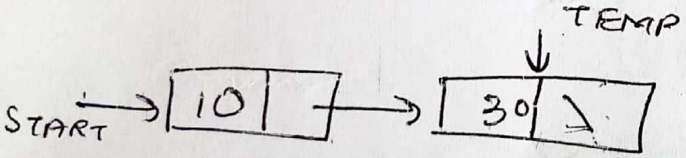
START = NULL



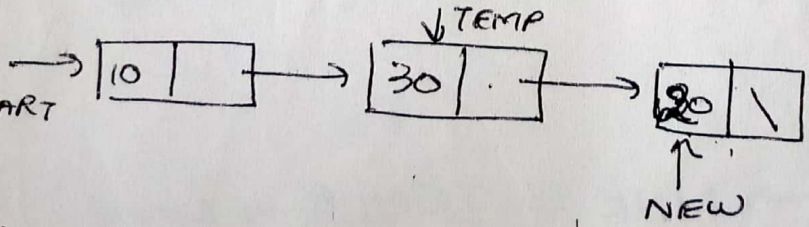
4) ELSE



TEMP = TEMP → LINK



5) TEM → LINK = NEW



INSERTION AT THE SPECIFIED POSITION

1. ~~[check for overflow]~~
   ~~if AVAIL = NULL, then~~
   ~~Print 'overflow'~~
   ~~exit~~

1. ~~else~~ [CHECK FOR OVERFLOW]
   NEW1 = (Node *) malloc (size of (node))
   ~~end if~~ if (NEW1 == NULL) Printf ("OVERFLOW"), exit
   else
2. NEW1 → INFO = ITEM
3. If START = NULL
   SET START = NEW
   SET NEW1 → LINK = NULL
   END IF
4. INITIALISE THE COUNTER (I) & POINTER (TEMP)
   Set I = 1
   Set TEMP = START
5. REPEAT STEP 6 & 7 UNTIL I < LOC
   {
6. SET TEMP = TEMP → LINK [UPDATE]
7. SET I = I + 1
   }
8. SET NEW1 → LINK = TEMP → LINK
9. SET TEMP → LINK = NEW1.

INSERTION INTO A LINKED LIST

⇒ If AVAIL = NULL → then algorithm will print the message OVERFLOW.

Variable NEW keep track of the location of new node

13

NEW = AVAIL

Notations used :

NEW = AVAIL →

AVAIL = LINK[AVAIL] →

OR

~~AVAIL → LINK = AVAIL~~

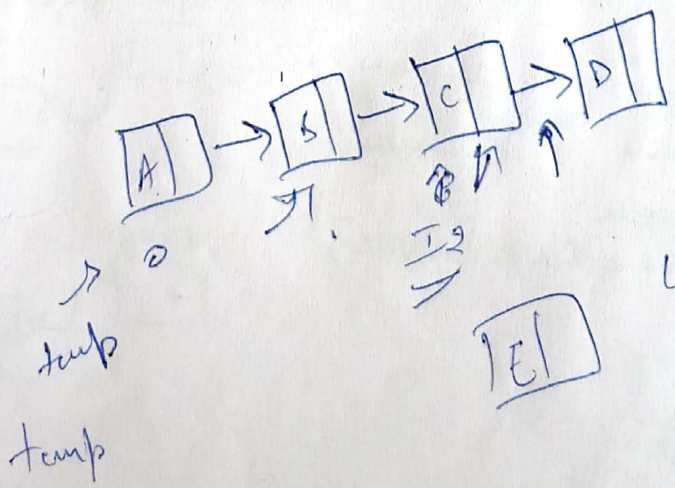
AVAIL = ~~LINK~~ AVAIL → LINK

INFO[NEW] = ITEM

OR

NEW → INFO = ITEM

loc



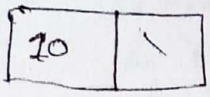
(1)

Reverse a linked list

```
void reverse()
{ struct node *prev, * current, *next;
  if(head == NULL)
  { print("list is empty");
    return;
  }
  current = head;
  prev = NULL;
  while(current != NULL)
  { next = current->next;
    current -> next = prev;
    prev = current;
    current = next;
  }
  Head = prev;
}
```

EXAMPLE

(1)

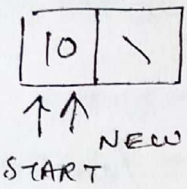


NEW → INFO = ITEM

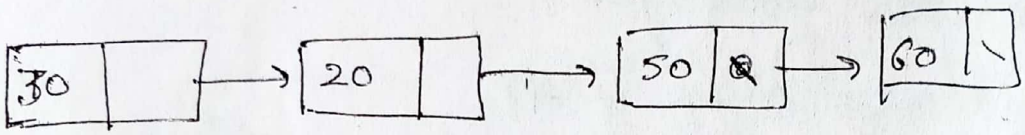
(15)

(2) If there is no item i.e. start = NULL

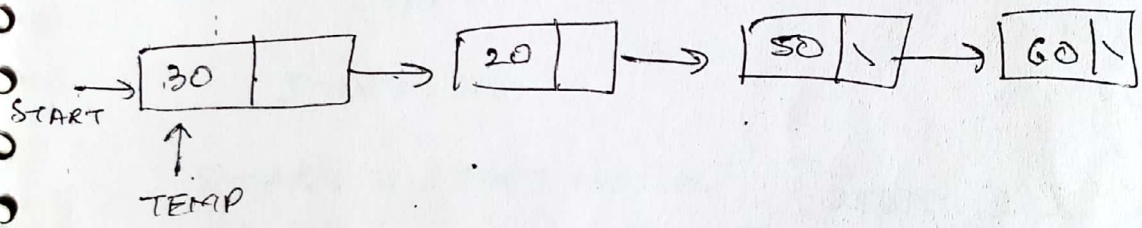
∴



(3) I = 1, if node to be inserted b/w ~~20~~ 50 & 60

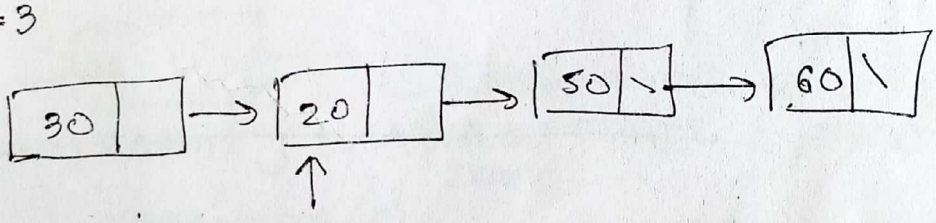


∴ LOC = 3



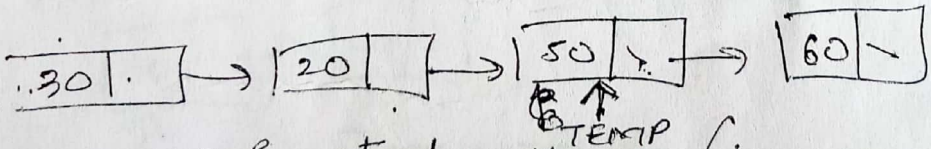
I <= 3

∴



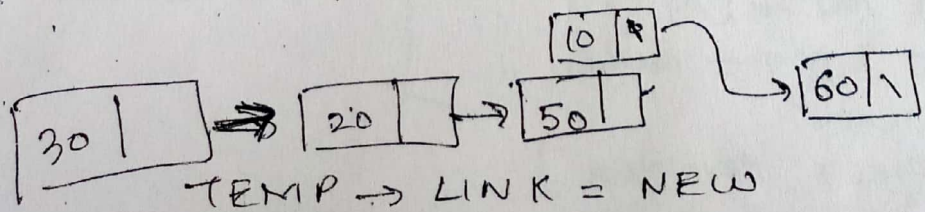
NOW I = 2

∴

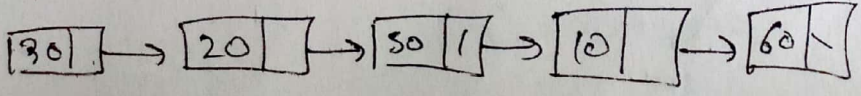


I = 3, as 3 is not less than 3 (i.e. I < LOC)

∴ NEW → LINK = TEMP → LINK



TEMP → LINK = NEW



~~DELETION AT THE FRONT~~

DELETION

1. DELETION AT THE FRONT
2. DELETION AT THE END
3. DELETION FROM THE PARTICULAR LOCATION

1) DELETION ~~AT~~ FROM THE FRONT/BEGINNING

1. [Check for underflow]
   If START = NULL then
   print "linked list is empty"
   exit
   end if
2. [Delete the node]
   P = START
   ITEM = ~~FRP~~ START → INFO
3. ~~Set START = START → LINK [INCREMENT]~~
4. ~~FREE NODE(P)~~
5. ~~EXIT~~

3) [LAST NODE TO BE DELETED]
   IF (START → LINK = NULL)
   FREE NODE(P)
   ~~FIRS~~ STRT = NULL
   ~~else~~ RETURN

   ] IF THERE IS ONE NODE IN LIST.

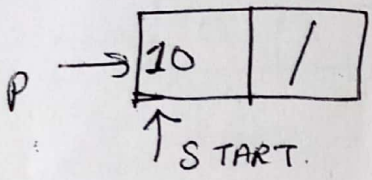
4) [UPDATE FIRST POINTER]
   ~~FIRST =~~ else: START = START → LINK
   FREENODE(P)

5) EXIT

EXAMPLE

CASE I WHEN THERE IS SINGLE NODE & DELETE FIRST NODE

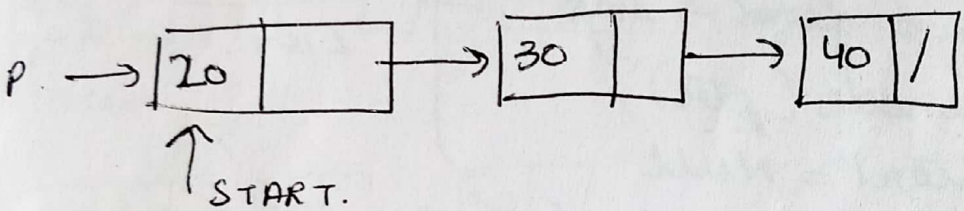
(16)



ITEM = 10

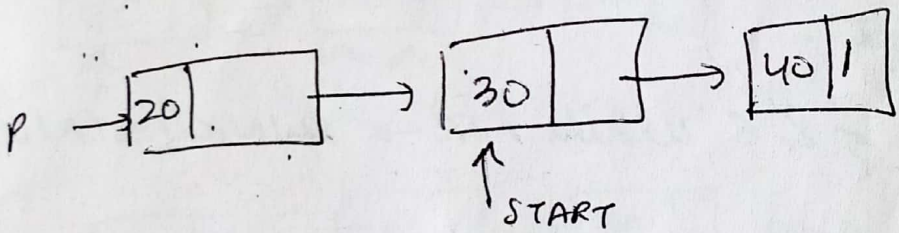
START = NULL

CASE-II WHEN FIRST NODE TO BE DELETED FROM LIST

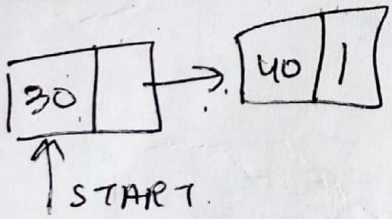


ITEM = 20

START = START → LINK



FREENODE (P)



(2) DELETION FROM END

1) [check for underflow]
   If start = NULL then
   Print List is empty
   return.
   end if

2) If start → Link = NULL then
   set PTR = start
   ~~set start = NULL~~
   item = ~~first~~ ptr → info
   freenode (ptr)
   start = Null
   return.

   ] WHEN SINGLE NODE IN LIST.

3) [locate the last node]
   ptr = ~~R~~ start.

4) Repeat steps 5 & 6 while PTR → ~~R~~LINK != NULL

5) Set LOC = PTR

6) set PTR = PTR → ~~NEXT~~ LINK

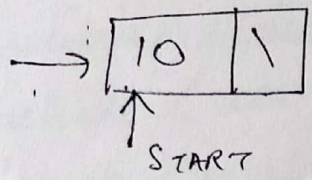
7) Set LOC → LINK = NULL

8) FREENODE (PTR).

9) EXIT.

~~CHAT~~ EXAMPLE

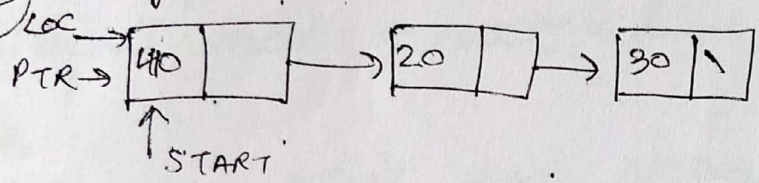
(1) If there is single node:

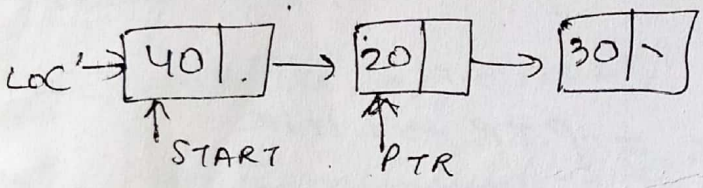


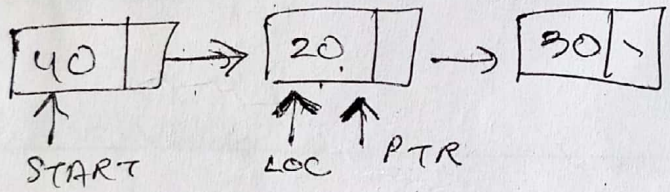
item = 10

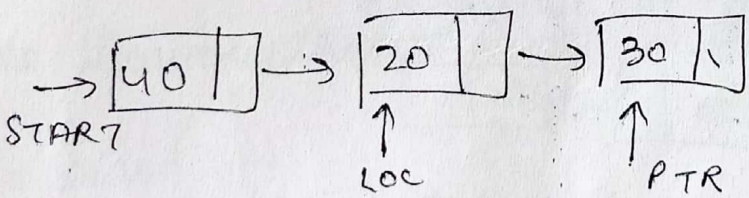
start = NULL

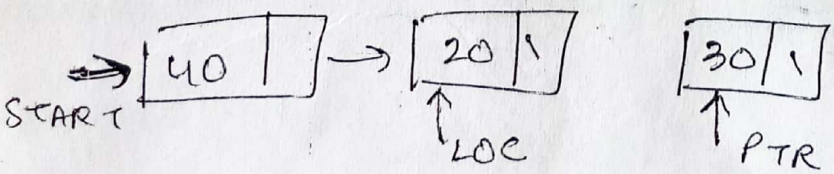
(2) If there are nodes in list.



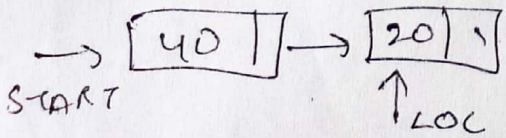








Freenode (PTR)

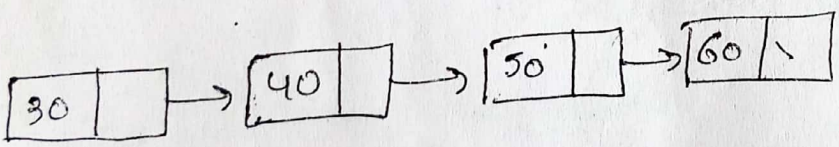


DELETION OF NODE FROM SPECIFIED POSITION

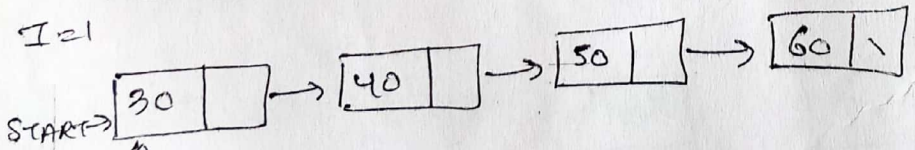
1. If (START = NULL) [check for underflow]
   Print ("List is empty)
   return
2. [Locate the node, J
   Set I = 1
   Ptr = START
3. Repeat step 4 to 6 until I < LOC
4. [set temp = PTR
5. PTR = PTR → LINK
6. SET I = I+1
7.] SET TEMP → LINK = PTR → LINK
8. FREENODE (PTR)

EXAMPLE: Delete 50 i.e. LOC=3

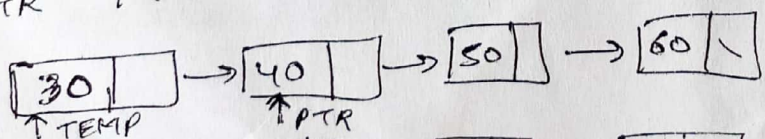
i)



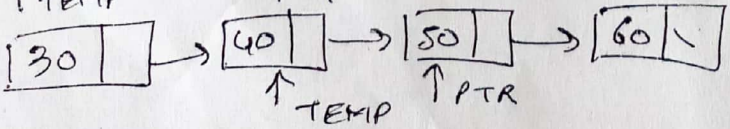
I=1



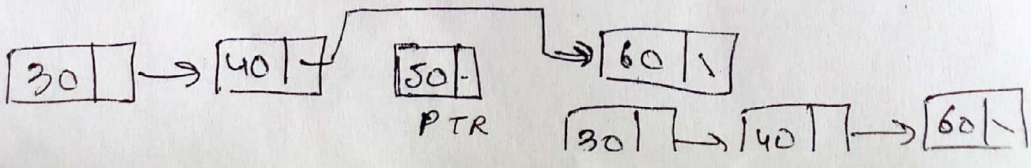
I < 3



I=2



I=3



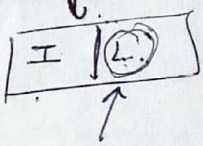
Program

TO insert node at first position & traverse

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>

struct node
{
int info;
struct node *next, *link;
};
typedef struct inode NODE;
NODE *start, *p, *new1;
void createemptylist(NODE *start)
{
  start = NULL;
}
void traversethelist(NODE *start)
{
  p= start;
  while(p!=NULL)
{
printf("%d\n", p->info);
  p= p->link;
}
}
```

self referencing structure

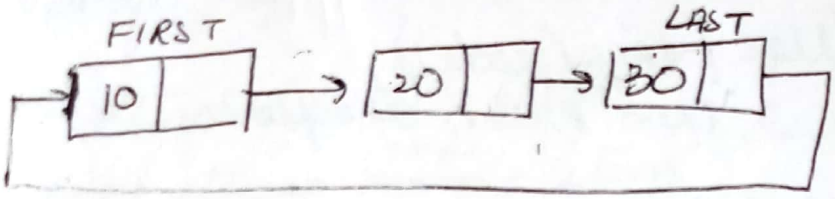


```
void insertatfront(int item)
{
NODE *new1;
new1 = (NODE *)malloc(sizeof(NODE));
new1 -> info = item;
if (start == NULL)
 new1 -> link = NULL;
 else
new1 -> link = start;
 start = new1;
}
void main()
{
int choice, item, ch;
char ch;
clrscr();
createemptylist(start);
do
{
printf("1. Insert element at begin");
 printf("2. Traverse the list");
 printf("3. Exit");
 printf("Enter Ur choice");
 scanf("%d", &choice);
 switch(choice)
{
case 1: printf("enter the item");
 scanf("%d", &item);
 insert at front(item);
 break;
```

```
case2: printf("traverse the list");
traverse the list(start);
break;
case3: return;
}
fflush(stdin);
printf("do u want to continue \n");
scanf("%c", &ch);
}
while((ch=='y')||(ch=='Y'));
getch();
}
```

CIRCULAR LINKED LIST

→ It is just singly linked list in which the link field of the last node contains the address of the first node of the list. i.e. the link field of the last node does not point to NULL rather it points back to the beginning of the linked list.



→ In ~~to this~~ circular list there is no end.

Code in C

```
struct node
{
  int num;
  struct node * next;
};
typedef struct node node;
node * start = NULL;
node * last = NULL;
```

INSERTION

1) Inserting a Node at the Beginning

i) [check for overflow]

~~If PTR = NULL, then~~
~~Print, Overflow~~
~~exit~~
~~else~~
PTR = (Node *) malloc (size(node)); // Allocation of memory at runtime
~~end~~ if (PTR == NULL), then Print Overflow, exit

ii) If start == NULL then
Set ptr → ~~A~~ info = item
set ptr → ~~A~~.link = ptr
set start = PTR
set last = PTR
end if

EMPTY LIST

3) else:
Set PTR → INFO = ITEM

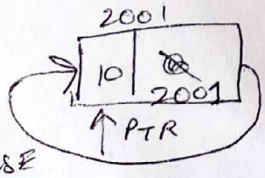
4) SET PTR → LINK = START

5) SET START = PTR

6) SET LAST → ~~NEXT~~ LINK = PTR

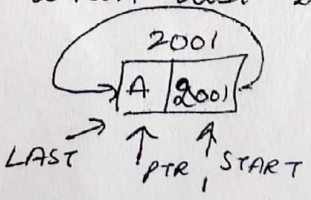
EXAMPLE:

(i) IF START = NULL

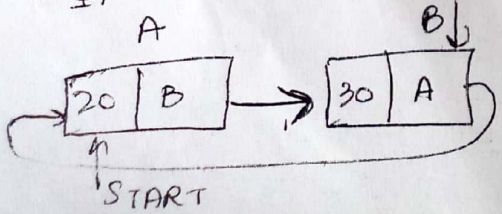


ELSE

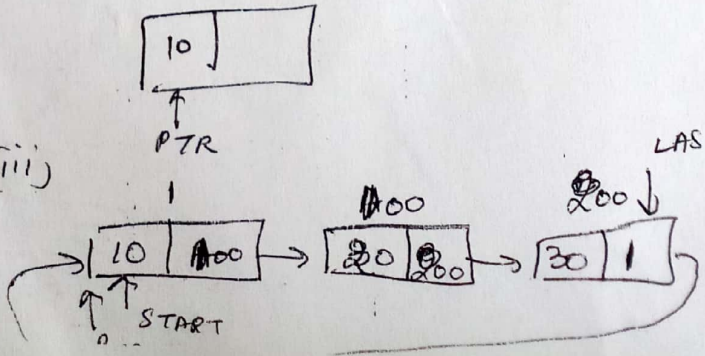
when list empty



(ii) IF



(iii)



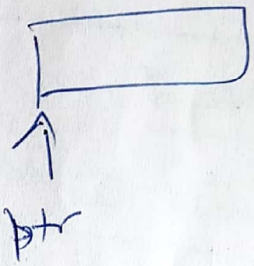
e) Inserting a new node at the end

i) ~~If ptr → NULL, Then~~
~~print overflow~~
~~exit~~
~~else~~

ii) PTR = (NODE*) malloc (size of (node));
~~end if~~ if (PTR == NULL), Then print Overflow, exit
else:

iii) If start = NULL, then
set ptr → ~~item~~ info = item
set ptr → link = ptr
set last = ptr
set start = ptr
end if

EMPTY LIST.



iii) else:
set ptr → info = item

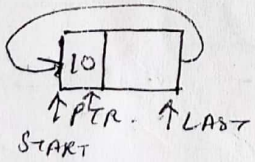
iv) set last → link = ptr

v) set last = ptr

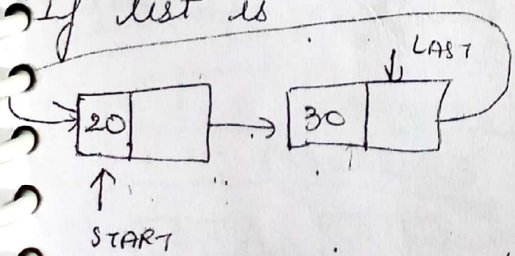
vi) set last → ~~ptr~~ link = start

EXAMPLE: when start = NULL

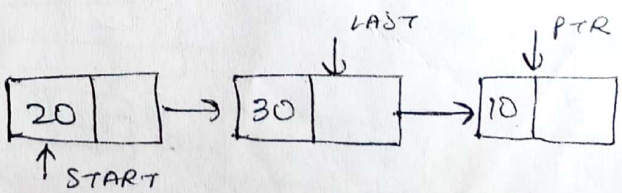
(i)



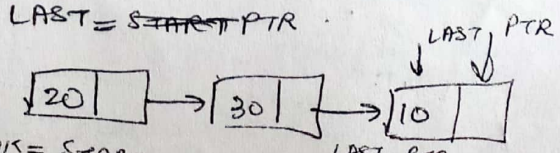
(ii) If list is



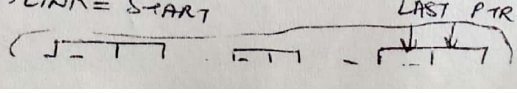
(iii) LAST → LINK = PTR



(iv) LAST = ~~START~~ PTR



(v) LAST → LINK = START

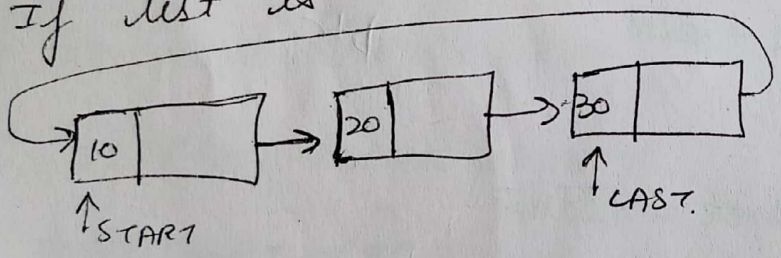


i) Deleting a node from the beginning

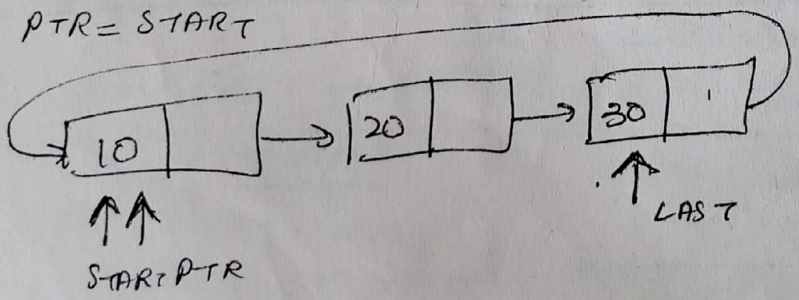
1) [check for underflow]
   if start = NULL then
   print 'circular list empty'
   exit
   end if
2) set ptr = start;
3) set start = start → link [UPDATE]
4) ~~print~~, set last → ~~next~~ LINK = start
5) free (ptr)

EXAMPLE:

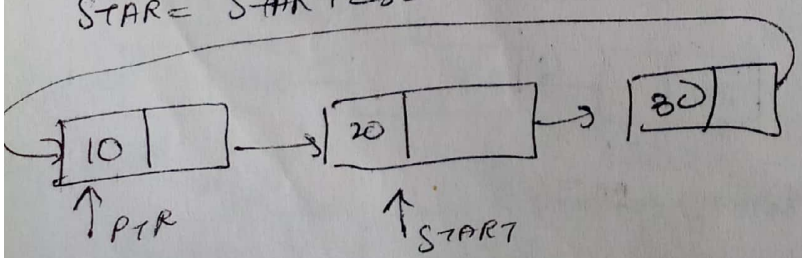
If list is



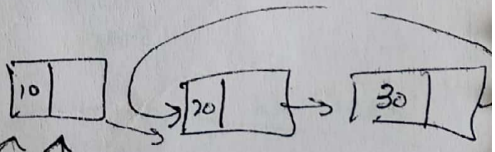
PTR = START



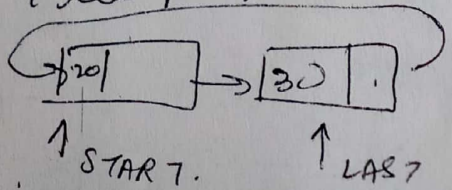
START = START → LINK



LAST → LINK = START



Free (ptr)



Deleting a node from the end

[Check for underflow]

if start = NULL then

print 'CIRCULAR list is empty'

exit

end if

set ptr = start

Repeat step 4 & 5 until

ptr->LINK != ~~NULL~~ START

set ~~ptr~~ temp = ptr

set ptr = ptr -> ~~next~~;

~~print "element deleted"~~

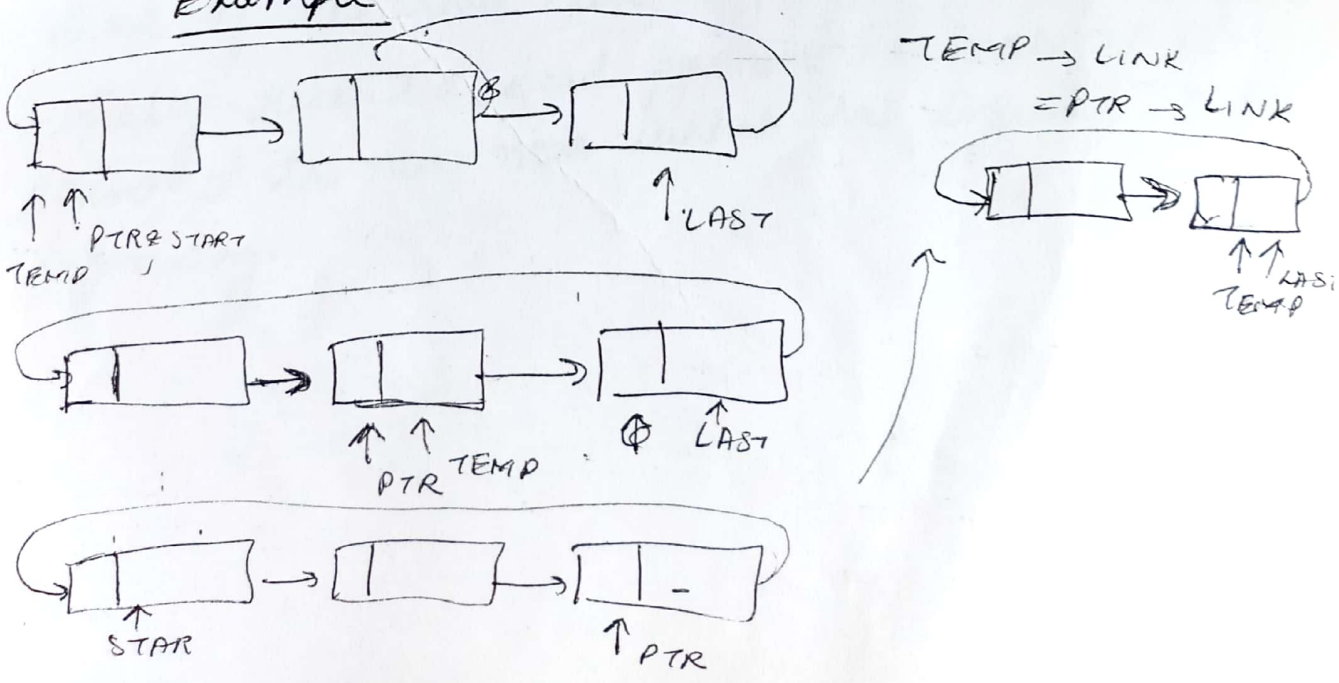
~~ptr->~~

Set temp -> link = ~~ptr~~ ptr -> ~~next~~ LINK

set last = temp

~~return start~~ Free (ptr)

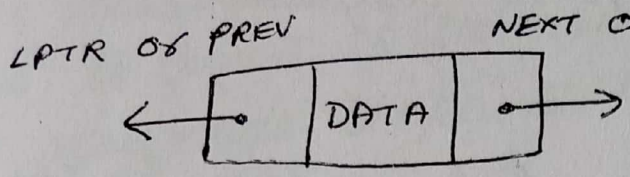
Example



DOUBLY LINKED LIST

L-6

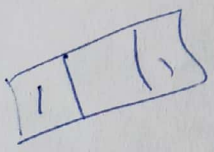
A doubly linked list is one in which all nodes are linked together by multiple number of links which help in accessing both successor node & predecessor node from the given node position. It provides bi-directional traversing.



INSERTION

(5) INSERTION A NODE AT THE FRONT

(i) Allocate memory for the new node
(ii) Assign value to the data field of the new node
(iii) Assign LEFT & RIGHT links to NULL
(iv) Make the RIGHT link of the new node to point to the head node of the list and make LEFT link of the head node to point to the new node
(v) Finally reset the head pointer i.e. make it to point to the new node which has inserted at the beginning.



i) [check for overflow]

~~If NEW1 = NULL then~~
~~Print overflow else~~
~~exit~~

1) ~~else~~
NEW1 = ((NODE*) malloc (sizeof(node));
if (NEW1 == NULL) then Print "overflow", exit

2) New1 → info = item

3) New1 → LLink = NULL // Since this is the first node & its LPTR field is NULL

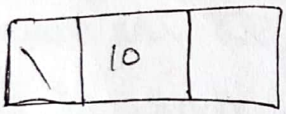
4) New → R link = RStart

6) Start → LLink = New1
Start = New1

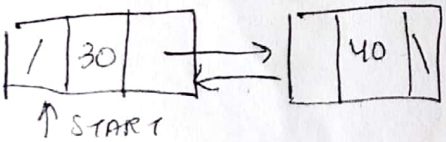
5) Exit.

For e.g.

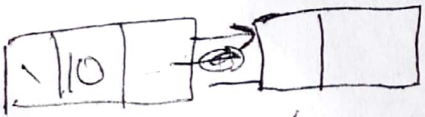
NEW → INFO = ITEM



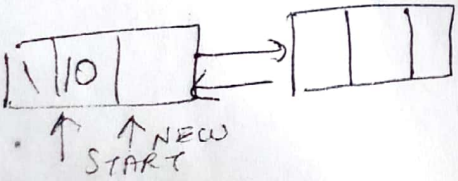
Suppose list is:



New → Rlink = Start



Start → Llink = New



(2) Insertion at the end

i) Allocate memory for the new node
ii) Assign value to the data field of the new node
iii) Set the LEFT & RIGHT links to the new node.
iv) If the list is not empty then traverse the list till the last & make the RIGHT link of the last node to point to the new node & LEFT link of the new node to point to the last node.

Algo:

1) if New1 = Null then
   Print overflow
   else exit

else
   New1 = ((NODE *) malloc (size of (node));

2) New → info = item

3) [check if list is empty]
   if (start = NULL)
   New → Rlink = NULL
   New → Llink = NULL
   start = New

4) else
   q = start

5) [Locate the last node]
   Repeat step 6 while ~~q → RLINK != NULL~~ q → Rlink != NULL

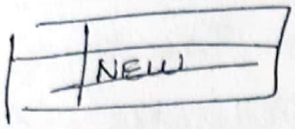
6) q = q → RLINK

7) New → Rlink = NULL
   q → RLINK = New
   New → Llink = q

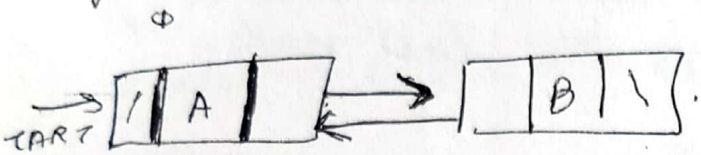
8) Exit.

Example.

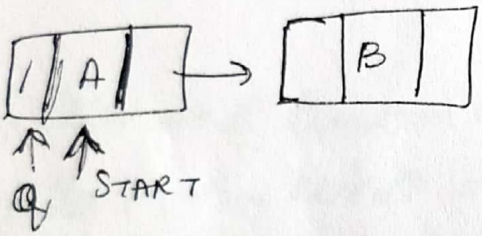
If start = NULL



If list is



q = start



(3) INSERTION IN THE MIDDLE (After the specified node)

L-10

i) if New = NULL then
    print "Overflow"
    exit
  else
New = (Node *) malloc (size of (node));

ii) New → info = item

iii) [Locate the specified node]
    q = start

iv) Repeat while (q → info != item)
        if (q = NULL)
        print ("specified node not found")
            return
        q = q → R link

v) New → L link = q
  New → R link = q → R link
  New → R link → L link = New
  q → R link = New

vi) Exit.

DELETION

(1) Deletion at front

(i) If the list is empty then display the message "Empty list - No deletion"

(ii) Otherwise make head pointer to point to the second node & if the second node is not null then make its LEFT link to point to NULL

(iii) Free the first node

Algo:

(i) [check for underflow]
```
if (start = NULL)
    Print ("list is empty");
    return;
```
(ii) [Delete the item]
```
item = start -> info
p = start
```
(iii) [check if last node of the list]
```
if (start -> Rlink = NULL)
    Free node (p)
```
(iv) [update first pointer]
```
if start = start -> Rlink
    start -> Llink = NULL
    freenode (p)
```
(v) Exit.

INSERTING A NODE AT THE BEGINNING

An algorithm to insert a new node at the begining (i.e. immediately after the head node) of the circular doubly linked list is given below:

ALGO

1. Allocate the memory for new node
2. Assign value to its data field
3. Make the RIGHT link of the head node to point to the new node & LEFT link of the new node to point to the head node. And similarly make the RIGHT link of the new node to point to the node immediately appearing after the head node and LEFT link of this node back to the new node.

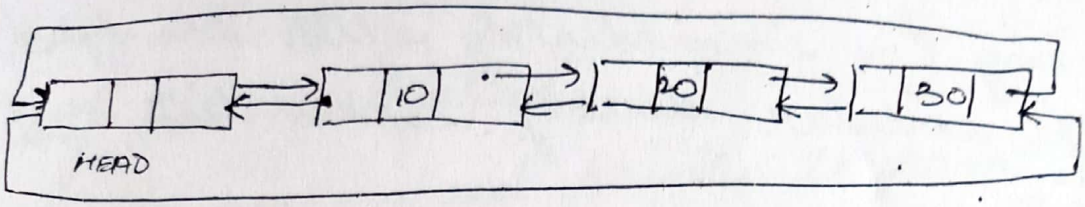
Code

```
insert (Node *head)
{
Node *ptr, *temp;
int item;
ptr = (Node *) malloc (size of (Node));
printf ("ENTER THE NO");
scanf ("%d", &item);
ptr -> info = item
else {temp = head -> right
head -> right = ptr
ptr -> left = head;
ptr -> right = temp;
temp -> left = ptr;
return (head);
}
```

```
if (head -> right == head
{
head -> right = ptr
ptr -> left = head
ptr -> right = head
head -> left = ptr
}
```

CIRCULAR DOUBLY LINKED LIST

circular doubly linked list with head node



(1) creation of circular doubly linked list

1. Allocate the memory to the node
2. Set its LEFT & RIGHT links to head
3. Set its data field to zero
4. Return the node.

code

```
Node * create_list()
{
  Node * head;
  head = allocate_node();
  head -> left = head -> right = head
  head -> num=0;
  return head;
}
```

(3) INSERTION A NODE AT THE END

Algorithm

An algorithm to insert a node at the end of the right most node of the circular doubly linked list below:

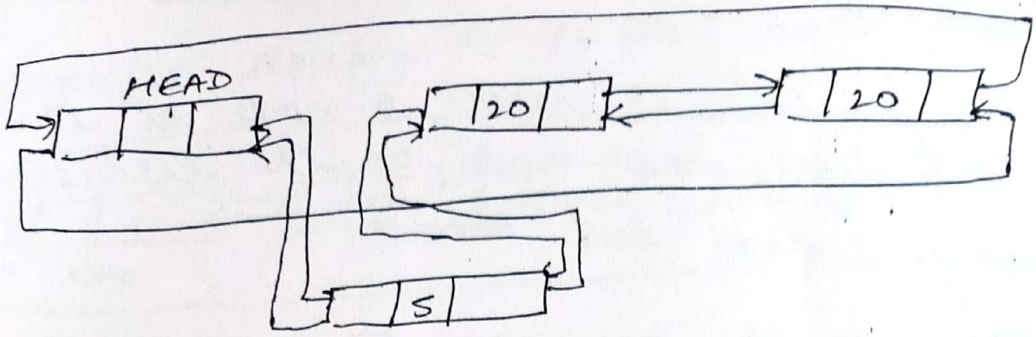
1. Allocate the memory to new node.
2. Assign value to its data field.
3. Make the left link of the head node to point to the new node & the RIGHT link of the right most node to point to the new node. And similarly make the RIGHT link of the new node to point back to the head node.

Code:

```
insert_end (Node * head)
{
  Node *ptr, *temp;
  int item;
  ptr = (Node *) malloc (sizeof (Node));
  printf ("ENTER THE NO");
  scanf ("%d", &item);
  ptr -> info = item;
  temp = head -> left;
  temp -> right = ptr;
  ptr -> left = temp;
  ptr -> right = head;
  head -> left = ptr;
  return (head);
}
```

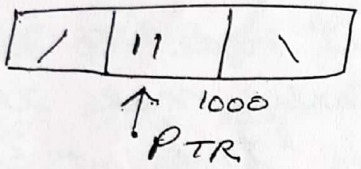
```
if (head -> right == head)
{
  head -> right = ptr
  ptr -> left = head
  ptr -> right = head
  head -> left = ptr
}
```

~~EXAMPLE~~



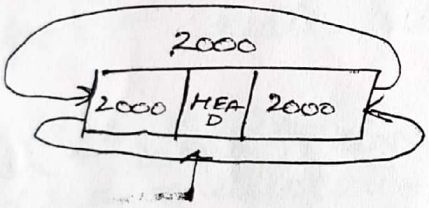
EXAMPLE

(1) create node

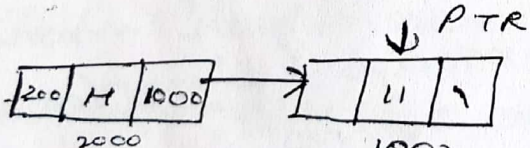


(2) Case I : when NO node in list exect

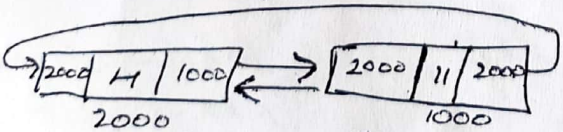
(a) header node



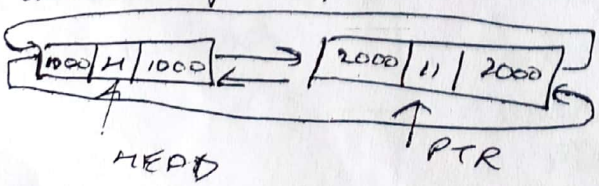
(b) head → right = ptr



(c) ptr → left = head
ptr → right = head

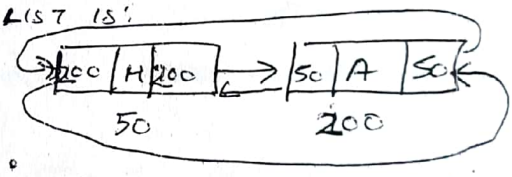


(d) head → left = ptr

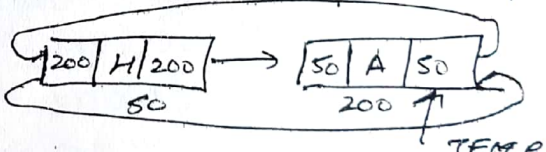


CASE II WHEN LIST HAVING NODES

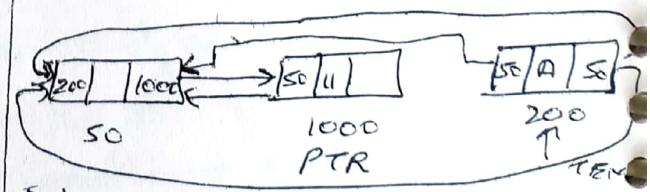
LIST IS:



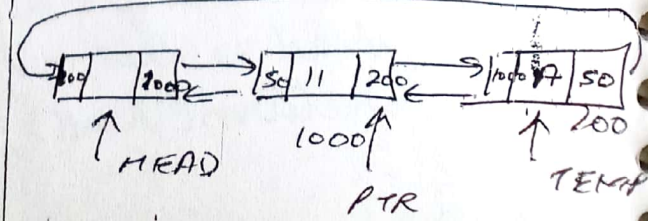
(i) TEMP = HEAD → RIGHT



(ii) HEAD → RIGHT = PTR
PTR → LEFT = HEAD

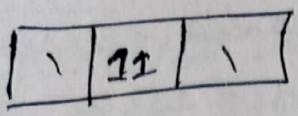


(iii) ~~PTR → LEFT~~
PTR → RIGHT = TEMP
TEMP → LEFT = PTR



EXAMPLE

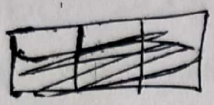
① create node



1000
PTR

case I :

→ when ~~node~~ list is empty, i.e. there is only header node



case II :

when list having node

(4) DELETING A NODE FROM THE BEGINNING

2-17

→ Deletion in a circular list is very simple since the current nodes successor & predecessor are immediately accessible.

Algorithm

An algorithm to delete a node from the beginning (i.e. the first node immediately after the head node) of a circular doubly linked list is given below:

(1) If the RIGHT link of the head node is pointing to itself then print the message "list is empty" & returns.

(2) Make the RIGHT link of the head node to point to the second node and the LEFT link of the second node to point to the head node.

(3) Free the first node & returns head node.

Code

```
del - beg (Node * head)
{
  Node * temp;
  if (head -> right == head)
  {
  Printf ("EMPTY LIST");
  return;
  }
  else
{
  temp = head -> right;
  Printf ("Deleted element is = %d \n", temp -> info);
  head -> right = temp -> right;
  temp -> right = temp -> left = head;
  free (temp);
} return (head);
```

5) DELETING A NODE FROM THE END: L-18

ALGORITHM

An algorithm to delete node from the end (i.e. rightmost node itself) of a circular doubly linked list is given below:

(1) If the RIGHT link of the head node is pointing to itself then print the message "Empty list and return

(2) Make the RIGHT link of the last but one node to ~~point~~ head node and similarly the LEFT link of the head node to point to last but one node on the list.

(3) Free the last node.

code

```
del_end (Node *head)
{
 Node *temp;
 if (head -> right == head)
  {
   printf ("EMPTY LIST");
    return;
  }
 else
 {
  temp = head -> left;
  printf ("Deleted element is = %d" temp -> info);
  head -> left = temp -> left;
  free (temp);
   return (head);
  }
}
```

ADVANTAGES OF DOUBLY LINK LIST

i) Insertion & deletion are simple as compare to other linked list.
ii) Efficient utilization of memory
iii) Bi-directional (both forward & backward) traversal helps in efficient & easy accessibility of nodes.

DISADVANTAGE

i) Uses two pointer, which results in more memory space requirement.