Deletions in singly Linked List:-

1. Deletion at Head position:-

In this operation, the first node of the linked list is removed. In a singly linked list, the head always points to the first node.

since the head pointer points to the first node, we simply move the head pointer to the next node.

Logic:

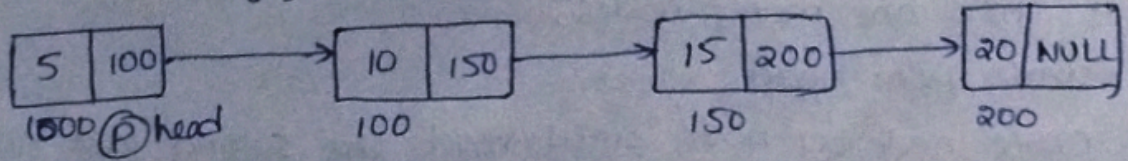
1. check if the linked list is empty (head == NULL)
2. store the first node in temp pointer (temp = head)
3. move head pointer to the next node (head = head → next)
4. Delete first node using free(temp)
5. then the second node becomes new head.

Algorithm: delete_at_head

1. start
2. if (head == NULL) // check whether the list is empty.
   exit. Deletion is not possible // As list is empty, deletion is not possible.
3. else
   p = head // 'p' is initialised to head
   head = head → next // head is moved to the next node
4. free (p) // memory freed
5. stop.

Tracing with an example:-

consider a singly linked list

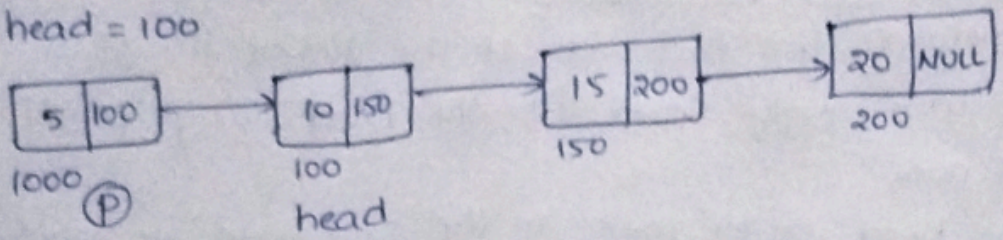


Initially p = 1000 = head

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

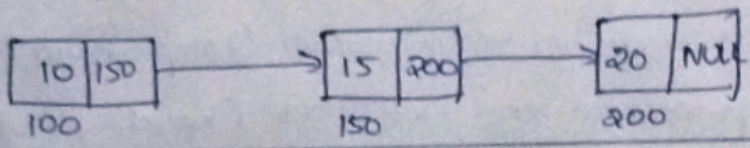
⇒ head = head → next

⇒ head = 100



In the next step, we should change free the memory of previous head ⇒ free(p)

After deleting a node at head, the list is



2. Deletion at tail position:

In this operation, the last node of the linked list is removed.

In a singly linked list, the last node always contains:

next = NULL

To delete last node, we must find the second last node and then make its next pointer = NULL.

Logic:

1. check if the list is empty
   - if head = NULL, the list is empty → Nothing to delete
2. check if there is only one node:
   - if head → next = NULL, delete that node and make head = NULL
3. If there are many nodes:
   - starts from head
   - move node by node until reach the second last node
   - make the second last node's next = NULL
   - Delete the last node.

Algorithm: delete_at_tail

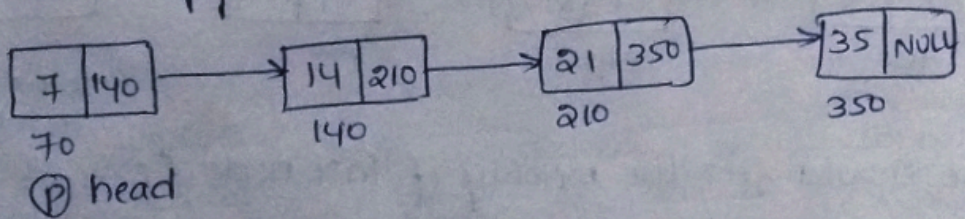
1. start
2. if head == NULL // check whether the list is empty
   exit. deletion is not possible // deletion not possible as there are no nodes.
3. else

```
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
 }
   q→next = NULL                // link updates
```

4. Free(p) // memory freed
5. stop.

Tracing with an example:

consider a singly linked list,



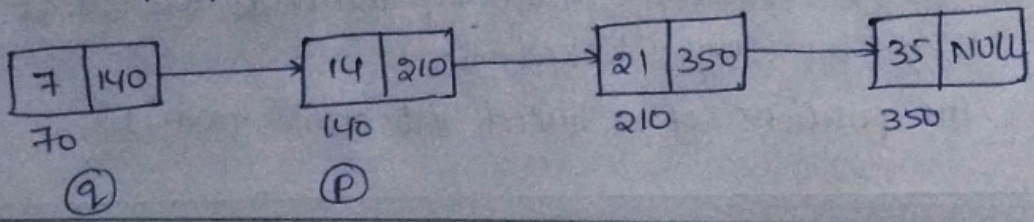
initially p=head

p → next = 140 ; p → next != NULL

140 != NULL (True)

then q=p ⇒ q=140 70

p = p → next ⇒ p=140

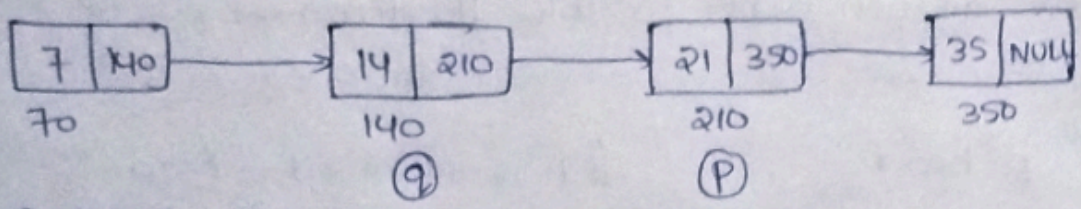


p → next = 210 ; p → next != NULL

210 != NULL (TRUE)

then q = p ⇒ q = 140

p = p → next ⇒ p = 210

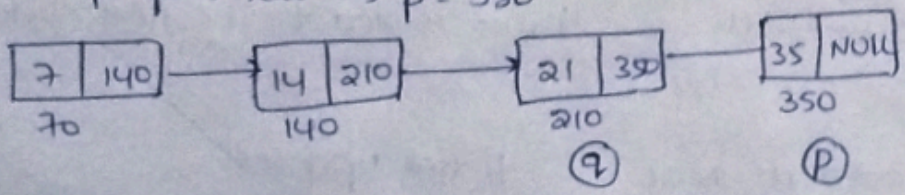


p → next = 350 ; p → next != NULL

350 != NULL (TRUE)

then q = p ⇒ q = 210

p = p → next ⇒ p = 350

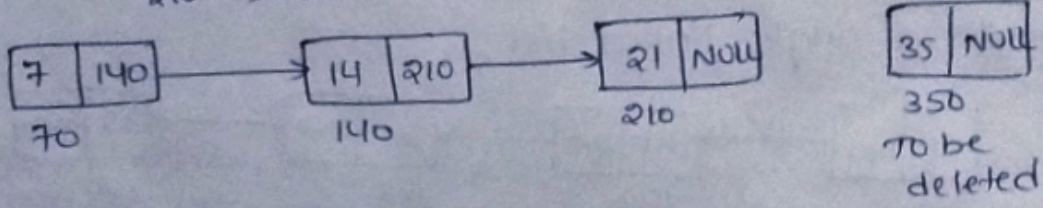


p → next = NULL ; p → next != NULL

NULL != NULL (False)

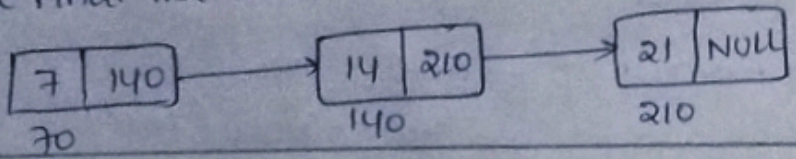
then q → next = NULL

210 → next = NULL



We should free the memory of last node. (free(p))

The final list is



3. Deletion at nth position:

In this operation, Removing the node located at position n in the singly linked list.

The positions of a linked list starts from "1"

Logic:

1. Check if the list is empty (head == NULL)
   - if empty ⟶ nothing to delete
2. If N = 1
   - Delete the first node (head)
   - move head to the next node
3. otherwise
   - starts from the head
   - move the pointer until reach the $(N-1)^{th}$ node
4. The Nth node is the node to be deleted.
5. connect $(N-1)^{th}$ node to $(N+1)^{th}$ node.
6. Delete the n^{th} node using free( ).

Algorithm: delete_at N^{th}_pos

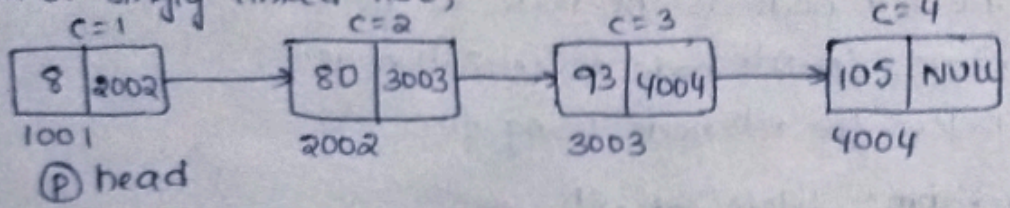
1. start
2. If head == NULL // checks whether the list is empty
   exit. deletion is not possible // if list is empty, no deletion possible.
3. else if pos = 1 // we have to delete at pos "1", delete at head.
   delete_at_head // function call
4. else if pos = l // we have to delete last position
   delete_at_tail
5. else if pos < 1 or pos > l // failure case
   exit the code
6. else
   c = 1 // count is stored which is initialised to "1"
   p = head // p is initialised to head
   while (c != pos) // traverse to required position
   {
   q = p // q is stores the value of p

```
    p = p→next
    c++                          // count goes up
}
    r = p→next                   // To delete the required node
    q→next = r                   // update links
```

7. free(p) // memory freed
8. stop

Tracing with an example:-

consider a singly linked list;



Initially p=head = 1001

Now I want to delete the node at 3rd position (N=3)

p = head = 1001

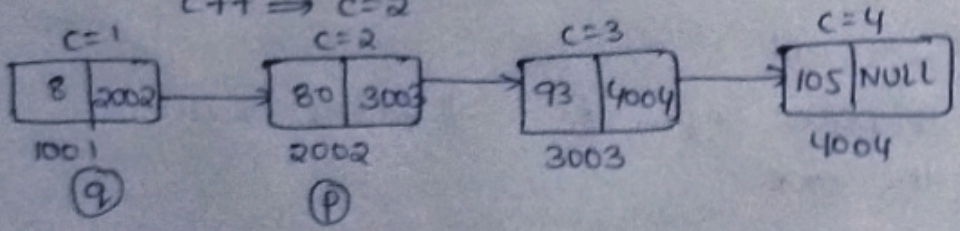
c = 1

(c != pos) = (1 != 3) → (True)

then q = p → q = 1001

p = p→next ⇒ p = 2002

c++ ⇒ c = 2



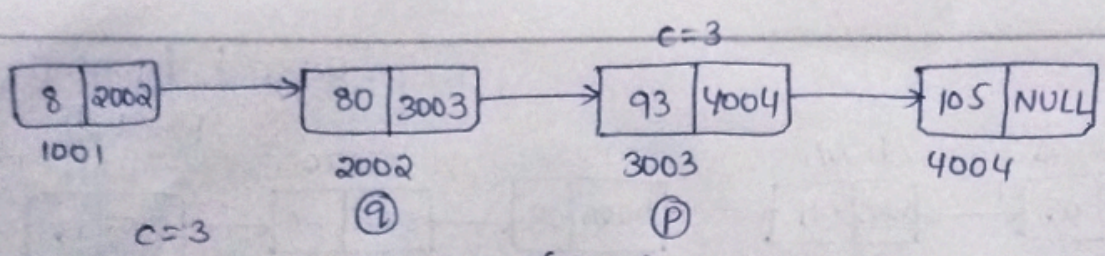
c = 2

(c != pos) = (2 != 3) → (True)

then q = p ⇒ q = 2002

p = p→next ⇒ p = 3003

c++ ⇒ c = 3



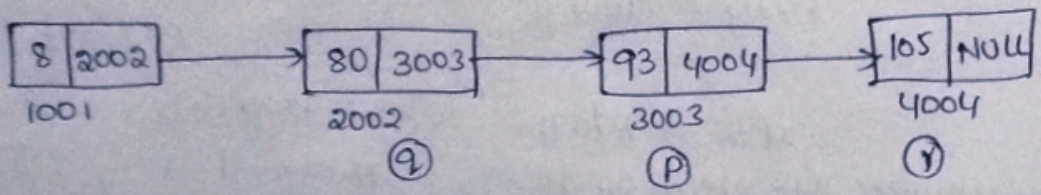
c=3

(c != pos) = (3 != 3) → (False)

So, r = p → next ⇒ r = 4004

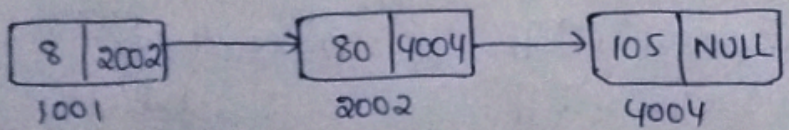
q → next = r ⇒ q → next = 4004

⇒ 2002 → next = 4004



Now we should free the memory of 'p' node ⇒ free(p)

The Final list is



4. Deletion at Before Nth position:

In this operation, Removing the node that comes immediately before Nth node in the singly linked list.

Logic:

1. check if the list is empty
   - if head = NULL, nothing to delete.
2. If N ≤ 2
   - There is no node before Nth position, deletion is not possible.
3. otherwise
   - starts from the head
   - move the pointer until reach (N-2)th position.
   - The (N-1)th node is the node to be deleted.
4. change the links

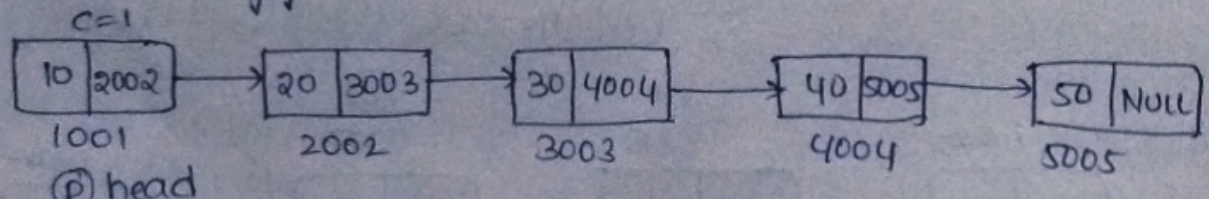
   temp → next = temp → next → next
5. Free the deleted node.

Algorithm: delete_at before N^{th}_pos

1. start
2. if head == NULL — // checks whether the list is empty
   exit. deletion is not possible — // if the list is empty, deletion is not possible.
3. else if (pos == 2) — // if (pos == 2), before is pos = 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 the code
6. else
   p = head — // p is initialised to head
   c = 1 — // count stored, initialises to "1"
   while (c != pos-1) — // traverse to required position
   {
   q = p; — // q is stores the value of p
   p = p → next; — // p is moved to the next node
   c++; — // count goes up
   }
   r = p → next; — // to delete the required node
   q → next = r; — // update links
7. free (p) — // memory freed
8. stop.

Tracing with an example :-

consider a singly linked list,



Initially p = head = 1001

Now I want to delete node at before 4th position. i.e, (pos=3)

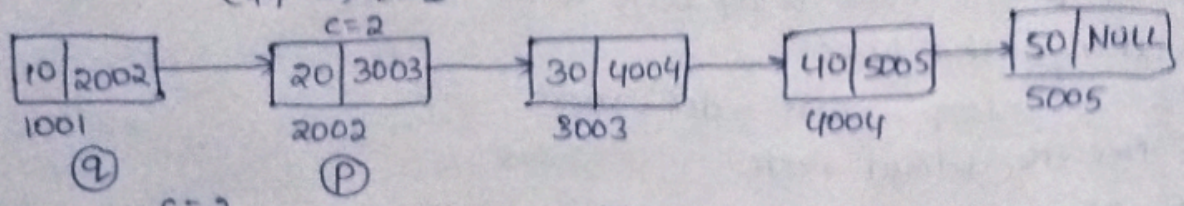
c=1

(c!=pos-1) = (1!=3) → (true)

then q=p ⇒ 1001

p=p→next ⇒ 2002

c++ ⇒ c=2



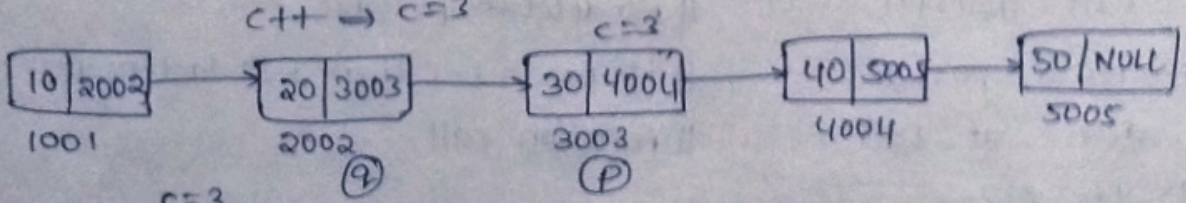
c=2

(c!=pos-1) = (2!=3) → (true)

then q=p ⇒ q=2002

p=p→next ⇒ p=3003

c++ ⇒ c=3

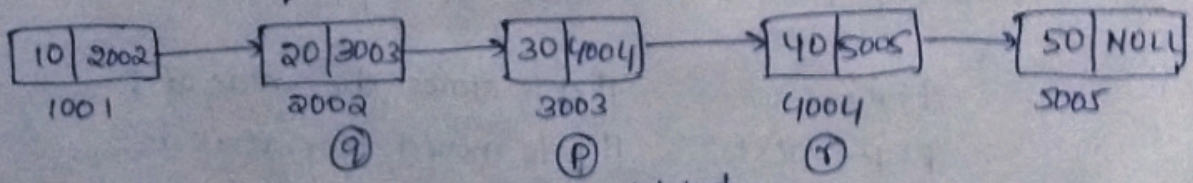


c=3

(c!=pos-1) = (3!=3) → (False)

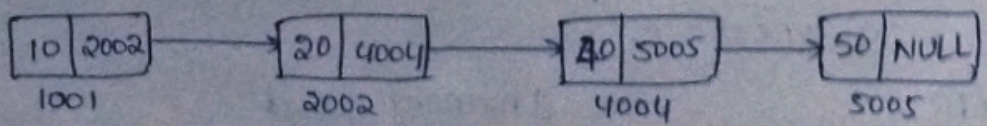
so, r=p→next ⇒ r=4004

q→next=r ⇒ 2002 → next = 4004



Now we should free the memory of p ⇒ free(p)

Then the final list after updating the links is:



5. Deletion at After Nth position:

In this operation, Removing the node that comes immediately after Nth node in the singly linked list.

Logic:

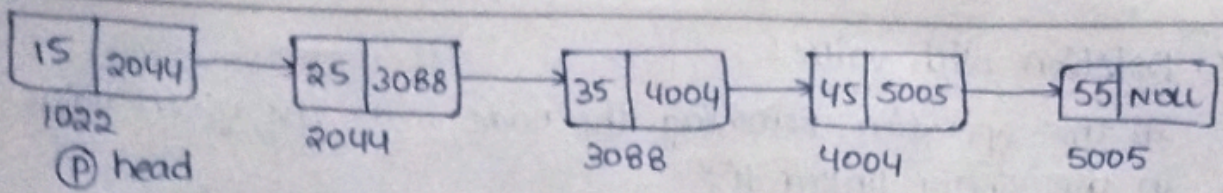
1. check if the list is empty
   - if head == NULL, nothing to delete
2. start from the head node
3. move the pointer until the Nth node
4. The (N+1)th node is the node to be deleted.
5. change the links
   temp → next = del → next
6. Free the deleted node.

Algorithm: delete_at after Nth_pos

```
1. start
2. if head == NULL          // checks whether the list is empty
   exit. deletion           // No deletion possible
3. else if pos = l-1        // if pos is l-1 then l is last node
   delete_at_tail           // function call
4. else if pos
       p = head             // p is initialised to head
       c = 1                // count stored, initialised to "1"
       while (c != pos+1)   // traverse to required position
         {
           q = p;           // q is stores the value of p
           p = p → next;    // p is moved to next node
           c++;             // count goes up
         }
       r = p → next;        // To delete require node,
                               update links
       q → next = r;
5. Free(p)                  // memory freed
6. stop
```

Tracing with an example:

consider a singly linked list,



Initially p = head = 1022

Now I want to delete the node at after 2nd position. i.e, (pos = 3)

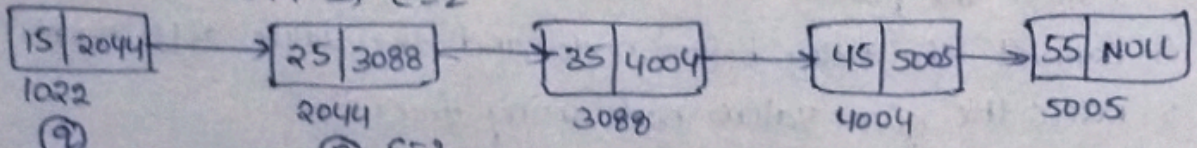
c = 1

(c != pos+1) = (1 != 3) → (True)

then q = p ⇒ q = 1022

p = p → next ⇒ p = 2044

c++ ⇒ c = 2



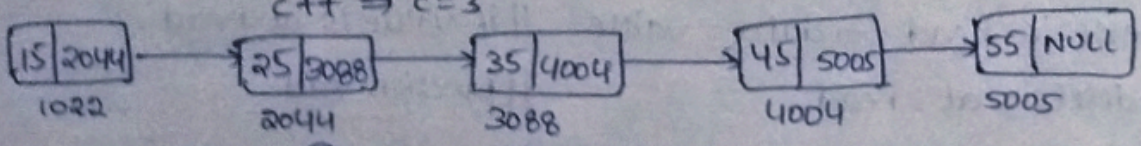
c = 2

(c != pos+1) = (2 != 3) → (True)

then q = p ⇒ q = ~~2044~~

p = p → next ⇒ p = ~~3088~~

c++ ⇒ c = 3



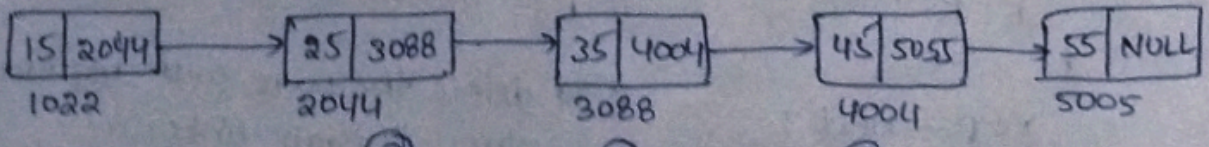
c = 3

(c != pos+1) = (3 != 3) → (False)

then ~~q = p ⇒ q = 3088~~

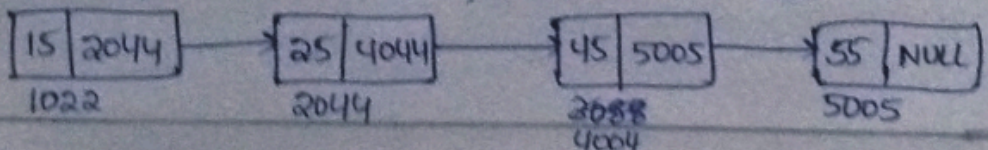
r = p → next ⇒ r = 4004

q → next = r ⇒ 2044 → next = 4004



Now we should free the memory of 'p' → free(p)

After update the links, the final linked list is:



6. Deletion with value:

In this operation, Removing the node with the given value in the singly linked list.

Logic:

1. check if the linked list is empty
2. If the given value is in first node (head), move the head to the next node.
3. otherwise
   - starts from head
   - move the pointer until reach the given value
4. connect the value before node and the value after node.
5. delete the given value node using free( ).

Algorithm : delete_with_value

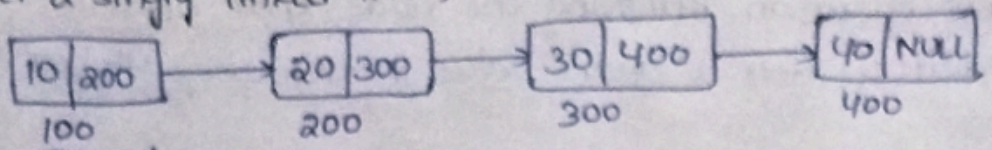
1. start
2. If head == NULL // checks whether the list is empty
   exit. deletion not possible // deletion not possible
3. else if head → data = value // if value is at head
   delete_at_head // function call
4. else

```
p=head                          // p is initialised to head
while (p→data != value)         // traversing the list to find the
{                                  required node
  q=p;                          // q is stores the value of p
  p=p→next;                     // p is moved to next node
}
  r=p→next;                     // To delete the node with required
  q→next=r;                        value. //update links
```

5. free(p) // memory freed
6. stop

Tracing with an example :-

consider a singly linked list ;



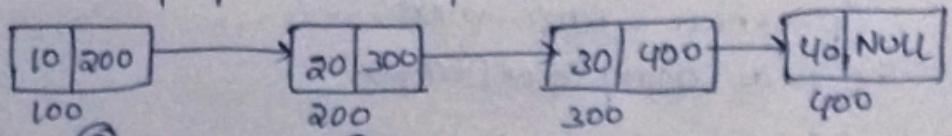
Now I want to delete the node with value 30

Initially p = head = 100

(p → data != value) = (10 != 30) → (True)

then q = p ⇒ q = 100

p = p → next ⇒ p = 200

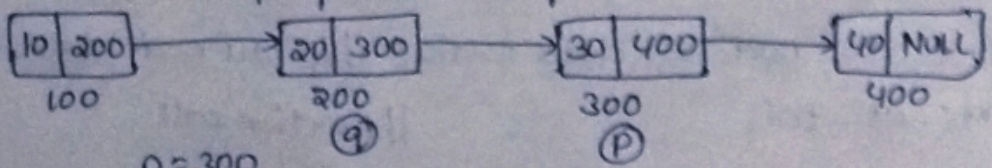


p = 200

(p → data != value) = (20 != 30) → (True)

then q = p ⇒ q = 200

p = p → next ⇒ p = 300

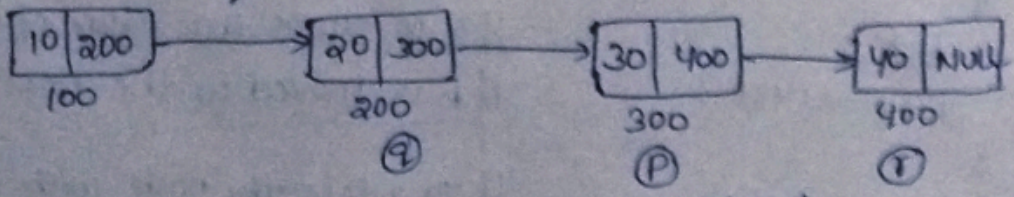


p = 300

(p → data != value) = (30 != 30) → (False)

So, r = p → next ⇒ r = 400

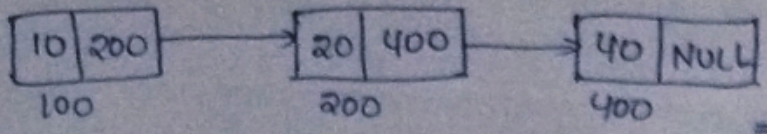
q → next = r ⇒ 200 → next = 400



To be deleted

Now we should free the memory of 'p' → free(p)

After updating the links, the final linked list is :



7. Deletion After a value:

In this operation, Removing the node that appears immediately after a given value in the singly linked list.

Logic:

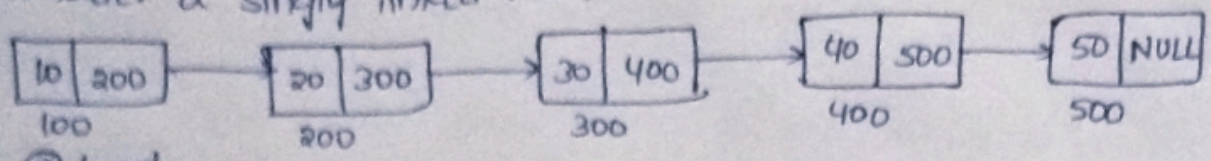
1. check if the linked list is empty
2. Traverse the list to find the node with given value
3. check if there is a node after that value.
4. store the next node in temp pointer. (temp=p->next)
5. link the current node to the next of that node
   (p->next = temp->next)
6. Delete the node using free(temp).

Algorithm: delete_after_value:

1. Start
2. if head == NULL // checks whether the list is empty
   exit. deletion is not possible // No deletion occurs
3. else if last node -> data = value // if last node data = value
   delete_at_tail // function call
4. else
   p = head // p initialised to head
   while (p->data != value) // traverse to find the required node
   {
   ~~q=P~~ // ~~q is stores the value of P~~
   p = p->next; // p is moved to the next node
   }
   q = p->next; // to delete the node with required value
   q->next = r;
   p->next = r;
5. Free(q) // memory freed
6. Stop

Tracing with an example:

consider a singly linked list;



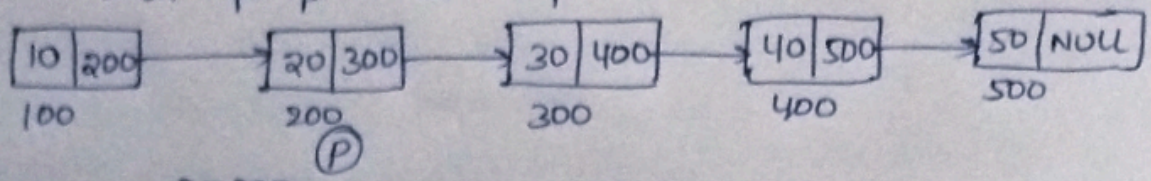
Initially p=head=100

Now we want to delete the node after the value (30)

p=100

(p→data != value) = (10 != 30) (TRUE)

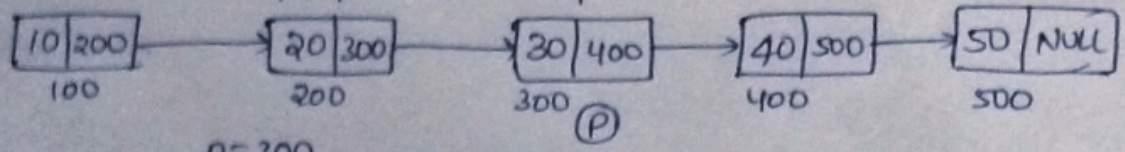
then p=p→next ⇒ p=200



p=200

(p→data != value) = (20 != 30) (TRUE)

then p=p→next ⇒ p=300



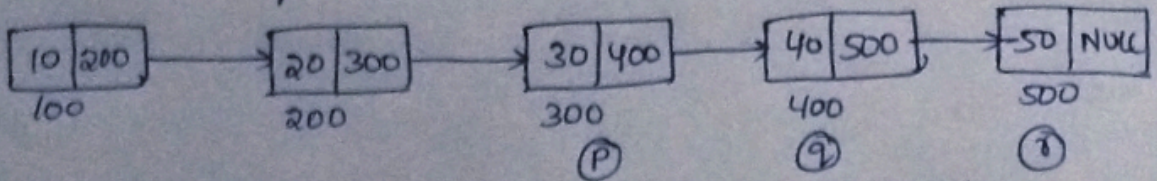
p=300

(p→data != value) = (30 != 30) (False)

so, q=p→next ⇒ q=400

q→next = r ⇒ 400→next = r

p→next = r ⇒ 300→next = r



Now we should free the memory of 'q' ⇒ free(q)

update the links, then final linked list is;