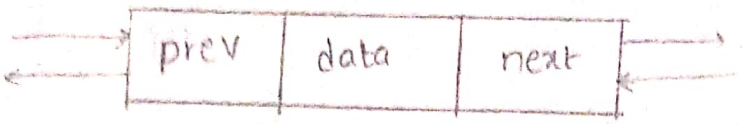
Double Linked List

Introduction:

A doubly linked list (DLL) is a linear data structure in which each node contains 3 components:

* prev → address of the previous node.
  data → data item
* next → address of next node

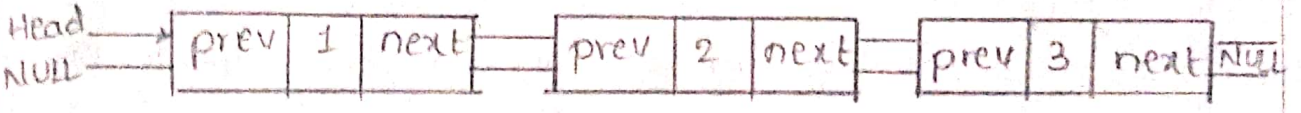


Each node is connected in both forward and backward directions.

Representation of doubly linked list:

The elements in a DLL are not stored in contiguous memory locations. Each node is connected using pointers.

Consider a doubly linked list with three nodes



Head pointer stores address of first node.

The first node:
prev = NULL (no previous node)
data = 1
next = address of second node

The second node:
prev = address of first node
data = 2
next = address of 3rd node

The third node:
prev = address of second node
data = 3
next = NULL

→ This forms bidirectional chain between nodes.

Comparision with singly linked list :

Feature	Singly linked list	Doubly linked list
Node structure	contains data + next pointer	contains data + prev pointer + next prev
Number of pointers	[contains data] One pointer - [next pointer]	[contains data] [prev pointer + next pointer] - two pointers
Node size	12 bytes	24 bytes
Traversal direction	Forward direction only	Forword and backward (bidirectional)
Deletion operation	Requires traversal to find previous node (O(n))	Direct deletion possible if node pointer given (O(1))

Memory representation of a doubly linked list in Data structures:

→ Represented using linear arrays in memory

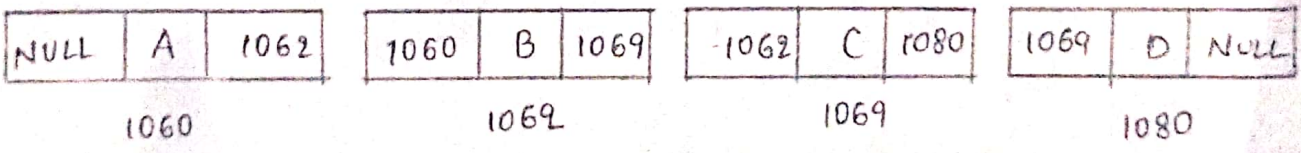
→ Each memory is stored in three components

Data → stores any value such as int, char, float, double, etc.

*Next → memory address of next element

* Prev → memory address of previous element

Ex:



1060, 1062, 1069, 1080 are the addresses where nodes A, B, C, D are stored.

Address	value	prev	Next
1060	A	-1	1062
1069	B	1060	1069
1069	C	1062	1080
1080	D	1069	-1

'-1' represents 'NULL'.

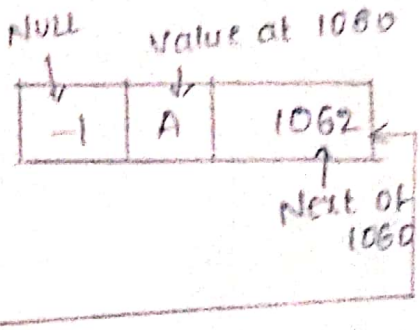
→ For node A,

address = 1060

value = A, prev = NULL

next value is at address 1062

since prev is NULL, 'A' is first node.

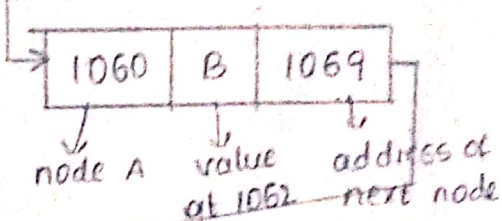


→ For node B,

It is stored at address 1062

prev = 1060 → previous node is A

Next = 1069 → Next node is C

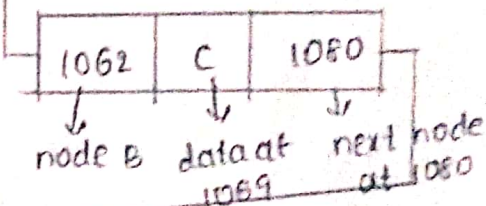


→ For node C,

'C' is at address 1069

prev = 1062 → 'B' is the previous node data.

next = 1080 → next value is at 1080.



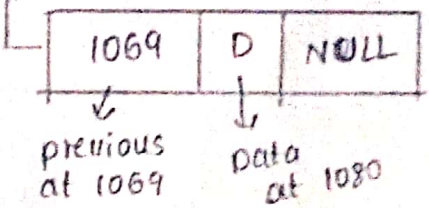
→ For node D,

'D' is at address 1080

It is the last node as Next = NULL

previous value of 'D' is at 1069

as Next = 1069



→ This memory representation shows the values need not be stored contiguously.

→ Traversing is done until '-1' in next of node is found (as it is the end of the list)

Basic operations Summary :

Operation	Time complexity	Special feature
Insert head/ tail	O(1)	Bidirectional links
Delete head/ tail	O(n)	Direct pointer access
Insert/Delete position	O(n)	Traverse from nearer end
memory per node	~24 bytes	data + 2 ptrs used in the node

→ Time complexity tells how fast or slow a program runs if the amount of data increases.

→ It is the measure of how much time an algorithm takes to run depending upon size of input data.

Operations on doubly linked list :-

Insertions :-

① DLL Insertion on doubly linked list at head :

Logic :

→ Create a newnode
→ Insert data into newnode
→ newnode→ prev = NULL
→ newnode→ next = head
→ If list is not empty
head→ prev = newnode
update head = newnode

Algorithm :

1. Create newnode //using dynamic memory allocation

newnode → data = given value // The data to be inserted

newnode → next = NULL

newnode → prev = NULL

2. If the list is empty (head == NULL)

head = newnode // head value is updated to newnode

3. Else (head != NULL) // list already exists

newnode → next = head

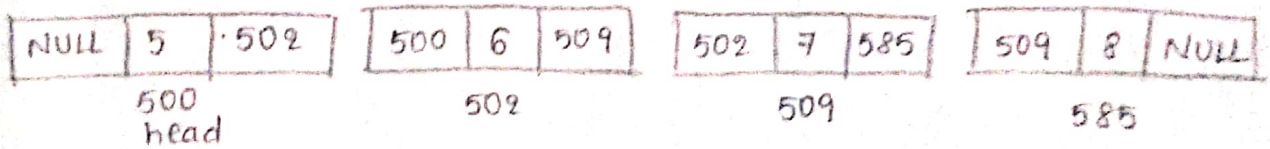
head → prev = newnode // newnode is made as first node

head = newnode // head is updated

4. Return head // head value that is changed is returned.

Example :

Let us consider a linked list



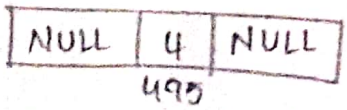
here head = 500 (as it is the first node)

1. Create a newnode

Let the data be 4

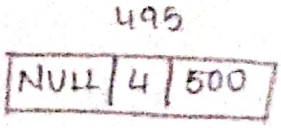
assume the address of newnode = 495

the newnode is

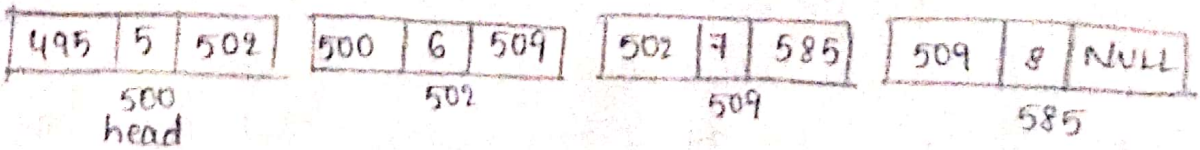


2. Since list is not empty and head = 500

newnode next value is at 500



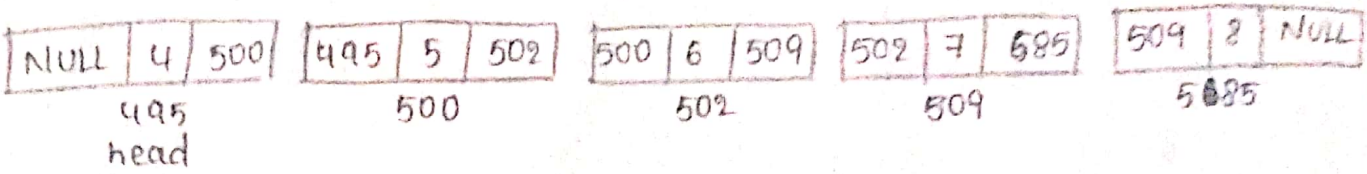
head → previous becomes 495



Now head is updated to newnode

=) head = 495

The list becomes, // The newnode is inserted at head



② DLL Insertion operation at Tail - Integer data algorithm

Logic:

→ Create a newnode
Insert data.
newnode → next = NULL
newnode → prev = NULL

→ Traverse list until last node 'p'
p → next = newnode
newnode → prev = p

Algorithm for insertAtTail function

1. create a newnode // using malloc()
newnode → data = value // The value to be inserted
newnode → next = NULL
newnode → prev = NULL // node without links is created

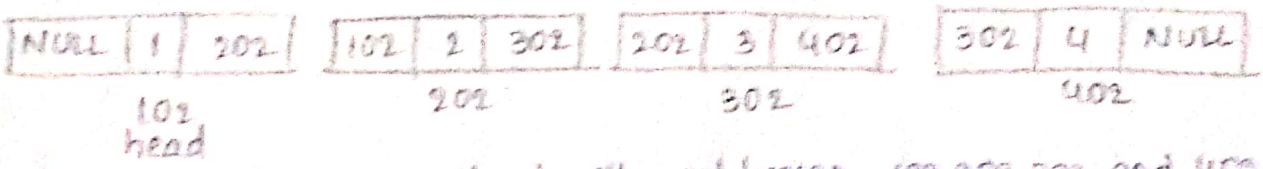
2. If list is empty (head == NULL)
head = newnode // head is updated i.e, newnode becomes first node

3. else // list exists
p = head // a pointer 'p' is assigned with head
while (p → next != NULL) // Traversal takes place to last node
p = p → next
p → next = newnode // last node next is newnode
newnode → prev = p // Before of newnode is the last one so the newnode becomes last node here (tail)

4. Return head // here head is unchanged and returned

Example:

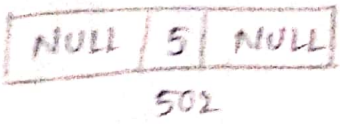
Consider a linked list:



1,2,3,4 values are stored at addresses 102, 202, 302 and 402

Here 102 is head and first node, 402 is the last node

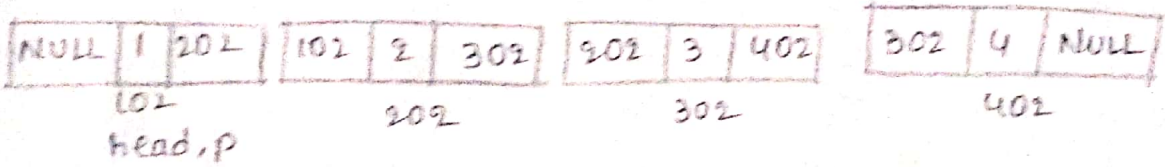
1. Create a newnode of data 5, assume its address as 502



=> newnode created

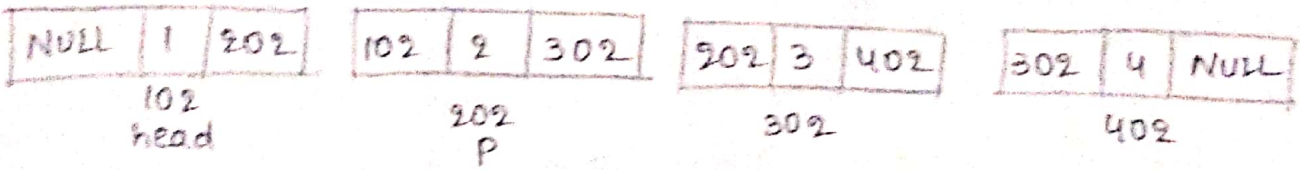
2. Since head is not null, p=head

=> p=102



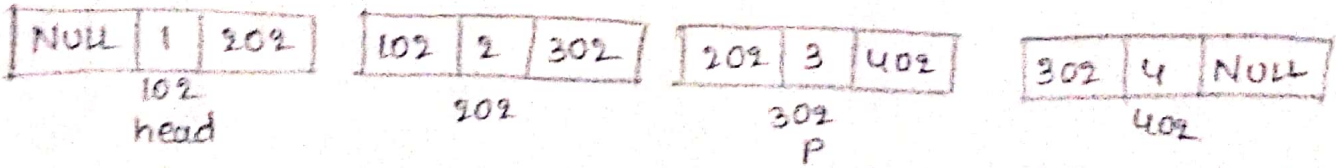
p->next = 202 != NULL -> True // Traversal begins

1st iteration :- p = p->next = 202



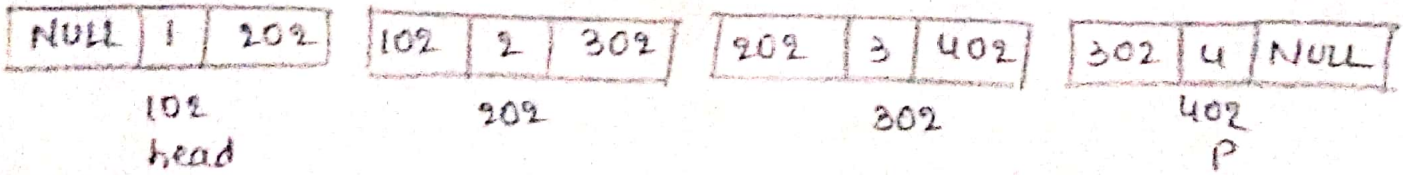
p->next = 302 != NULL -> True

2nd iteration :- p = p->next = 302



p->next = 402 != NULL -> True

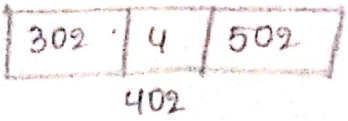
3rd iteration :- p = p->next = 402



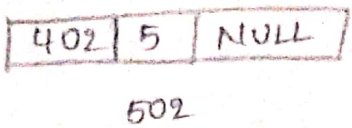
p->next = NULL != NULL → False // Traversal ends

Now, p = 402

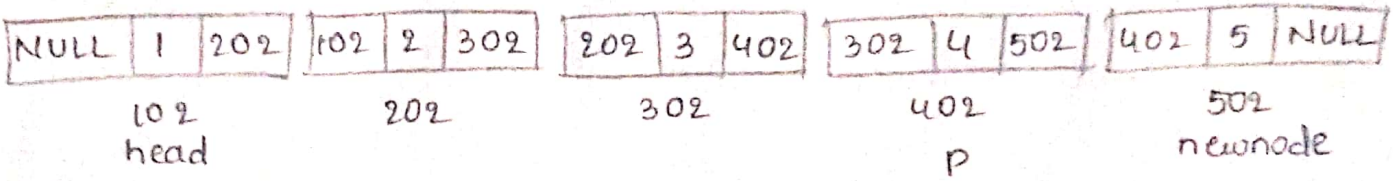
p->next = newnode = 502



newnode->prev = p = 402



The list is:



③ DLL Insertion Operation at n^{th} position – Integer data

Logic:

Create a newnode

Traverse list until position (n-1) 'p'

newnode next should be p next

newnode previous should be p

previous of p ~~next~~ is newnode

next of p next should be newnode

Algorithm for insertAtposition function

1. Create a newnode // using dynamic memory allocation
   newnode -> data = value // The value to be inserted
   newnode -> next = NULL
   newnode -> prev = NULL
2. If pos is zero (pos == 0)
   insertAt head // value to be insert at head
3. else
   p = head // p is assigned head
   for(int c=1; c<pos; c++) // Traverse to (pos-1) node (p)
   p = p->next

Set q = p->prev // (pos-1)th node will be q

newnode->next = p // newnode is placed before p

newnode->prev = q

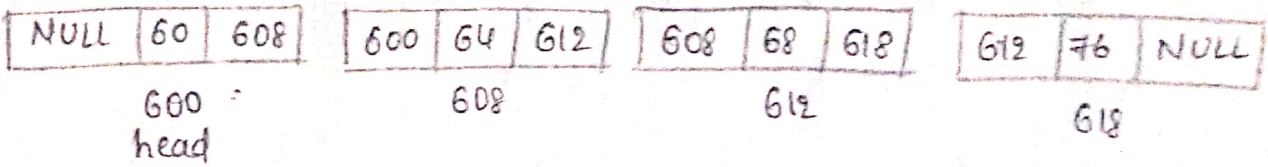
q->next = newnode

p->prev = newnode // newnode is placed between p and q, bidirectional links are arranged

4. Return head // the head is returned to main

Example:

consider a linked list

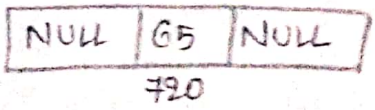


Here, head = 600

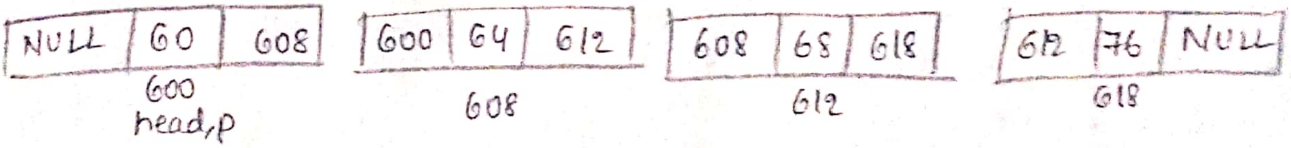
Let the position, pos = 3

1. Create a newnode of data, 65

assume the address is 720 => newnode is

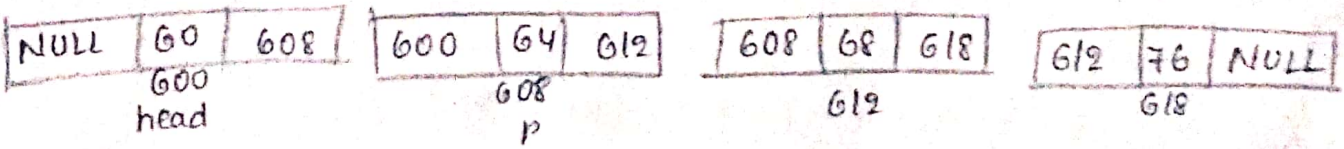


2. Since pos(3) != 0, p = head = 600 // Traversing to posth node starts



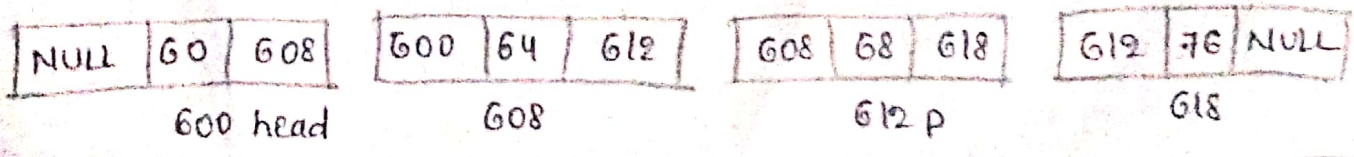
C = 1, pos = 3 C < pos => 1 < 3 -> True

So, p = p->next = 608



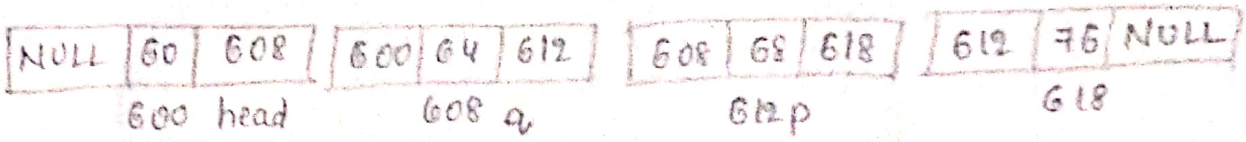
C++ => C = 1 C < pos => 2 < 3 -> True

So, p = p->next = 612



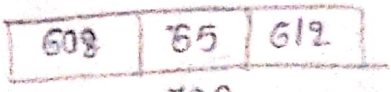
C++1, C<pos => 3<3 → False // Traverse ends

q = p→prev = 608



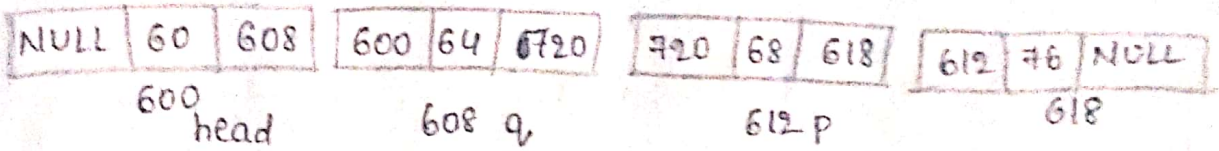
newnode → next = p = 612

newnode → prev = q = 608

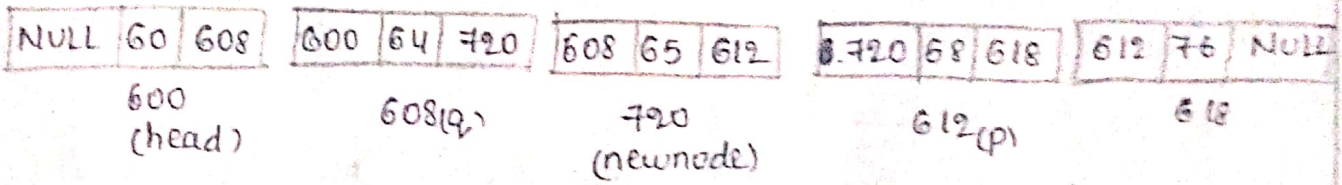


p → prev = newnode

q → next = newnode



The list is : // The node is inserted before pos 4 i.e., 3



(ii) DLL Insertion operation before Nth position - Integer data

Logic :

→ Traverse list to nth node temp

→ create a newnode

→ newnode previous should be temp previous

→ newnode next is temp

→ temp previous next and temp previous are changed to newnode

Algorithm for insert before position function :

1. Create a newnode // using malloc()

newnode → data = x

newnode → next = NULL

newnode → prev = NULL

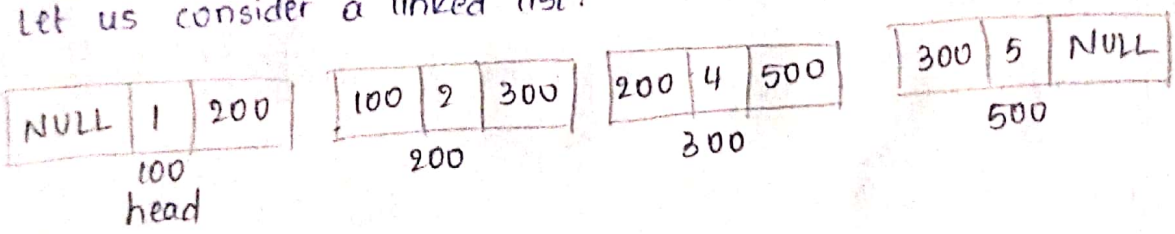
2. If pos==1 // Node is inserted before 1st node (head).
insertAthead // function call

3 else
p=head // head is assigned to pointer p
for(int c=1; c<pos-1; c++) // Traverse to (pos-1)th node using p
p=p->next
set q=p->prev // q is the (pos-2)th node
newnode->next=p
newnode->prev=q
p->prev=newnode
q->next=newnode // newnode is inserted between p and q by 4-step DLL linkage.

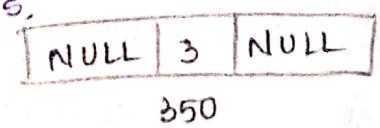
4. Return head

Example:

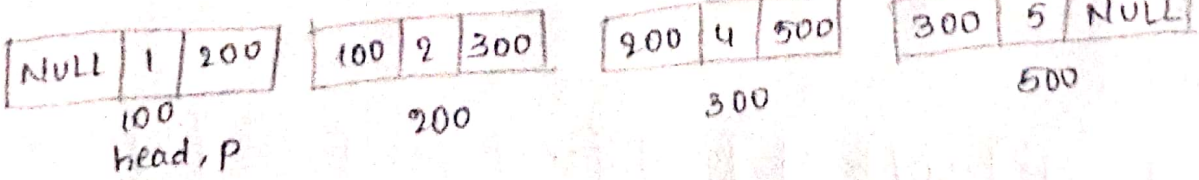
Let us consider a linked list:



1. Create a newnode of data=3
assume the address of newnode = 350
the newnode is,



2. head = 100,
Let pos = 4, pos-1 = 3
pos!=3 so, p=head=100



3. // Traversal starts

c=1, pos-1=3 c < pos-1 => 1 < 3 → True

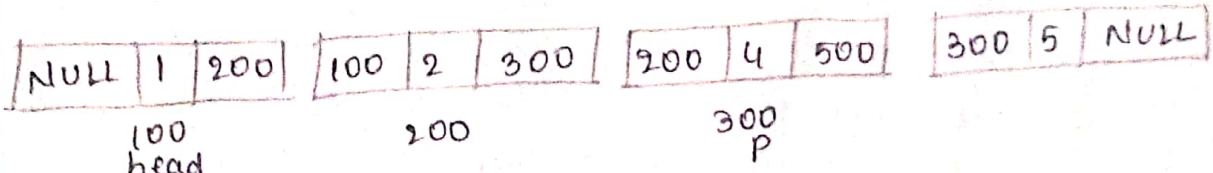
1st iteration: p = p->next = 200

c++ => c = 2

c=2, pos-1=3 c < pos-1 = 2 < 3 → True

2nd iteration: p = p->next = 300

c++ => c = 3

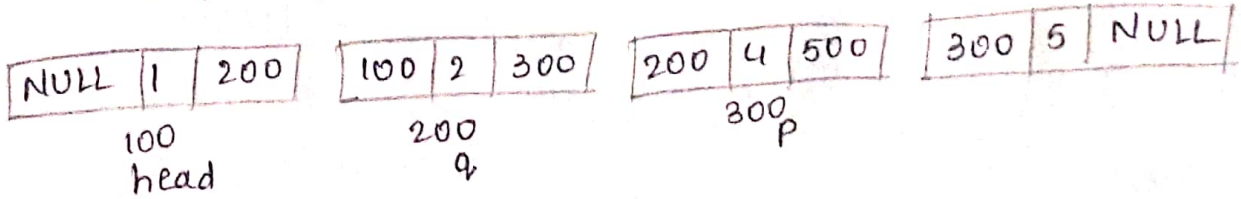


c=3, pos-1=3 c < pos-1 => 3 < 3 → False

// Traversal to (pos-1)th node ends

4. q = p->prev = 200

// q is (pos-2)th node



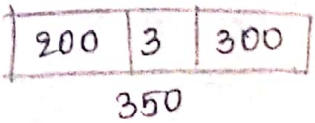
5. // 4 step DLL linkage

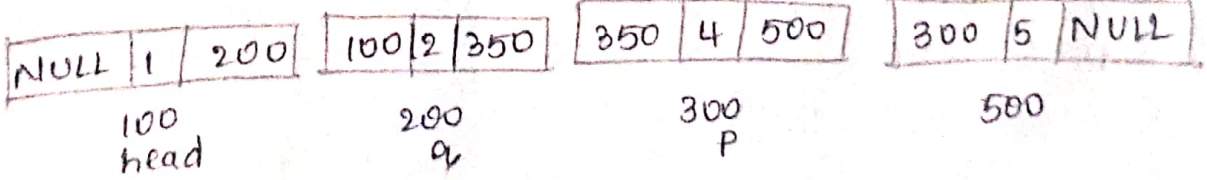
newnode->next = p = 300

newnode->prev = q = 200

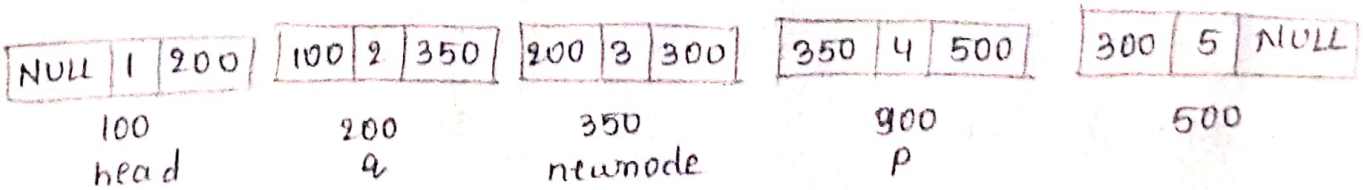
p->prev = newnode = 350

q->next = newnode = 350





The list is:



⑤ DLL Insertion Operation after Nth position - Integer data

Logic:

Create a newnode

Traverse list to n^{th} node next [$(n+1)^{th}$ node] temp

newnode next is temp

newnode previous is prev of temp

temp previous is newnode

next of temp prev should be newnode

Algorithm for InsertAfterPosition function

1. Create a newnode // using malloc

   newnode → data = x

   newnode → next = NULL

   newnode → prev = NULL

2. If (head == NULL) // list is empty

   head = nn // head is updated to newnode

   return head

3. p = head // head is assigned to p

   ct = 1

   while (p->next != NULL) // Traverse to last node

   ct ++

   p = p->next // The length of list is stored in ct

   if (pos == ct) // The node is inserted at tail

   InsertAtTail // Function call

   return head

4. p = head

   for (int c = 1; c < pos+1; c++) // Traverse to $(pos+1)^{th}$ node using p

   p = p->next

   set q = p->prev // q is now pos^{th} node

   q->next = nn

   p->prev = nn

   nn->prev = p

nn->next = q // newnode is placed after posth node by 4 step DLL linkage.

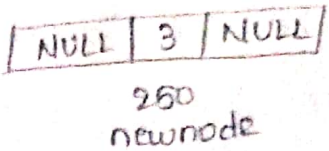
5. return head

Example :

Consider a list

1. Create a newnode of data 3

assume the address of newnode = 250



2. let pos = 2

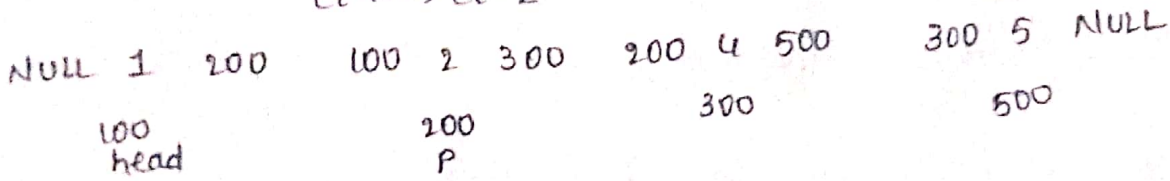
p = ct=1

p = head = 100

p->next != NULL => p->next = 200 != NULL -> True

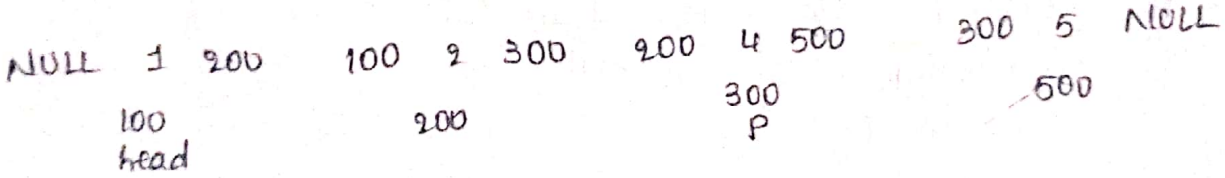
p = p->next = 200

ct++ => ct = 2



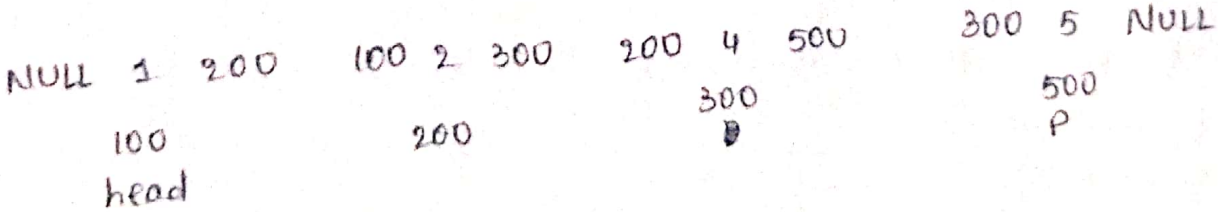
p->next = 300 != NULL -> True

p = p->next = 300 , ct++ => ct = 3



p->next = 500 != NULL

=> p = p->next = 500 => ct++ => ct = 4



p->next = NULL != NULL -> False so, ct = 4

pos=2

c!=pos => 4==2 -> False

so,

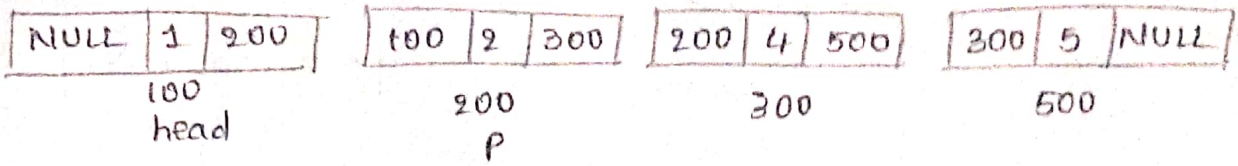
p=head=100

c=1 , pos+1=3

1st iteration: c<pos+1 => 1<3 -> True // Traversal to (pos+1) starts

p=p->next=200

c++ => c=2



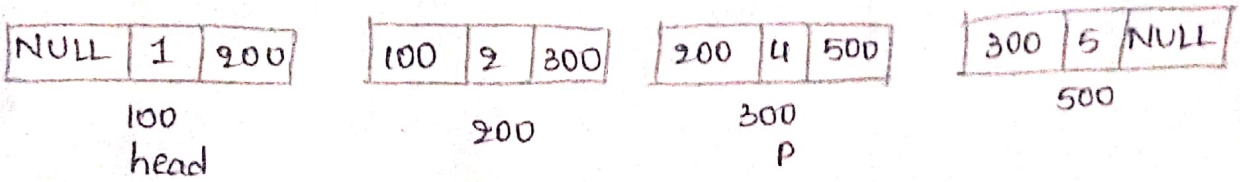
c=2, pos+1=3

c<pos+1 => 2<3 -> True

2nd iteration:

p=p->next=300

c++ => c=3



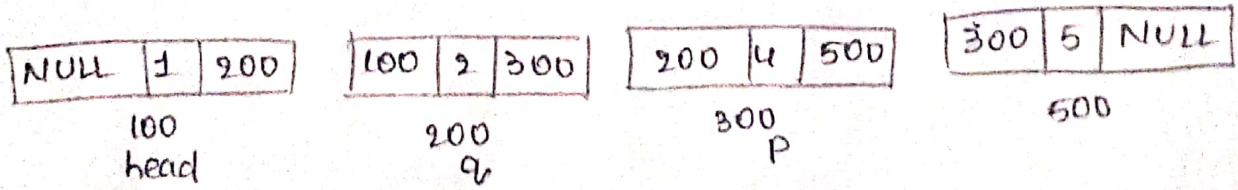
c=3, pos+1=3

c<pos+1 => 3<3 -> False

// Traversal to (pos+1)th node ends

q=p->prev

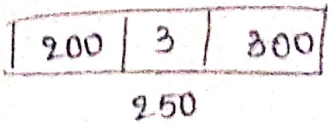
=> q=200



// 4 step DLL linkage

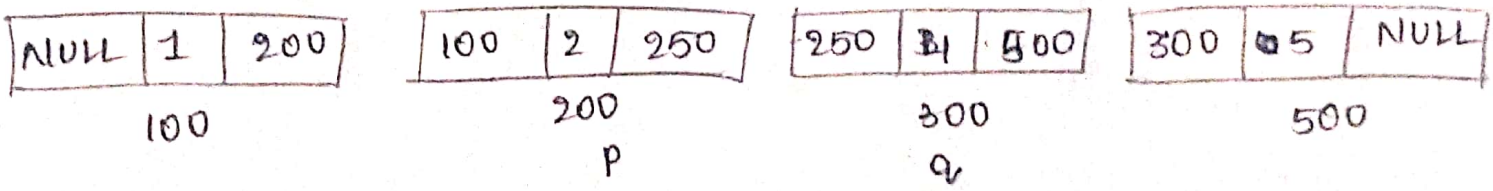
newnode->next=p=300

newnode->prev=q=200

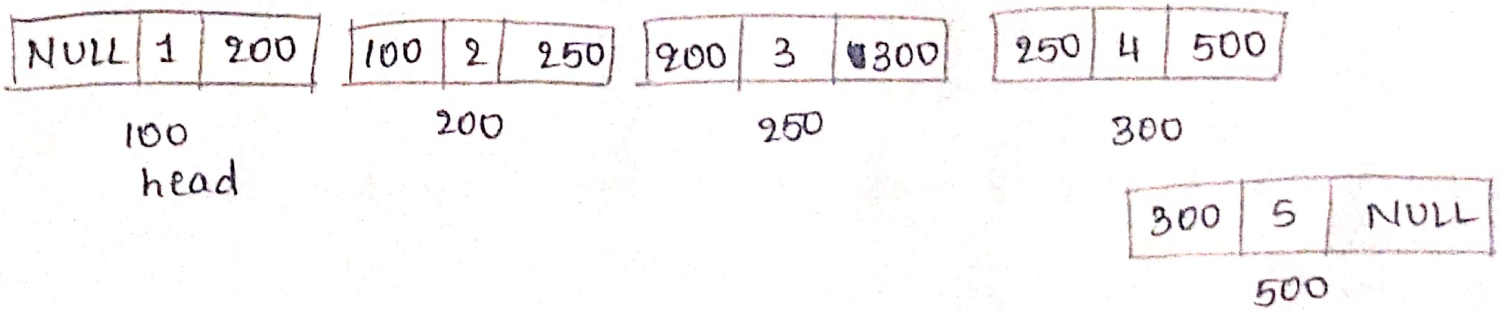


q->next = newnode -> 250

p-> prev = newnode = 250



The list is :



Traversals in Doubly linked list:

→ In doubly linked list, two types of traversals are possible.

1. Forward Traversal
2. Reverse Traversal.

Forward Traversal: [Head → tail]

→ Visiting each and every node starting from head to tail is called Forward traversal.

→ In simple words head to tail traversal is called as forward traversal.

Logic:

→ starts from head
→ visits each node and prints the data.
→ moves to next node.
→ Repeat until "null" is reached.

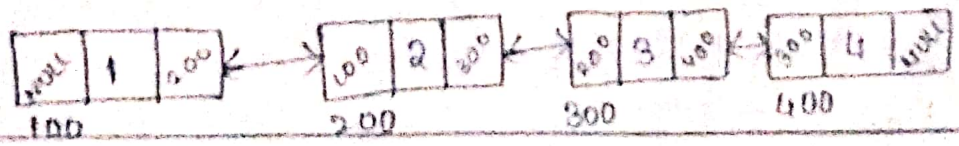
Algorithm:

Algorithm - Forward traversal.

1. Start
2. p = head // head is stored in "p".
3. if (p == NULL) // checks for empty list
   list is empty, traversal not possible // As no node is present No traversal takes place.
4. else
   while (p != NULL) // Traversing the list
   print "p->data" // prints the data while traversing
   p = p->next // moves to the next node
5. stop

Tracing with an Example:

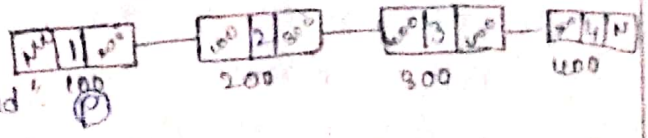
Let us consider a linked list as follows:



Here head = 100.

As the list is not empty, traversal is possible

In first Iteration; head = 100



$\Rightarrow$ p = 100 as "p = head"

and p != NULL i.e 100 != NULL (True)

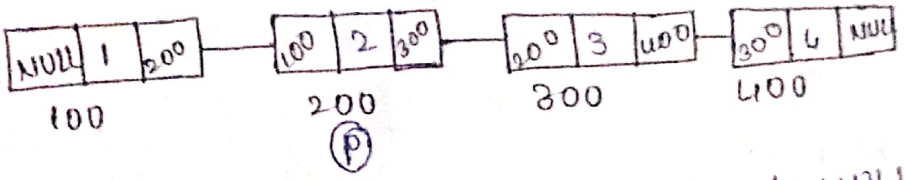
Output after 1st iteration:

1 <-> NULL

So now p = p->next

$\Rightarrow$ p = 200

after 1st iteration:



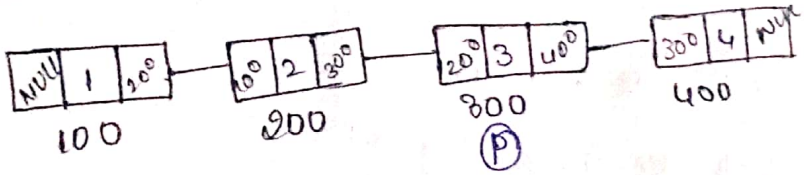
Second iteration; p = 200 p != NULL i.e 200 != NULL (True)

Output after second iteration:

1 <-> 2 <->

now p = p->next $\Rightarrow$ p = 300

after second iteration:



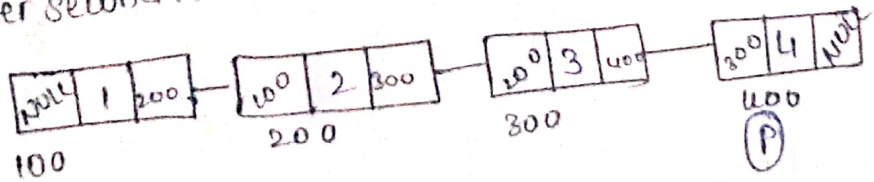
Third iteration; p = 300 p != NULL i.e 300 != NULL (True)

Output after second iteration:

1 <-> 2 <-> 3 <->

now p = p->next $\Rightarrow$ p = 400

after second iteration:



Fourth iteration; p = 400 p != NULL i.e 400 != NULL (True)

Output after fourth iteration:

1 <-> 2 <-> 3 <-> 4.

after fourth iteration:

p = p->next

⇒ p = NULL.

p != NULL ⇒ NULL != NULL (False) i.e condition fails.

→ loop execution stops

→ final output is 1 ↔ 2 ↔ 3 ↔ 4.

Backward-Traversal: [Tail to Head]

→ visiting each and every node starting from tail until we reach head is called backward traversal.

→ In simple words tail to head traversal is called as a Backward traversal.

Logic:

→ Starts from head and go to the last node.

→ Again from there on, visit each and every node in backward direction.

→ moves in reverse direction [tail to head]

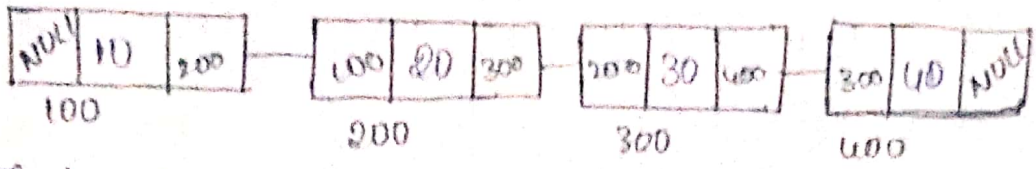
→ repeats until we reach "Head".

Algorithm:

Algorithm - Backward-Traversal.

1. Start
2. if (head == NULL) // checks for empty list
   list is empty, traversal not possible // NO node present so NO traversal.
3. else p = head // head is stored in p
   while (p -> next != NULL) // Traversed to last node.
   {
   p = p->next
   }
   do
   {
   print "p->data // Prints the data of node
   p = p->prev // Traverses backward
   } while (p != NULL)
4. Stop.

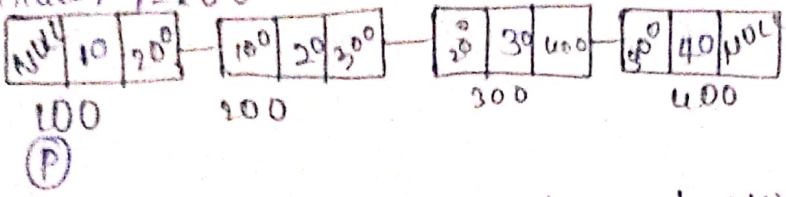
Tracing with an example:
consider a linked list as



Here head = 100 which is not NULL so the iterations of the code are as follows.

Initially we have to first traverse to the last node. for which the following iterations occur.

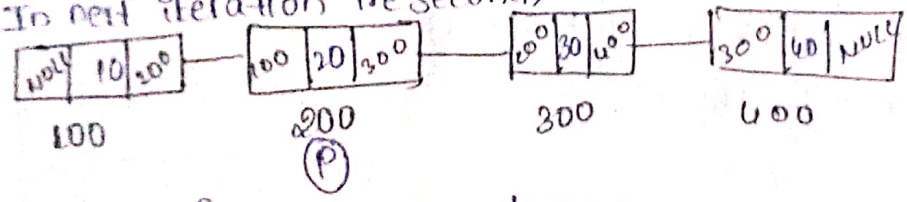
Initially P=100



P=100, p->next != NULL ⇒ 200 != NULL True

p = p->next ⇒ p=200

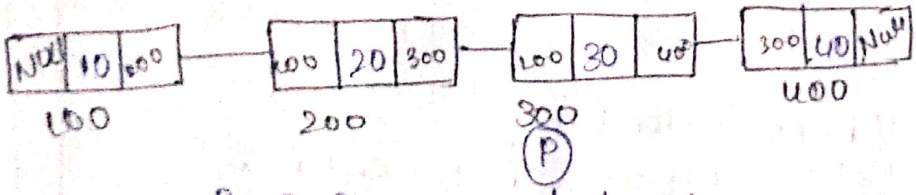
In next iteration i.e second;



P=200 p->next=300

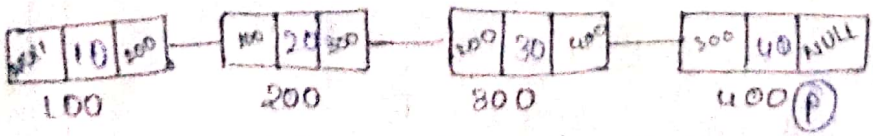
300 != NULL True

⇒ p=300.



P= 300 p->next= 400.

400 != NULL True ⇒ p= 400.

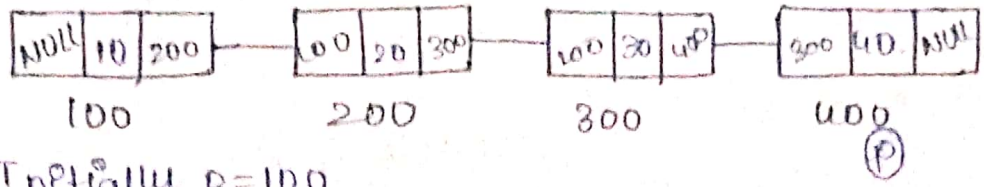


P= 400 p->next = NULL

NULL != NULL False

-> Iteration stops as we reach the last node.

→ As we reach the last node we will again traverse back and as we traverse backward the iterations are as;



Initially p=100

We're using do while so first loop statements are executed and then the condition is checked so acc. to that.

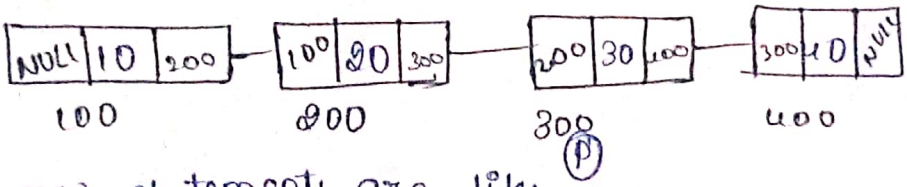
10 ↔ this is printed then 'p' is moved on node backward i.e p= p→prev

⇒ p = 300

Now condition is checked i.e p != NULL.

⇒ 300 != NULL (True)

So again loop is continued

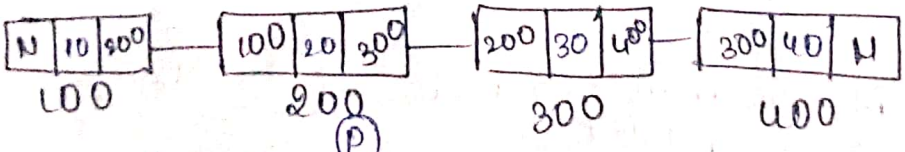


Now statements are like

40 ↔ 30 ↔

p = p→prev ⇒ p=200.

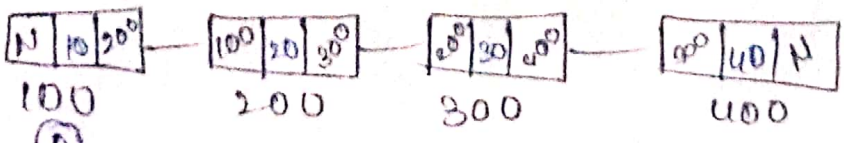
200 != NULL True.



Execution is like as follows: 40 ↔ 30 ↔ 20 ↔

p= p→prev ⇒ p= 100

100 != NULL ⇒ True



-Again statements in loop are executed as;

40 ↔ 30 ↔ 20 ↔ 10

p = p->prev ⇒ p = NULL

NULL != NULL -false

condition-fails.

→ Execution stops and final out put is

40 ↔ 30 ↔ 20 ↔ 10.

Deletions of Doubly linked lists:

1. Deletion at head position:

Logic:

→ Head node pointer is updated to it's next node i.e 2nd node.
→ 2nd node previous is Made "NULL".
→ Head is shifted to second node
→ Memory space of head node is freed.

Algorithm:

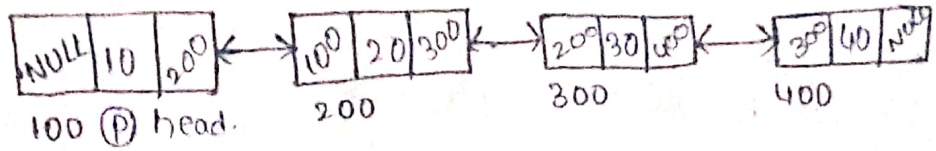
Algorithm - deletefat head

1. Start
2. if head == NULL // checks whether list is empty.
   exited. Deletion not possible // As list is empty. Deletion not possible.
3. else
   p = head // 'p' is initialised to head.
   head = head->next // head is moved to next node.
   if (head != NULL)
   head->prev = NULL // if more than one node exists the prev. link is updated.
   else
   {
   head = NULL // only one node exists.
   }
4. free (p) // memory freed

Stop

Tracing with an example:

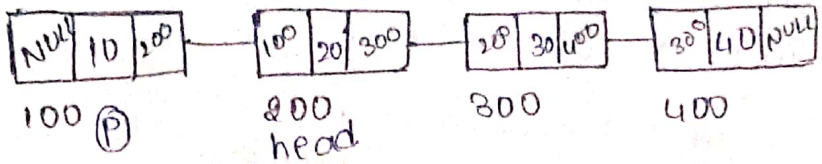
Consider a linked list as



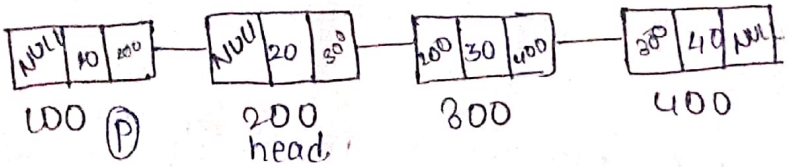
Initially p=100=head

To delete head node we should move head to it's it's next node.

⇒ head = head → next ⇒ head = 200.

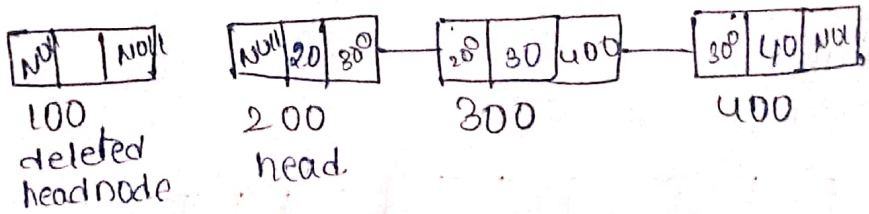


→ In the next step we should change head (present) prev to NULL to break the linkage between 1st node (Head node) and Second node.

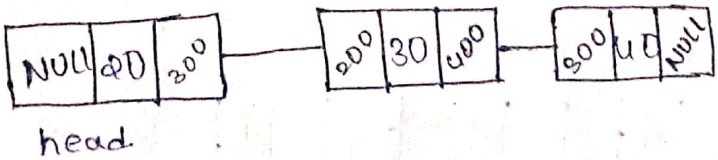


→ We should free the memory of previous head.

⇒ free(p);



After deleting a node at head; list is



2. Deletion at tail position:

Logic:

→ Traverse to the last node one node. and call it as 'q'.

→ last but one next node is taken as 'p' (i.e last node)

→ To delete last node make last but one (q) next as NULL

→ free(p) i.e last node.

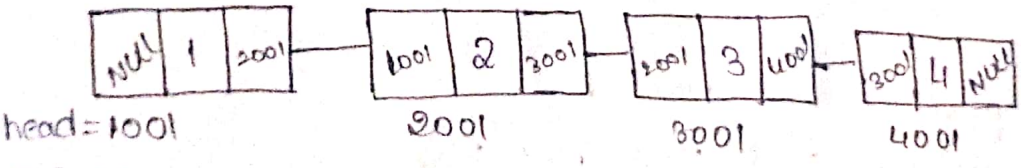
Algorithm:

Algorithm - deleteatlast.

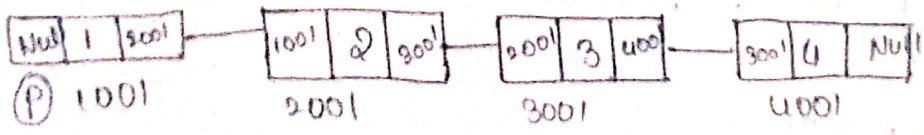
1. Start
2. if head = NULL // checks of Empty list
   exit deletion not possible // Deletion not possible as there are no nodes.
3. else
   p = head // 'p' is initialised to head
   while(p->next != NULL) // Traversal to last node.
   {
   q = p // 'q' is stores the value of p
   p = p->next // p is moved to the next node
   }
   if (q != NULL) // link updations when more than one node is present
   q->next = NULL
   else
   head = NULL. // when only one node is present head will be null if we delete the only node
4. Stop.

Tracing with an example:

consider a linked list as follows.



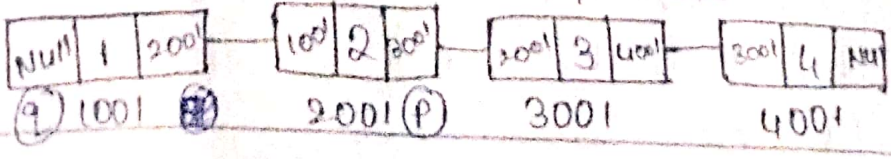
initially p = head



p->next = 2001 p->next != NULL => 2001 != NULL TRUE

q = p => q = 1001 p = p->next

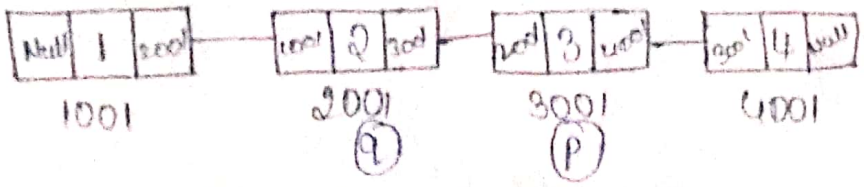
=> p = 2001



p->next = 3001 3001 != NULL (True)

⇒ q = p p = p->next

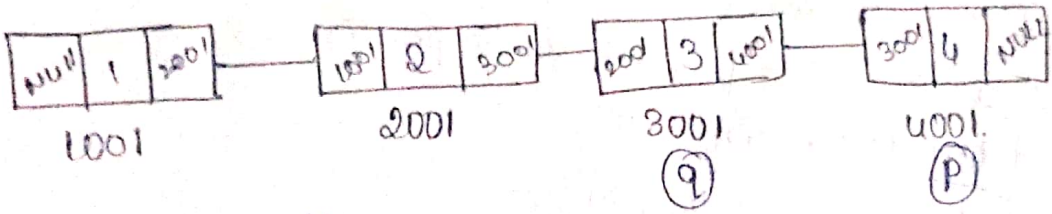
⇒ q = 2001 p = 3001.



p = p->next = 4001 4001 != NULL (True)

⇒ q = p p = p->next

q = 3001 p = 4001.



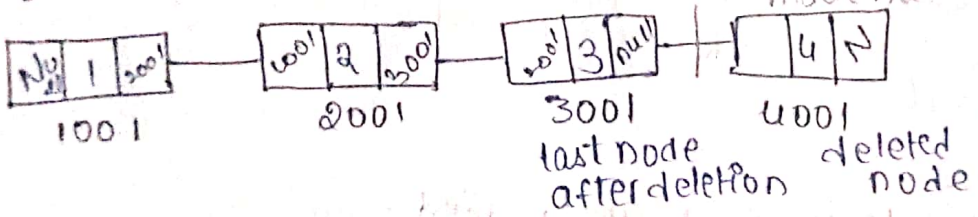
p->next = NULL NULL != NULL (false)

~~⇒ q = p~~ ⇒ NO updations in p, q.

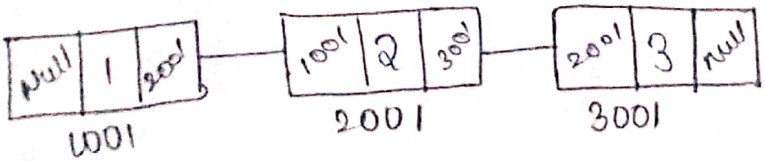
q = 3001 p = 4001.

q->next = NULL and free(p)

link is broken after q->next is made 'null' and 'p' is freed



After deletion the final list is;



3. Deletion at n^{th} position:

Logic:

→ Move to n^{th} node call it as p.

→ call $(n-1)^{th}$ node as q and $(n+1)^{th}$ node as r.

→ Make necessary link updations i.e.,

i) $(n-1)^{th}$ → next to $(n+1)$

ii) $(n+1)^{th}$ → prev to $(n-1)$

→ free / delete n^{th} node i.e., free (p).

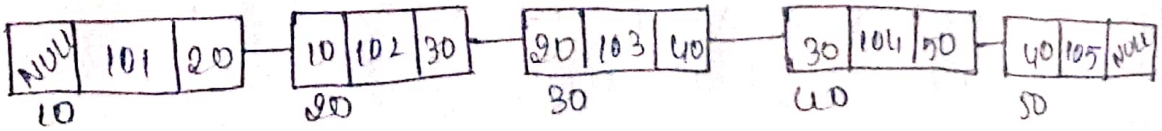
Algorithm:

Algorithm - deleteatposition

1. start
2. if head == NULL // checks for empty list.
   exit // No deletion possible if list is empty
3. else if pos = 1 // we have to delete at Pos '1' delete at head
   delete at head // function call.
4. else if pos = l // we have to delete at last position, delete at-tail.
   delete at tail. // function call.
5. else if pos < 1 or pos > l // failure cases
   exit the code.
6. else c = 1 // count is stored which is initialised to 1.
   p = head // 'p' is initialised to head.
   while (p->c != pos) // Link updations inside the loop to traverse to last.
   q = p
   p = p->next
   c++
   r = p->next // To delete the required node.
   q->next = r
   r->prev = q
7. free (p)
8. stop.

Tracing with an example:

consider

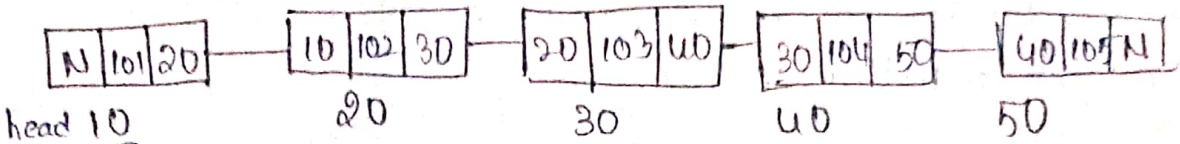


→ Head != NULL so we

→ let us assume that we have to delete a node at 3rd position

Here as pos != Head → we cannot use delete at head algorithm also pos != l (length of list so we can not use delete at tail algorithm as well.

Initially p=head and c=1



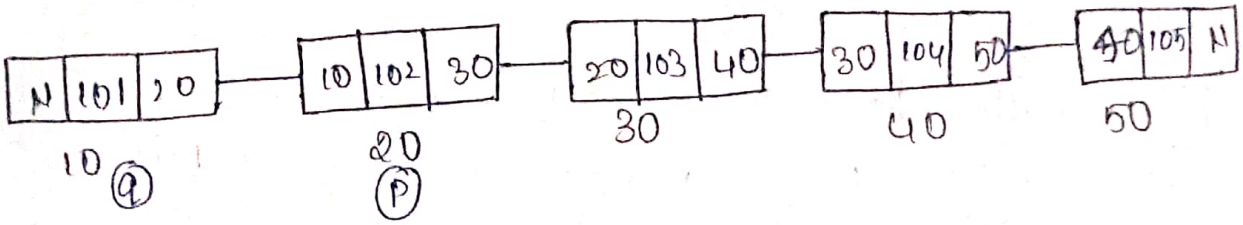
(P) c=1.

c != pos ⇒ 1 != 3 True

⇒ q = p ⇒ q = 10

p = p->next ⇒ p = 20

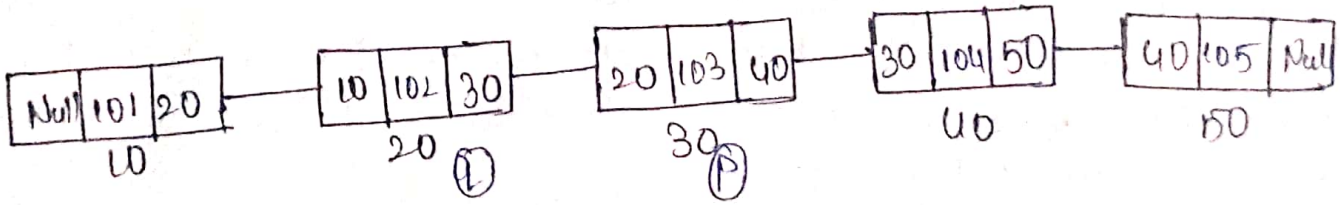
c++ ⇒ c = 2



c = 2

2 != 3 (True) ⇒ q = 20 and p = 30

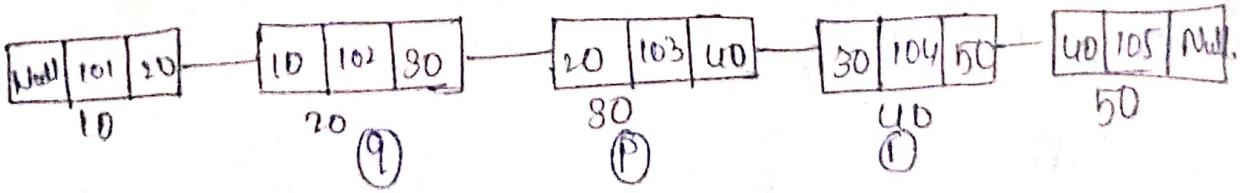
also c = 3



c = 3

3 != 3 (false) ⇒ q = 20 p = 30 and r = p->next

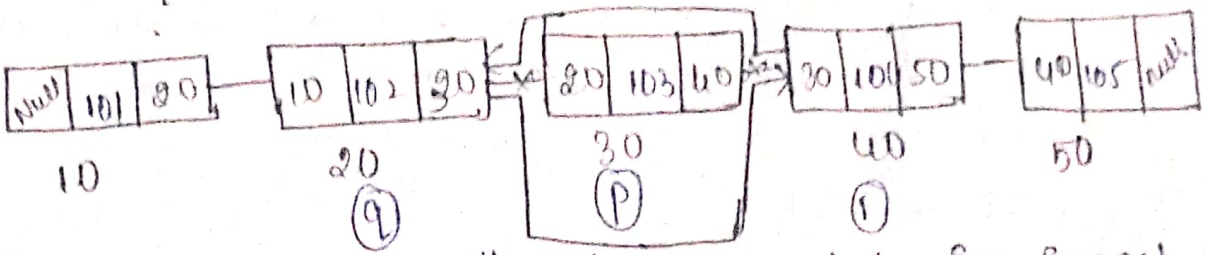
⇒ r = 40



Now as the link updations are as follows

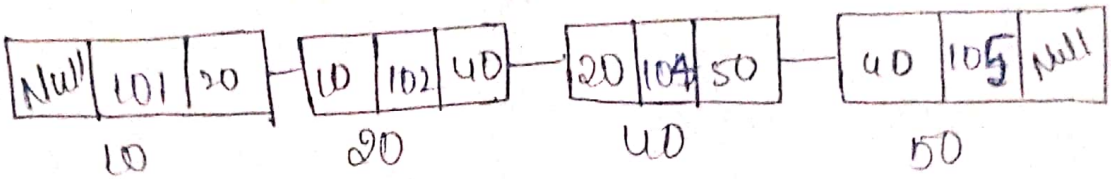
q->next = l and l->prev = q

⇒ q->next = 40 and l->prev = 20



then we can delete p^{th} n^{th} node i.e; 'p' using free(p)

And after deletion; list is as follows;



Note: special cases included in algorithm:

1. pos = 1 ⇒ node is head node. so we can use delete at head algorithm.

2. if pos = length(l) ⇒ node is the last since position is equal to length of the list. As the node is last node i.e tail of the list we can use delete at tail algorithm.

3. if pos < 1 or pos > l ⇒ Once pos < 1 then it is 0^{th} node and there is no 0^{th} node. Also if pos > l i.e if total nodes are 4 and if we want to delete 5^{th} or 6^{th} node then it is not possible because there exists no 5^{th} or 6^{th} nodes. So these two cases are failure cases.

Logic or algorithm to find length of the linked list:

```
p = head
l = 1
while (p->next != NULL)
{ l++;
  p = p->next;
}
```

4. Delete before a position:

Logic:

→ Goto $(n-1)^{th}$ node and called it as P.

→ $(n-2)^{th}$ node is called q and n^{th} node is called as r.

→ Once we identified the required nodes the following links/ updations has to be done;

i) $(n-2)^{th}$ → next to n

ii) n^{th} → prev to n-2

→ Then delete the $(n-1)^{th}$ node. i.e 'p' [free(p)].

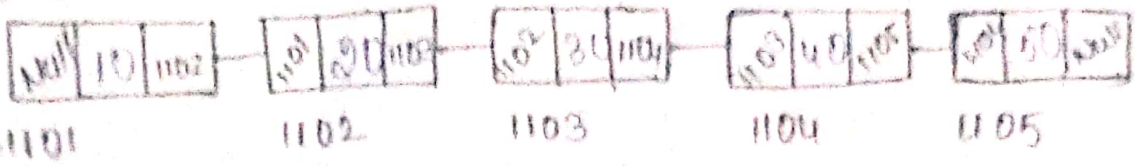
Algorithm:

Algorithm - delete before position,

1. Start
2. if head = NULL //checks for empty list.
   exit // No deletion takes places.
3. else if (pos = 2) // if pos=2, before is '1' so delete at head.
   delete at head. //function call.
4. else if pos = l+1 // if pos = l+1, it's before is 'l', so delete at tail
   delete at tail. //function call
5. else if pos < 2 or pos > l+1 //failure cases
   exit, deletion not possible.
6. else
   p = head //'p' is initilised to head
   c = 1 // count is stored in 'c' which is initialised to '1'
   while (c != pos-1) //Traversing to n^{th} position.
   {
   q = p;
   p = p->next
   c = c+1 //count goes up.
   }
   r = p->next // link updations
   q->next = r // q->next is made as r
   r->prev = q // r->previous is made q.
   free (p) // Memory deallocation.

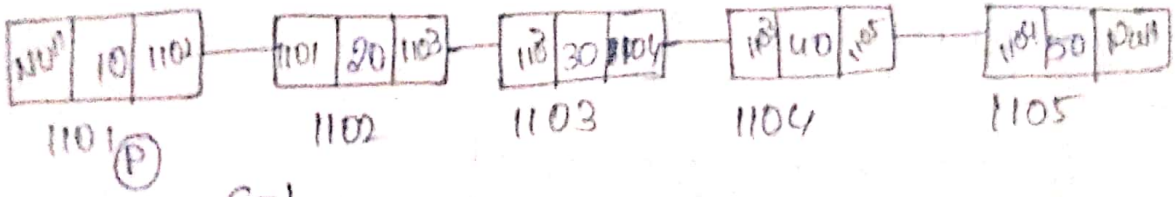
Tracing with an example:

Consider



Initially head = 1101. And also assume we want to delete a node before 5th position. we have to initially traverse to n^{th} (5^{th}) position i.e $n-1^{th}$ i.e 4^{th} node and $n-2^{th}$ i.e 3^{th} node

head = p ⇒ p = 1101 and c = 1

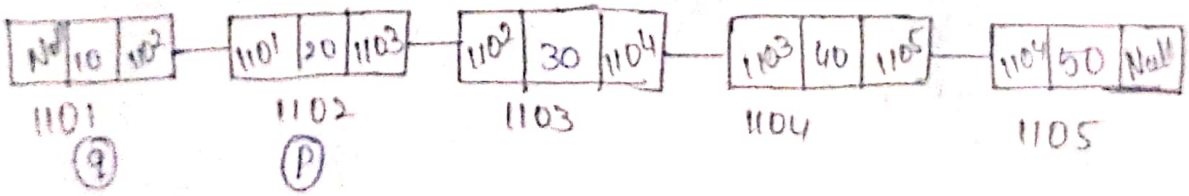


c = 1

$c != pos - 1 \Rightarrow c != 4 \Rightarrow 1 != 4$ True

⇒ q = p p = p→next c++ ⇒ c = 2

→ q = 1101 p = 1102

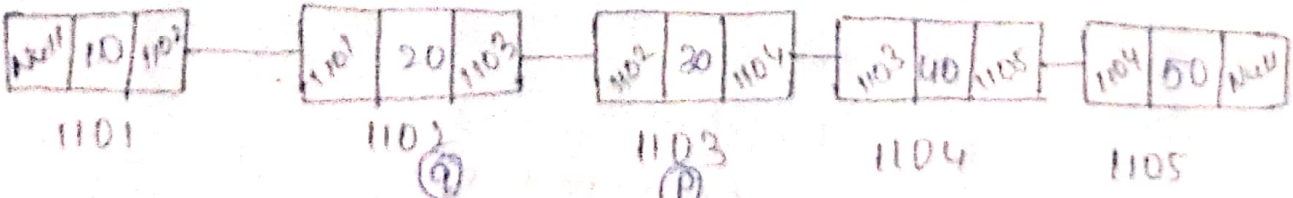


c = 2

$c != 4 \Rightarrow 2 != 4$ (True) ⇒ q = 1102

p = 1103

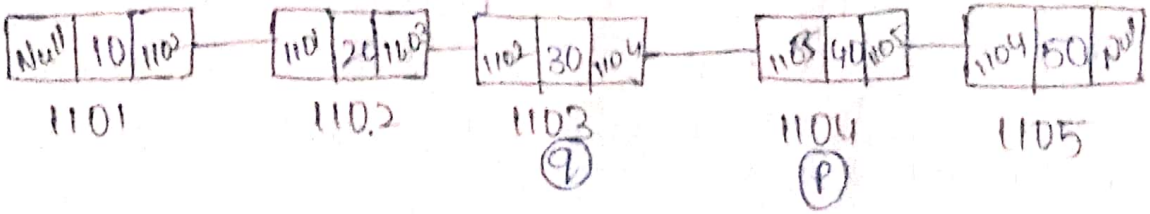
c = 3



c = 3

$3 != 4$ (True) ⇒ q = p ⇒ q = 1103

p = 1104 c = 4



c = 4

4 != 4 (False)

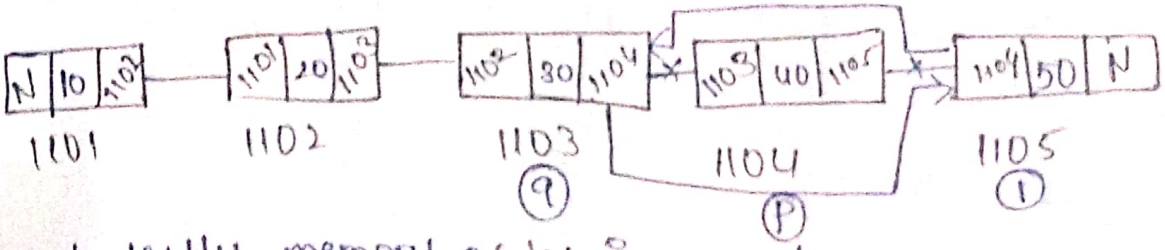
so execution stops

and q = 1103 p = 1104

t = p -> next => t = 1105

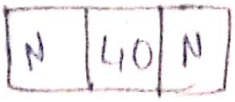
to remove ith node; q -> next = t => q -> next = 1105

t -> prev = q => t -> prev = 1103



and lastly memory of 'p' is freed.

updated linked list.



1104
deleted
node

Note: Special cases included in Algorithm:

1. if pos = 2; then its before position is '1' which is head node so in this case we can call delete at head algorithm.
2. if pos = l+1 then it's, before position would be 'l' which will be our last node. i.e 'l' is length of list and l^{th} node is the last node (tail). So we can use delete at tail algorithm
3. If pos < 2 or pos > l+1; In these cases, first considering pos < 2 then position stands from '1' and before first node we will have

no node. Also if pos > l+1 then, it's before node will be 'l+1' if given position is l+2 which is not present in the list as list consists of only 'l' nodes. So these two cases come under failure cases.

5. Deletion after a position:

Logic:

→ Go to $(n+1)^{th}$ node call it as p

→ n^{th} node as q

→ $(n+2)^{th}$ node as r

→ Make the updations as follows.

i) n^{th} node → next should be n+2

ii) $(n+2)^{th}$ node → prev should be n.

→ Delete the $(n+1)^{th}$ node.

Algorithm:

Algorithm - deleteafter position

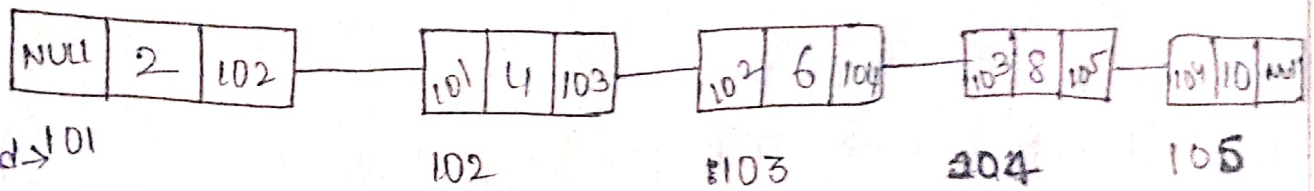
1. start
2. if head = NULL // checks for empty list
   exit() NO possible deletions // No deletions possible as list is empty
3. else if pos = 0 //
   delete at head
4. else if pos = l-1 // if position is l-1 then l is last one so delete at tail.
   delete at tail // function call.
5. else if pos < 0 or pos > l-1 // failure cases
   deletions not possible.
6. else
   p = head // p is initialised to head.
   c = 1 // count initialised to 1.
   while (c != pos+1) // Traversal to required nodes.
   {
   q = p
   p = p->next
   c++
   }
   r = p->next

q->next = r // q->next is made 'r'.
r->prev = q // r->prev is made 'q'.
free(p) // memory is freed.

7. stop.

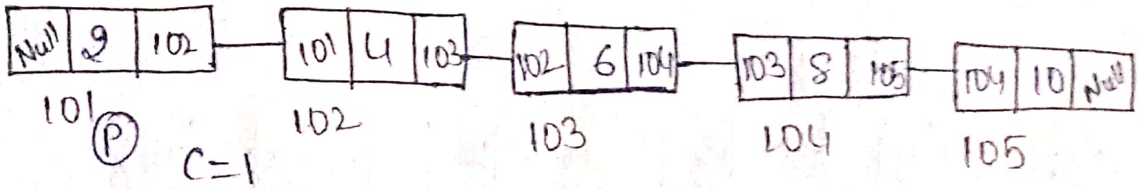
Tracing with an example:

Let the Linked list be;



Now let us assume that we want to delete a node after 3rd position. In this list head = 101.

Initially p=101 [∵ p=head] and c=1.



c=1

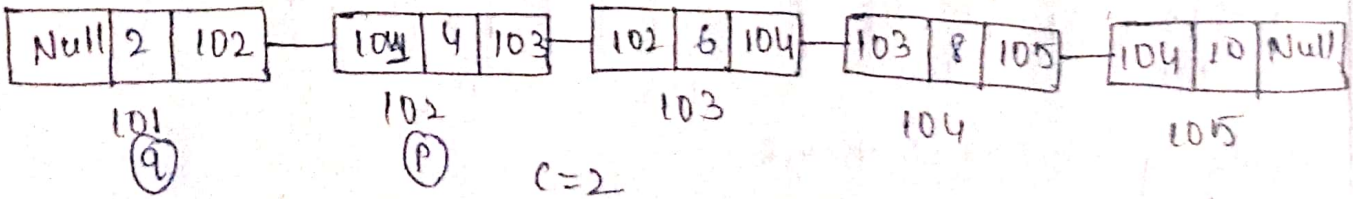
Checking the condition c!=pos+1

⇒ 1!=3+1 ⇒ 1!=4 (True)

so q=p ⇒ q=101

p=p->next ⇒ p=102

c=2

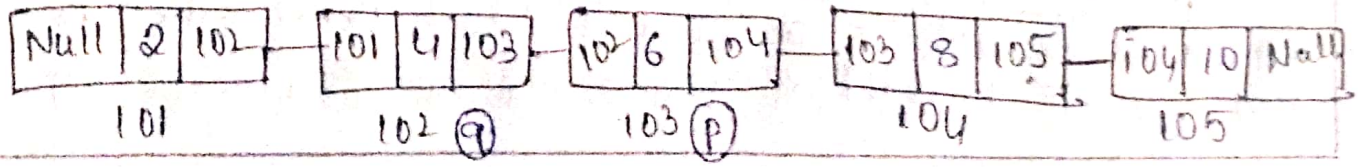


c=2

2!=4 True q=p ⇒ q=102 and

p=p->next ⇒ p=103.

c=3

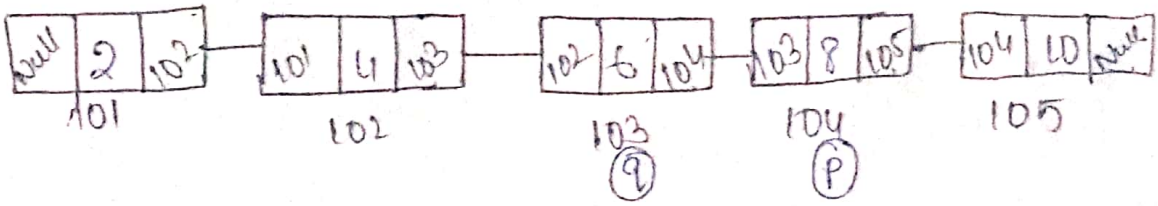


c = 3

c != pos+1 => 3 != 4 True

=> q = 103 and p = 104

c = 4.



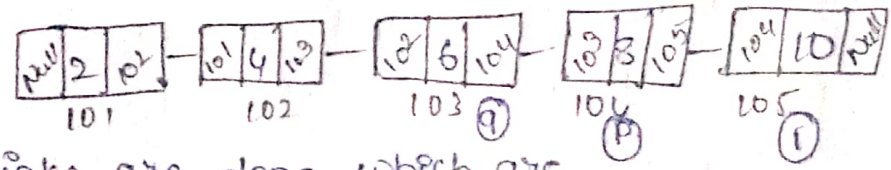
c = 4

c != pos+1 => 4 != 4 False.

→ comes out of loop and q = 103 p = 104 also

r = p → next

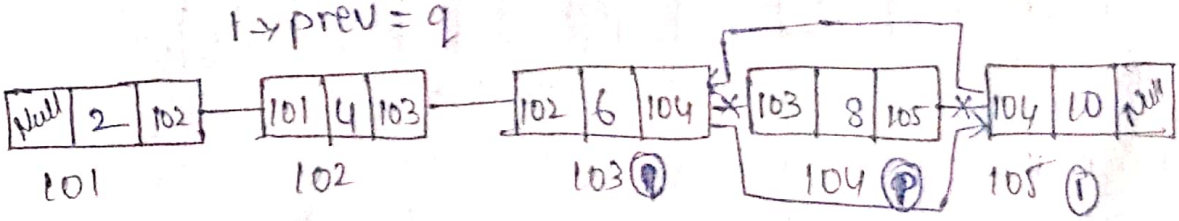
=> r = 105.



→ Updates for the links are done which are

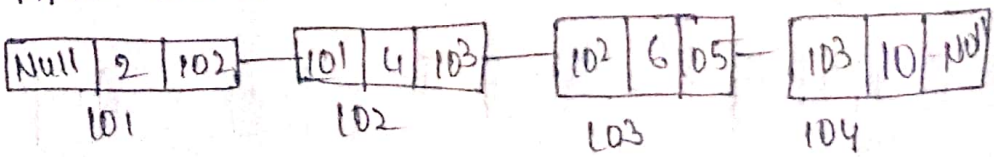
q → next = r

r → prev = q



→ after links are updated, we free the (p+1)th node. using free(p).

→ After the deletion, updated linked list is as follows:



Note: Special cases included:

1. If pos = l; the the node after it will be 1st which is the ~~first~~ last node so we can use delete at tail algorithm.

2. If pos < 1 or pos > l-1 the deletions will be failed because there will no node after 'l' or no node before '1'. Hence it will be not possible to delete if these positions are given. i.e. failure cases.

5. Delete with a value:

Logic:

→ First we traverse to the node contain that value and call it as p.

→ It's before node as q and it's after node as r.

→ Make link updates as follows.

i) q node next as r

ii) r node previous as q.

→ free the node with required value i.e free(p).

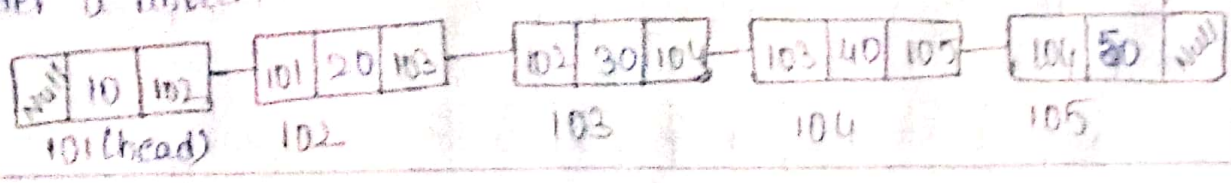
Algorithm:

Algorithm - delete with value

1. Start
2. If head = NULL // checks for empty list
   exit; deletion not possible // Deletion not possible
3. else if head -> data = value // if value in head node is required value delete at head algorithm used
   delete of head
   // function call
4. else
   p = head // 'p' is initialised to head
   while (p -> data != value) // Traversing list to find the node with required value.
   {
   q = p
   p = p -> next
   }
   r = p -> next // link updates
   q -> next = r // q -> next linked to r
   r -> prev = q // r -> prev linked to q.
   free (p) // freed memory space
5. Stop.

Tracing with an example:

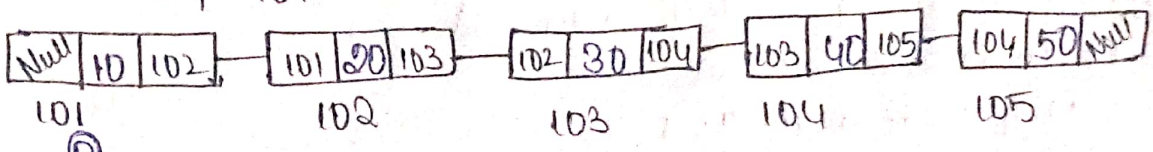
Consider a linked list



Suppose we want to delete a node with value 40.

Initially, p = head

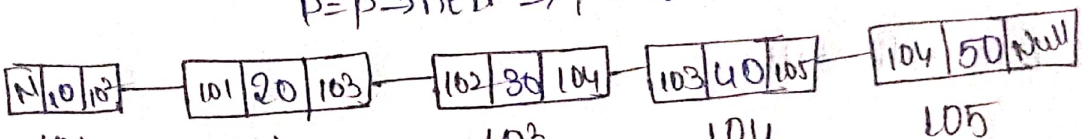
⇒ p = 101.



p → data != 40 ⇒ p 101 != 40 (True)

⇒ q = p ⇒ q = 101

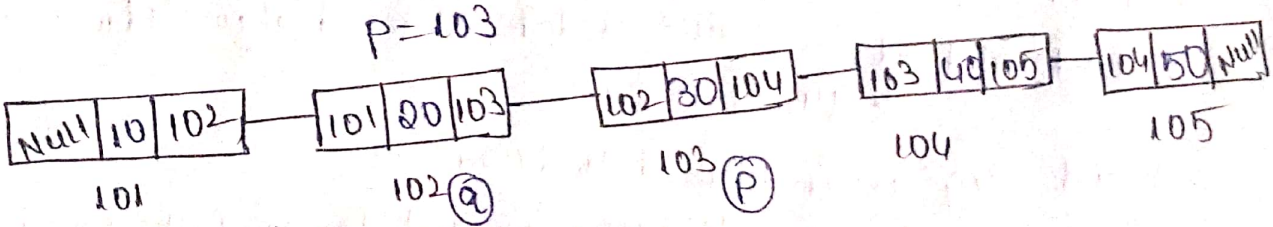
p = p → next ⇒ p = 102



p → data = 20; 20 != 40 (True)

⇒ q = 102

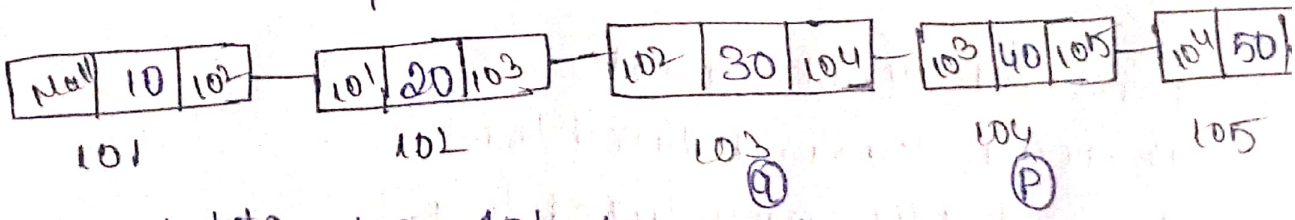
p = 103



p → data = 30 ; 30 != 40 (True)

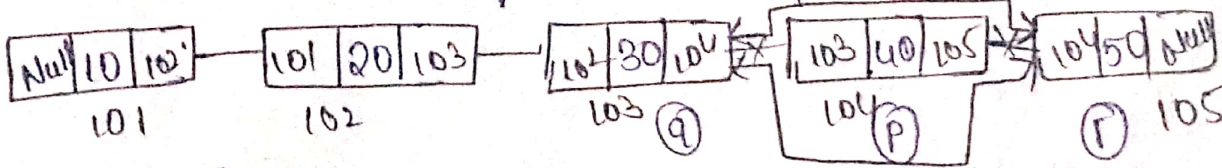
q = 103

p = 104.



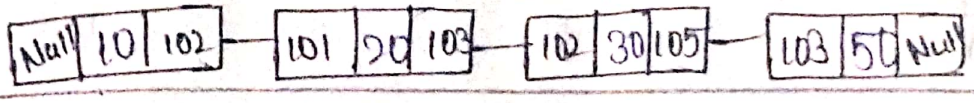
p → data = 40; 40 != 40 (False)

so loop stops executing. Now r = p → next = 105



q → next will be made 'r' and r previous should be made

and final list after deleting the node is;



//.

Special cases included in algorithm:

1. if head->data = value i.e value at head node is equal to the value we want to delete then we can use delete at head algorithm:

II. Delete at before a value:

Logic:

→ Go to (Traverse) to the node with the value and call it as 'p'.

→ It's previous node as 'q' (p->prev)

→ Also q->prev as 'r'.

→ Make the following link updations:

i) r->next = p

ii) p->prev as r.

→ Then free(q) node

Algorithm:

Algorithm - delete before value

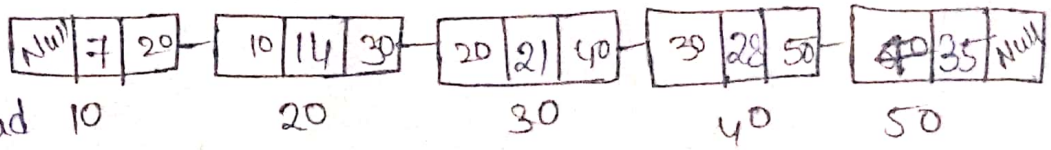
1. Start
2. if head = NULL // checks for empty list
   print; deletions not possible. // Deletion not possible if list is empty
3. else if head->next->data = value // and node value is the required value.
   delete at head
   // delete at head function is called
4. else
   p = head
   while (p->data != value) // Traversal to locate / find required nodes and stored in p, q, r. pointers
   q = p
   p = p->next
   r = q->prev
   r->next = p // r->next is made p

p->prev = r // p->prev is made r.

free(q) // memory deallocations.

5. Stop.

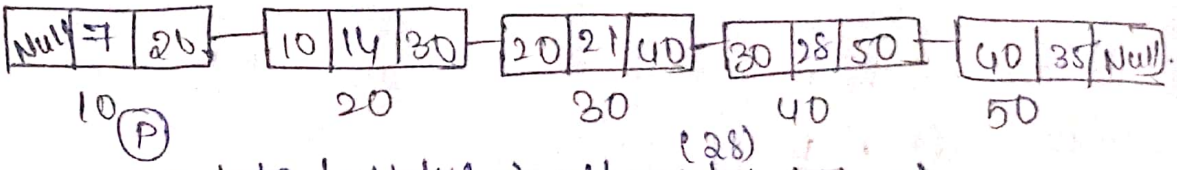
Tracing with an example:

Consider 

Suppose we want to delete a node at before value 28.

Initially; head = 10

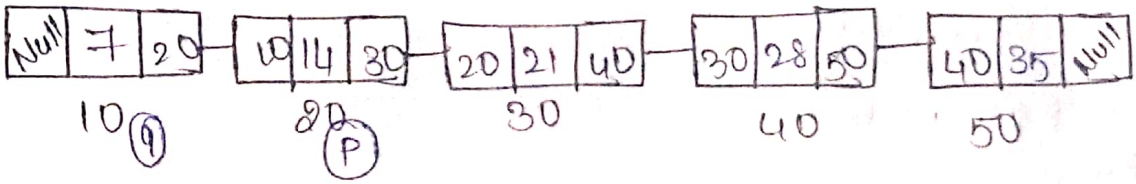
p = head ⇒ p = 10.



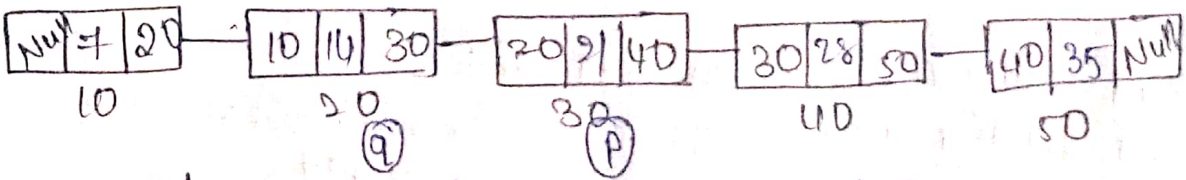
p->data != value ⇒ 7 != value (28) (True.)

⇒ q = p ⇒ q = 10

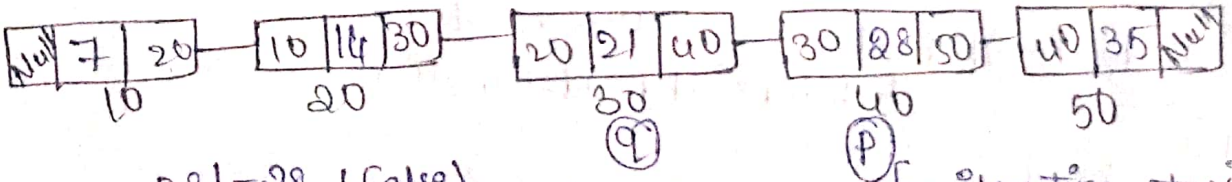
p = p->next ⇒ p = 20.



14 != 28 (True) q = 20 p = 30



21 != 28 (True) q = 30 p = 40



28 != 28 (false) ⇒ q = 30 p = 40 [∵ iteration stops]

Now r = q->prev ⇒ r = 20