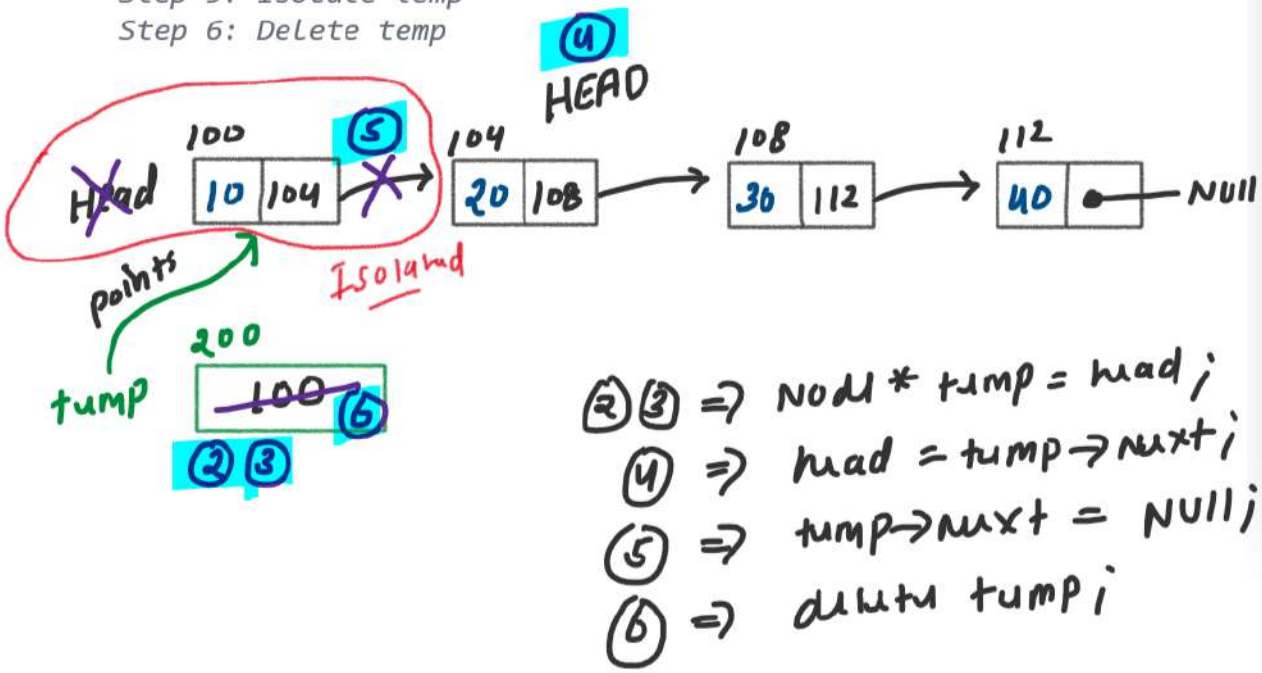
4/11/2023

LINKED LIST CLASS - 2

1. Deletion Operations

1. Delete a node from the head

Step 1: Set Corner Cases: Empty LL and Single Element LL
Step 2: Create a temp node
Step 3: Attached temp node with head
Step 4: Update head
Step 5: Isolate temp
Step 6: Delete temp



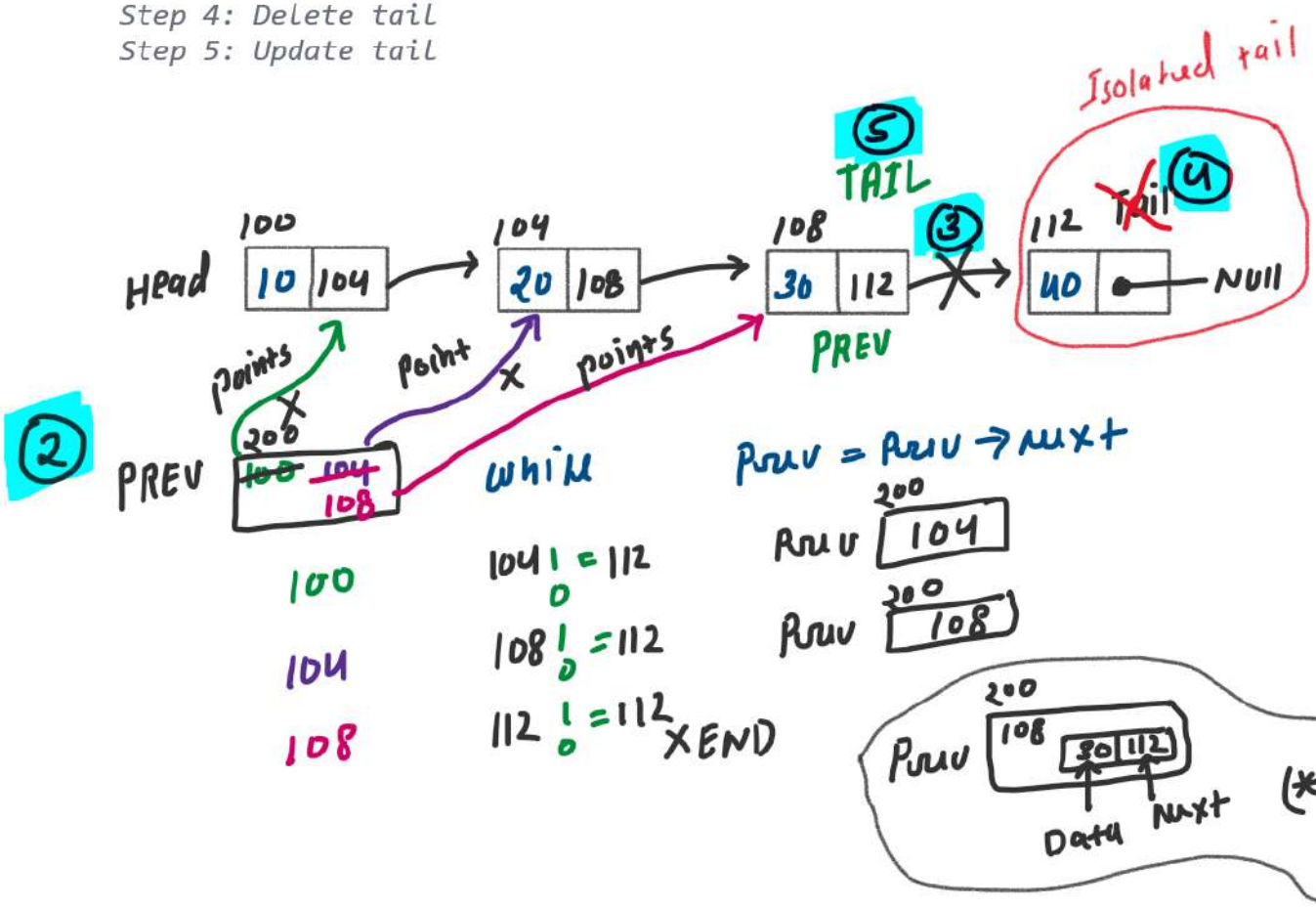
(2)(3) ⇒ Node* temp = head;
(4) ⇒ head = temp->next;
(5) ⇒ temp->next = NULL;
(6) ⇒ delete temp;

```cpp
// Delete a node from the head
void deletefromHead(Node* &head, Node* &tail)
{
    // Step 1: Set Corner Cases: Empty LL and Single Element LL
    if(head == NULL){
        // Empty LL
        cout << "Can't delete, because of empty linked list" << endl;
        return;
    }
    if(head == tail){
        // Single Element LL
        Node* temp = head;
        delete temp;
        head->next = NULL;
        tail->next = NULL;
    }

    // Step 2 and 3: Create a temp node and Attached temp node with head
        Node* temp = head;
    // Step 4: Update head
        head = temp->next;
    // Step 5: Isolate temp
        temp->next = NULL;
    // Step 6: Delete temp
        delete temp;
}
```

Output ⇒ 20 30 40

📂 2. Delete a node from the tail

Step 1: Set Corner Cases: Empty LL and Single Element LL
Step 2: Traverse second last node (PREV)
Step 3: Isolate PREV
Step 4: Delete tail
Step 5: Update tail




```cpp
// 📂 2. Delete a node from the tail
void deleteFromTail(Node* head, Node* tail){
    // Step 1: Set Corner Cases: Empty LL and Single Element LL
    if(head == NULL){
        // Empty LL
        cout << "Can't delete, because of empty linked list" << endl;
        return;
    }
    if(head == tail){
        // Single Element LL
        Node* temp = head;
        delete temp;
        head->next = NULL;
        tail->next = NULL;
    }

    // Step 2: Traverse second last node (PREV)
    Node* prev = head;
    while (prev->next != tail)
    {
        prev = prev->next;
    }

    // Step 3: Isolate PREV
    prev->next = NULL;

    // Step 4: Delete tail
    delete tail;

    // Step 5: Update tail
    tail = prev;
}
```

Out ⇒ 10 20 30

📂 *3. Delete a node from any position*

Step 1: Set Corner Cases

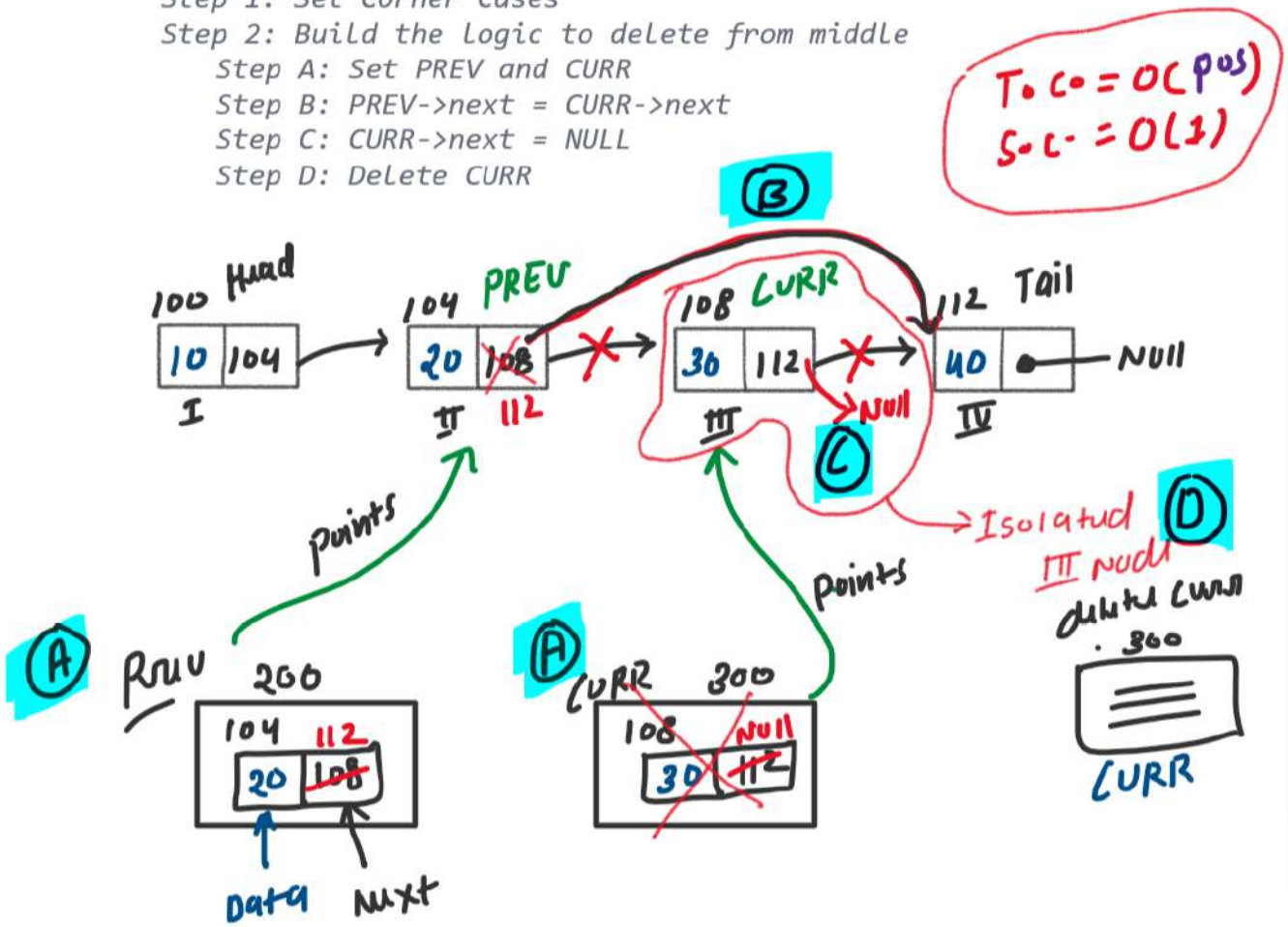
Step 2: Build the logic to delete from middle

Step A: Set PREV and CURR

Step B: PREV->next = CURR->next

Step C: CURR->next = NULL

Step D: Delete CURR



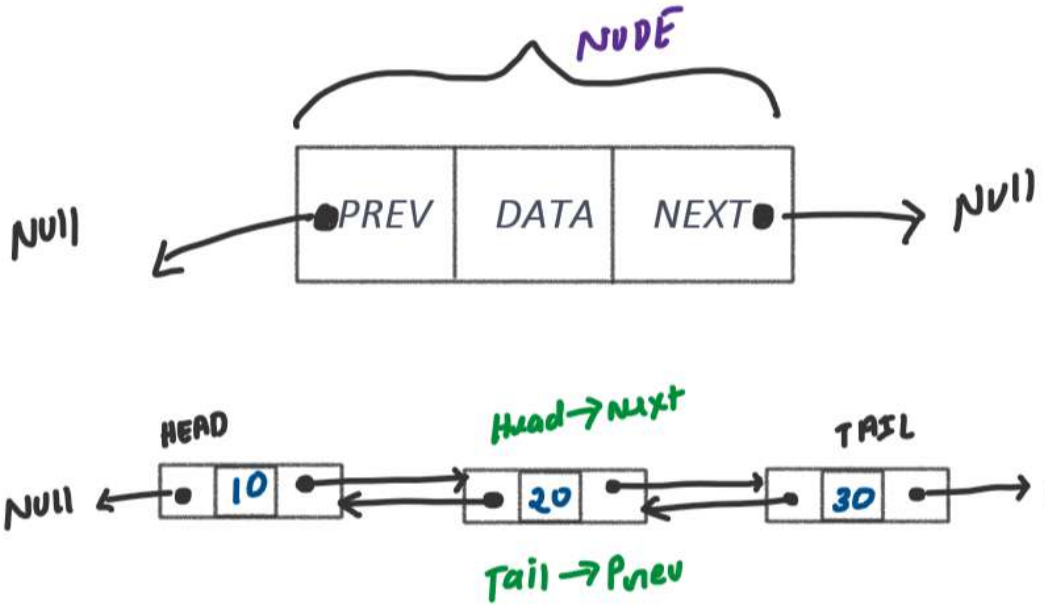

```cpp
// 📂 3. Delete a node from any position
void deleteFromPosition(Node* &head, Node* &tail, int position){
    // Step 1: Set Corner Cases: Empty LL and Single Element LL
    if(head == NULL){...}
    if(head == tail){...}

    // Delete node from head
    if(position == 1){
        deletefromHead(head, tail);
    }
    // Delete node from tail
    else if (position == getLength(head)){
        deleteFromTail(head, tail);
    }
    // Step 2: Build the logic to delete from middle
    else{
        // Step A: Set PREV and CURR
        Node* prev = NULL;
        Node* curr = head;
        while (position != 1)
        {
            prev = curr;
            curr = curr->next;
            position--;
        }

        // Step B: PREV->next = CURR->next
        prev->next = curr->next;

        // Step C: CURR->next = NULL
        curr->next = NULL;

        // Step D: Delete CURR
        delete curr;
    }
}
```

We want to delete node at pos III

OutP ⇒ 10 20 40

2. Double linked list

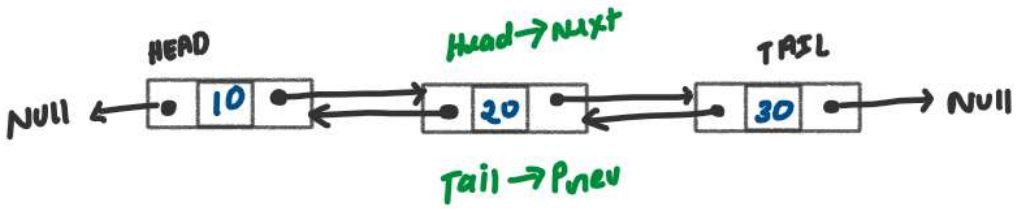
```cpp
// Double Linked List Node

class Node
{
    public:
        int data;
        Node* prev;
        Node* next;

        Node(){
            this->prev = NULL;
            this->next = NULL;
        }

        Node(int data){
            this->data = data;
            this->prev = NULL;
            this->next = NULL;
        }
};
```



📂 **Double linked list program**
- Print linked list
- Find length of linked list

✅ Insertion operations of DLL
- Insert node at the head
- Insert node at the tail
- Insert at any position

❌ Deletion operations of DLL
- Delete a node from the head
- Delete a node from the tail
- Delete a node from any position



Length = 3 and **Print DLL** = 10->20->30->

Both are same function as single linked list

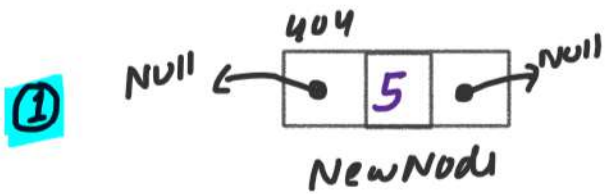
```cpp
// Print linked list
void printDLL(Node* &head)
{
    Node* temp = head;

    while(temp != NULL){
        cout << temp->data << "->";
        temp = temp->next;
    }
    cout<<endl;
}
```

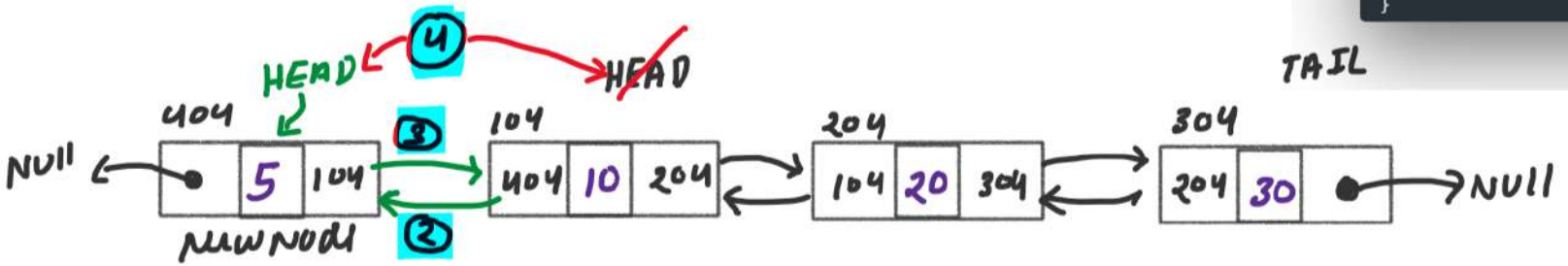
```cpp
// Find length of linked list
int findLength(Node* &head)
{
    Node* temp = head;
    int count = 0;

    while(temp != NULL){
        count++;
        temp = temp->next;
    }
    return count;
}
```

✅ Insertion operations of DLL

📂 1. Insert node at the head

Step 1: Create a new node
Step 2: Head->PREV = new node
Step 3: new node->NEXT = Head
Step 4: Head = new node

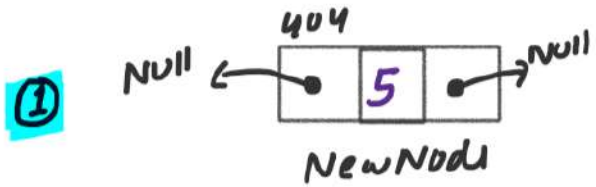


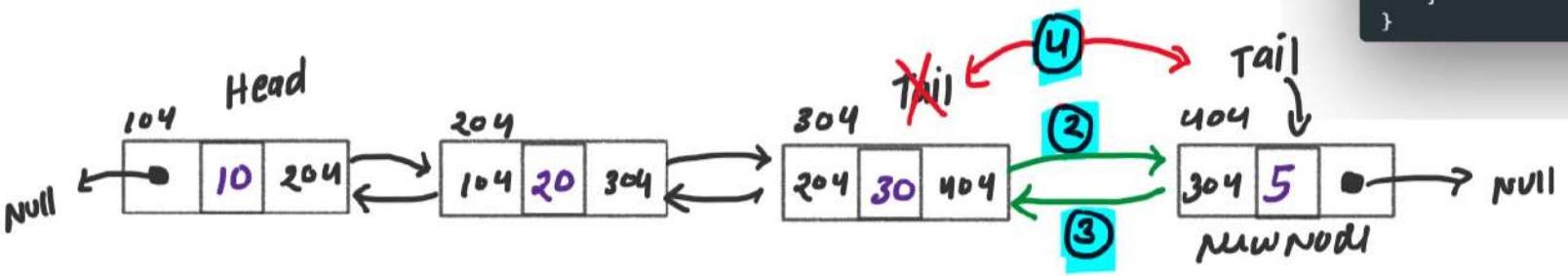



```cpp
// 📂 1. Insert node at the head
void insertAtHead(Node* &head, Node* &tail, int data)
{
    // Corner case 1: Empty DLL
    if(head == NULL){
        Node* newNode = new Node(data);
        head = newNode;
        tail = newNode;
    }
    // Non Empty DLL
    else{
        // Step 1: Create a new node
        Node* newNode = new Node(data);

        // Step 2: Head->PREV = new node
        head->prev = newNode;

        // Step 3: new node->NEXT = Head
        newNode->next = head;

        // Step 4: Head = new node
        head = newNode;
    }
}
```

📂 2. Insert node at the tail
Step 1: Create a new node
Step 2: Tail->NEXT = new node
Step 3: new node->PREV = Tail
Step 4: Tail = new node

```cpp
// 📂 2. Insert node at the tail
void insertAtTail(Node* &head, Node* &tail, int data)
{
    // Corner case 1: Empty DLL
    if(head == NULL){
        Node* newNode = new Node(data);
        head = newNode;
        tail = newNode;
    }
    // Non Empty DLL
    else{
        // Step 1: Create a new node
        Node* newNode = new Node(data);

        // Step 2: Tail->NEXT = new node
        tail->next = newNode;

        // Step 3: new node->PREV = Tail
        newNode->prev = tail;

        // Step 4: Tail = new node
        tail = newNode;
    }
}
```

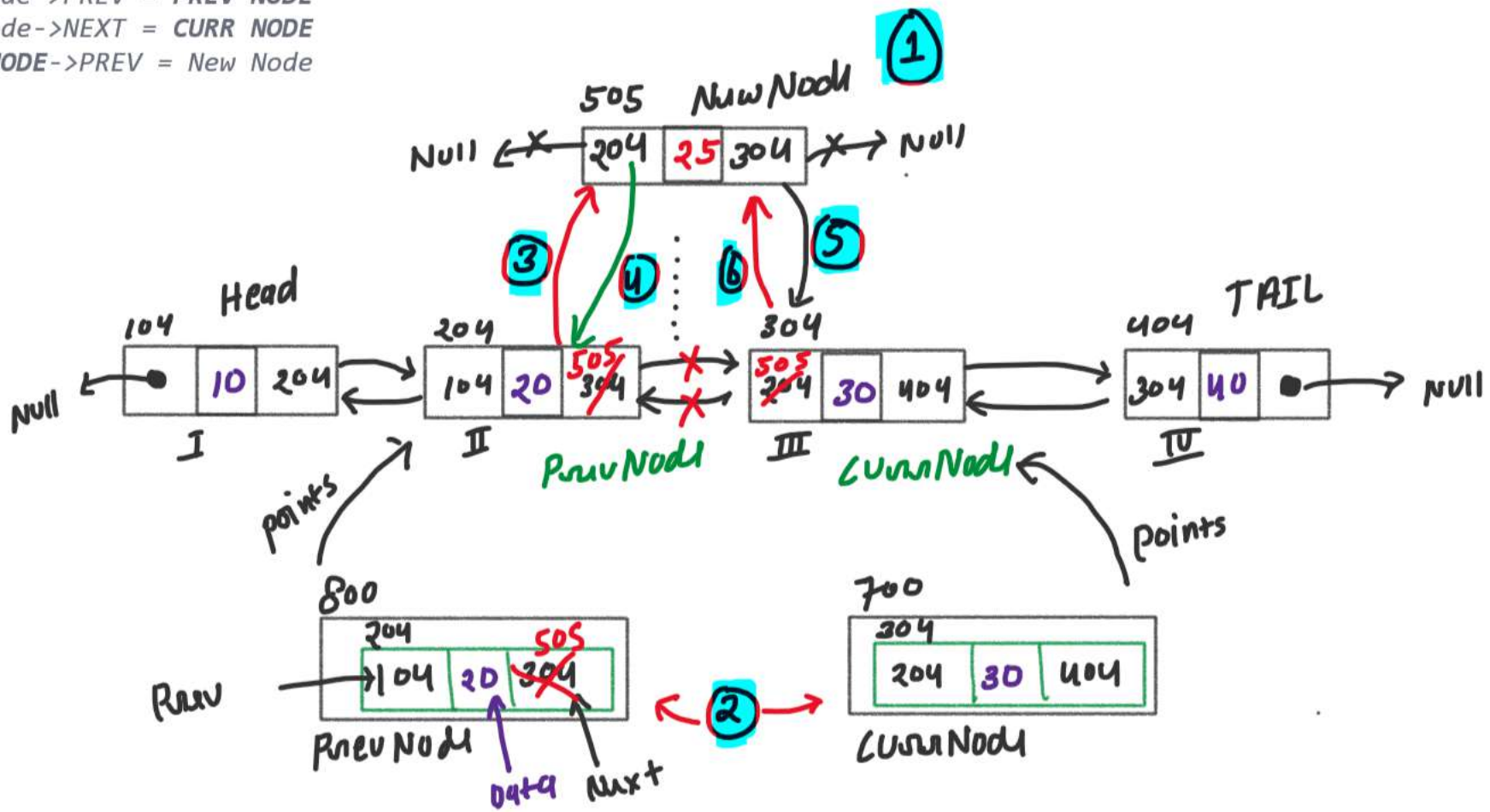




📂 *3. Insert node at any position*

Step 1: Create a new node
*Step 2: Set **PREV NODE** and **CURR NODE***
*Step 3: **PREV NODE**->NEXT = New Node*
*Step 4: New Node->PREV = **PREV NODE***
*Step 5: New Node->NEXT = **CURR NODE***
*Step 6: **CURR NODE**->PREV = New Node*

We want to insert a node at III position and it's value 25

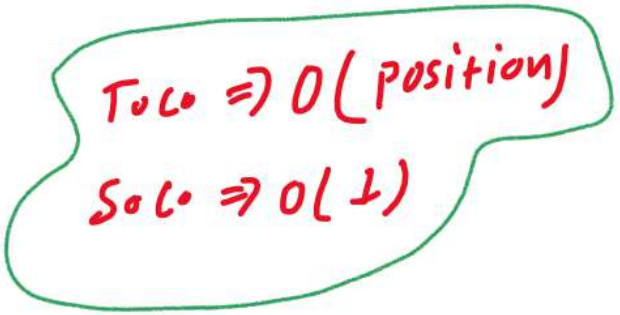


```cpp
// 📁 3. Insert node at any position
void insertAtPosition(Node* &head, Node* &tail, int
data, int position){
    if(position == 1){
        insertAtHead(head, tail, data);
    }
    else if(position == (findLength(head)+1)){
        insertAtTail(head, tail, data);
    }
    else{
        // Insert in middle of linked list
        // Step 1: Create a new node
        Node* newNode = new Node(data);

        // Step 2: Set PREV NODE and CURR NODE
        Node* prevNode = NULL;
        Node* currNode = head;

        while (position != 1)
        {
            prevNode = currNode;
            currNode = currNode->next;
            position--;
        }
        // Step 3: PREV NODE->NEXT = New Node
        prevNode->next = newNode;

        // Step 4: New Node->PREV = PREV NODE
        newNode->prev = prevNode;

        // Step 5: New Node->NEXT = CURR NODE
        newNode->next = currNode;

        // Step 6: CURR NODE->PREV = New Node
        currNode->prev = newNode;
    }
}
```

① pos == 1 insert at head

② pos == length+1 insert at tail

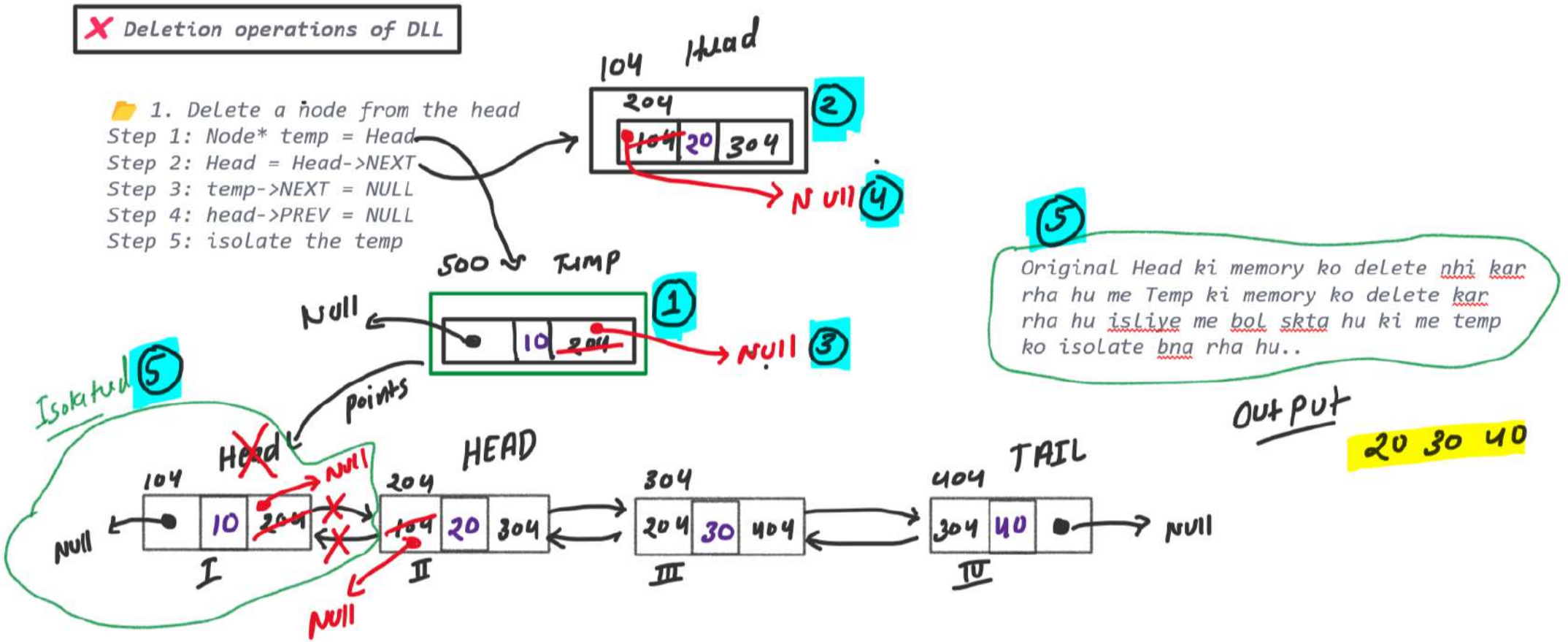
③ pos > 1 && pos < length insert at middle



✗ Deletion operations of DLL

📁 1. Delete a node from the head

Step 1: Node* temp = Head
Step 2: Head = Head->NEXT
Step 3: temp->NEXT = NULL
Step 4: head->PREV = NULL
Step 5: isolate the temp



5

Original Head ki memory ko delete nhi kar rha hu me Temp ki memory ko delete kar rha hu islive me bol skta hu ki me temp ko isolate bna rha hu..

Output

20 30 40

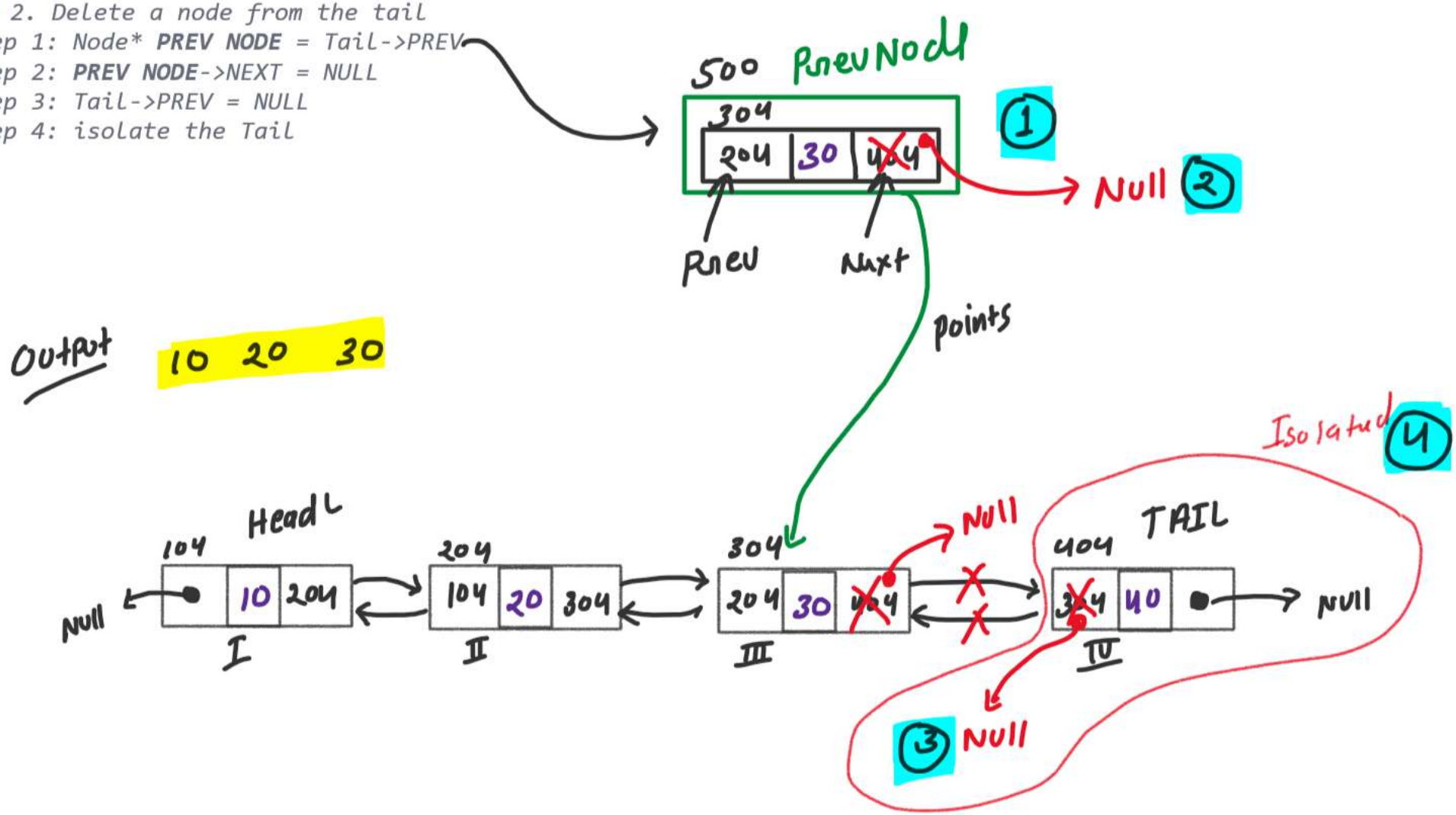
📂 2. Delete a node from the tail

Step 1: Node* **PREV NODE** = Tail->PREV

Step 2: **PREV NODE**->NEXT = NULL

Step 3: Tail->PREV = NULL

Step 4: isolate the Tail



```cpp
// 📂 1. Delete a node from the head
void deletefromHead(Node* &head, Node* &tail)
{
    // Set Corner Cases: Empty LL and Single Element LL
    if(head == NULL){
        // Empty LL
        cout << "Can't delete, because of empty linked list" << endl;
        return;
    }
    if(head == tail){
        // Single Element LL
        Node* temp = head;
        delete temp;
        head->next = NULL;
        tail->next = NULL;
        return;
    }
    else{
        // Step 1: Node* temp = Head
        Node* temp = head;

        // Step 2: Head = Head->NEXT
        head = head->next;

        // Step 3: temp->NEXT = NULL
        temp->next = NULL;

        // Step 4: head->PREV = NULL
        head->prev = NULL;

        // Step 5: isolate the temp
        delete temp;
    }
}
```

```cpp
// 📂 2. Delete a node from the tail
void deleteFromTail(Node* &head, Node* &tail)
{
    // Set Corner Cases: Empty LL and Single Element LL
    if(head == NULL){
        // Empty LL
        cout << "Can't delete, because of empty linked list" << endl;
        return;
    }
    if(head == tail){
        // Single Element LL
        Node* temp = head;
        delete temp;
        head->next = NULL;
        tail->next = NULL;
        return;
    }
    else{
        // Step 1: Node* PrevNode = Tail->PREV
        Node* prevNode = tail->prev;

        // Step 2: PrevNode->NEXT = NULL
        prevNode->next = NULL;

        // Step 3: Tail->PREV = NULL
        tail->prev = NULL;

        // Step 4: isolate the Tail
        delete tail;
    }
}
```

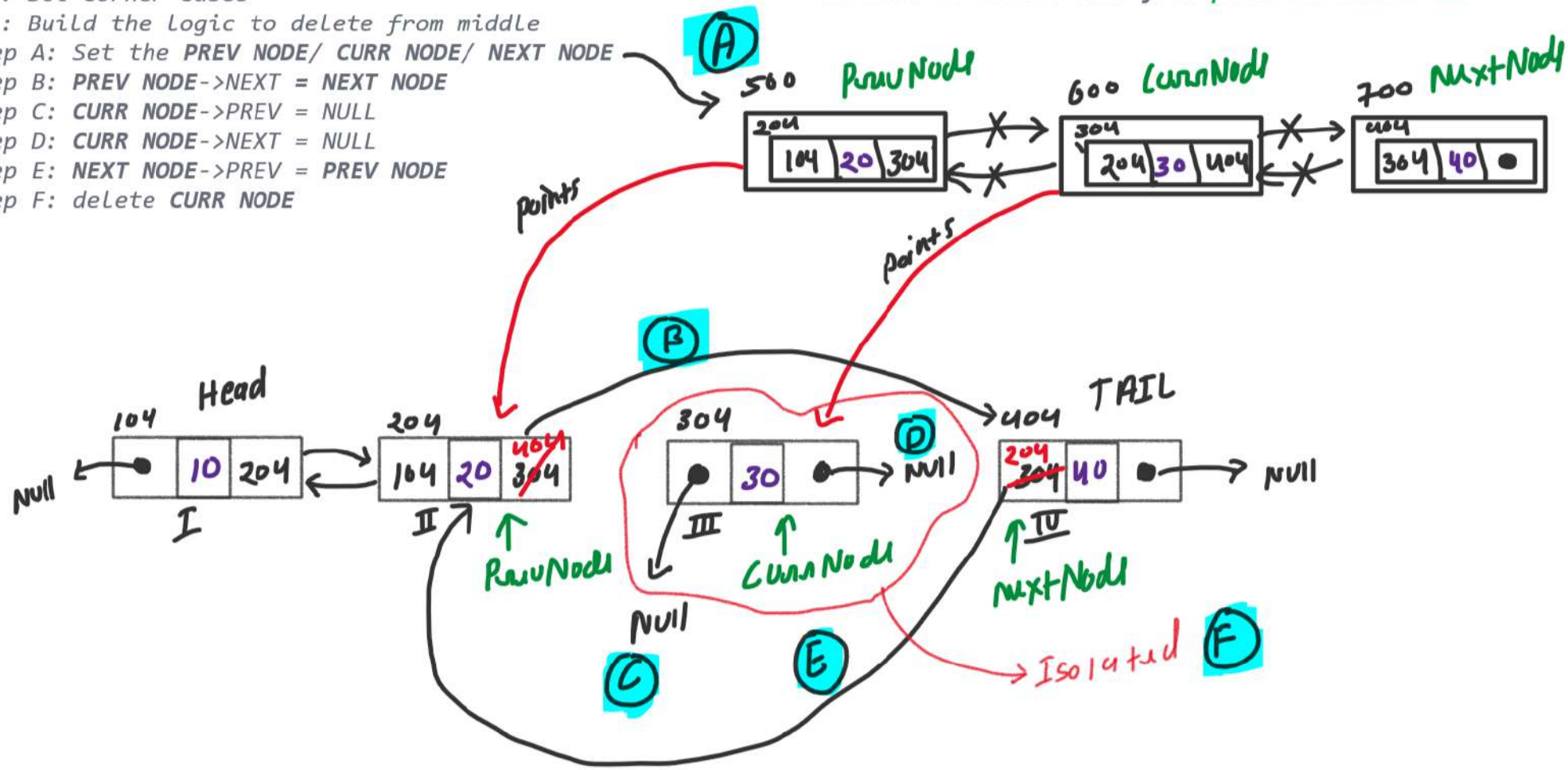
📂 *3. Delete a node from any position*

Step 1: Set Corner Cases

Step 2: Build the logic to delete from middle

- *Step A: Set the **PREV NODE/ CURR NODE/ NEXT NODE***
- *Step B: **PREV NODE**->NEXT = **NEXT NODE***
- *Step C: **CURR NODE**->PREV = NULL*
- *Step D: **CURR NODE**->NEXT = NULL*
- *Step E: **NEXT NODE**->PREV = **PREV NODE***
- *Step F: delete **CURR NODE***

*We want to delete node from **position number III***



```cpp
// 📂 3. Delete a node from any position
void deleteFromPosition(Node* &head, Node* &tail, int position)
{
    // Step 1: Set Corner Cases
    if(head == NULL){/*Single Element LL*/}
    if(head == tail){/*Single Element LL*/}

    if(position == 1){
        deletefromHead(head, tail);
    }
    else if(position == findLength(head)){
        deleteFromTail(head, tail);
    }
    else{
        // Step 2: Build the logic to delete from middle
        // Step A: Set the PREV NODE/ CURR NODE/ NEXT NODE
        Node* prevNode = NULL;
        Node* currNode = head;
        while (position != 1)
        {
            prevNode = currNode;
            currNode = currNode->next;
            position--;
        }
        Node* nextNode = currNode->next;

        // Step B: PREV NODE->NEXT = NEXT NODE
        prevNode->next =nextNode;

        // Step C: CURR NODE->PREV = NULL
        currNode->prev = NULL;

        // Step D: CURR NODE->NEXT = NULL
        currNode->next = NULL;

        // Step E: NEXT NODE->PREV = PREV NODE
        nextNode->prev = prevNode;

        // Step F: delete CURR NODE
        delete currNode;
    }
}
```

T.C. ⇒ O(Position) OR O(N)

S.C. ⇒ O(1)