CSCI 2270: Data Structures

Recitation #4 (Section 101)

Office Hours

- Name: Himanshu Gupta
  Email: himanshu.gupta@colorado.edu

- **Office Hours - 10am to 2pm on Mondays in ECAE 128**
  - In case that doesn't work for you, shoot me an email. We will figure something out that works for both of us.
  - Also, you can attend any TA's office hours. Timings are available on moodle in calendar.

Logistics

- From this recitation onwards, you will have two types of problems to solve in your recitation, **Silver Problem** and **Gold problem**.
  - It is **mandatory** for you to finish the “Silver Problem” and get it checked.
  - You are ENCOURAGED to solve the “Gold Problem” as it will improve your concepts.
  - Finish your code. If stuck, call us and ask for help.
  - Show your code to the TA or CA and ensure there is a tick against your name on the attendance sheet.

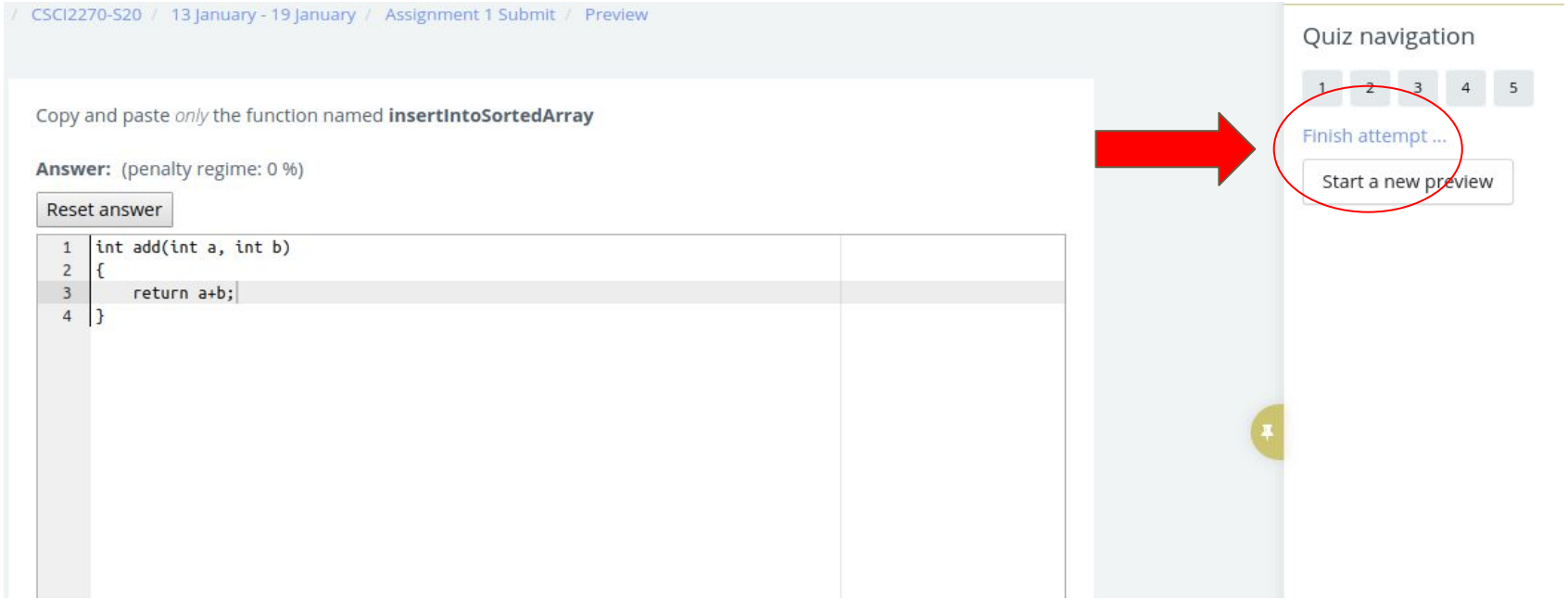
- **Assignment 3 is due on Sunday, February 9 2020, 11:59 PM.**
  GOOD LUCK!
  - **Stuff we will discuss today is what you need to do in Assignment 3.**

Logistics

- Recitation slides are posted on my github page.
  - Or do you guys prefer via email?

- Link - https://github.com/himanshugupta1009/CSCI-2270-Data-Structures

Please click on “Finish Attempt” after you are done!



Today's Agenda

- Review (20 ~ 30 mins)
  - Linked List
  - Traversing a Linked List
  - Insertion in a Linked List
  - Deleting from a Linked List
  - Assignment 3

- Exercise
  - Silver Problem - Delete a node from a linked list at a given position.
  - Gold Problem - Swap the first and the last node in an array.

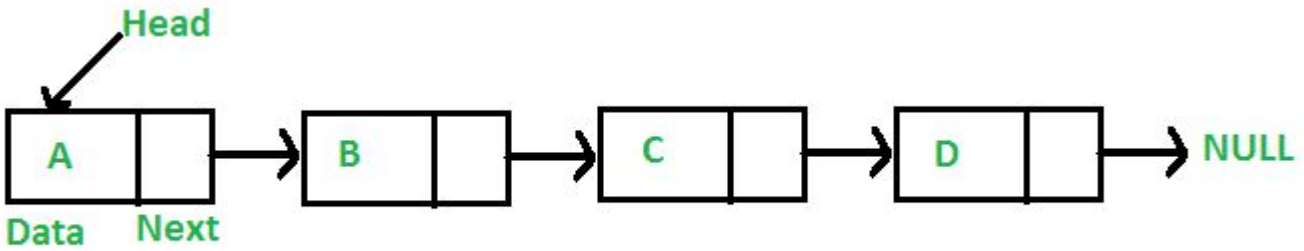
Any questions?

- On Course Logistics?
- Material from previous recitation/assignment

Linked Lists

- A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations.
  - Every element in a linked list is called a **“node”**
  - Most basic Linked list has a node with two objects in it.
    - int data; (value stored in that node)
    - *node next; (pointer to the next node)

- The elements in a linked list are linked using pointers as shown in the below image:



Nodes in Linked Lists

Using Structs

```
struct node
{
    int data;
    node *next;
};
```

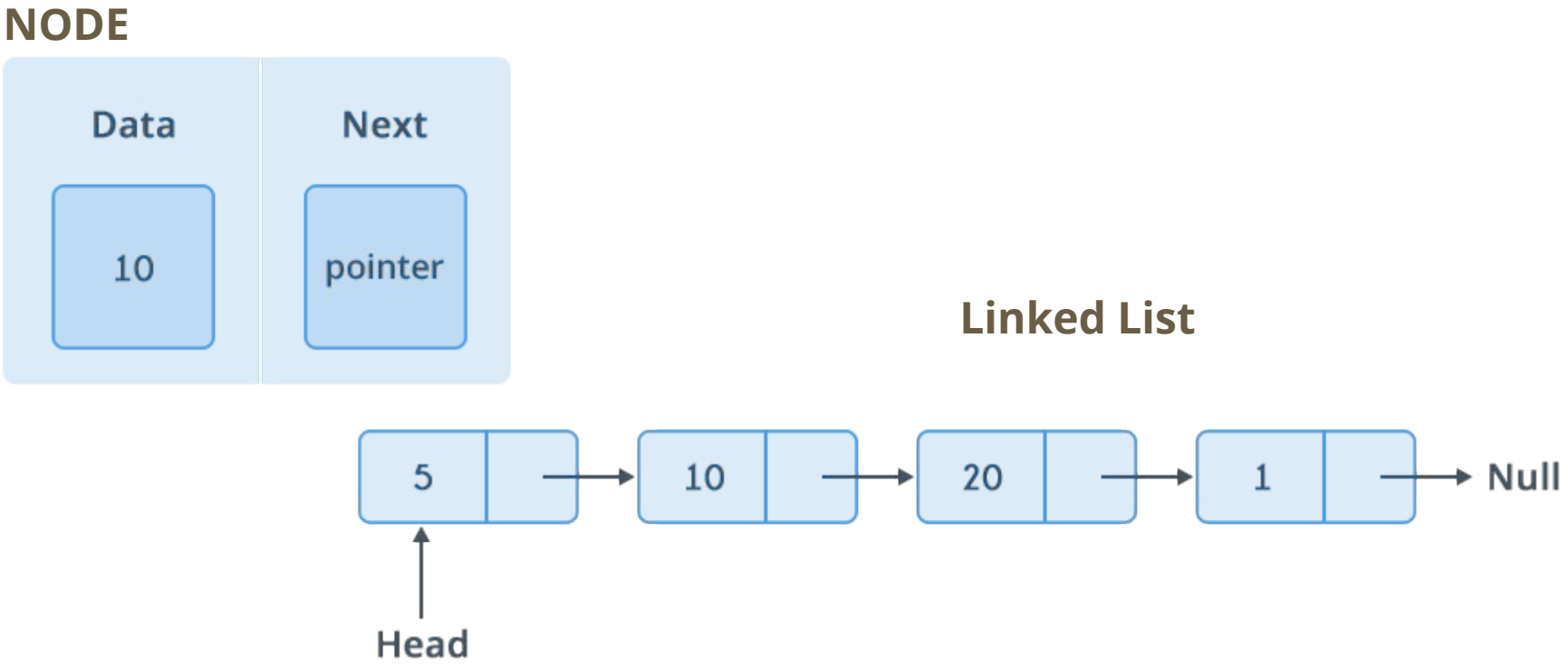
Using Class

```
class Node
{
Public:
        int data;
        Node* next;
};
```

Important things about Linked List

- Linked list is a collection of nodes which are linked using pointers.
- **“head”** - is a pointer that we want to point to the first element of the linked list. “head” always points to the first node in the linked list.
- The last node in the linked list always points to a **“NULL”** value.
  - This is how you figure out you have reached the end of a linked list.

This is what a linked list looks like



I have arrays to store data. Why do I need linked lists?

- Well, arrays always allocate memory in contiguous blocks. This leads to inefficient/ sub-optimal CPU memory usage. LLs solve this problem.
  - For example:

a[0]	a[1]	a[2]	c	d	b[0]	b[1]	b[2]	b[3]	b[4]	e

  - Now, let's say you delete c, d and e

a[0]	a[1]	a[2]			b[0]	b[1]	b[2]	b[3]	b[4]	

  - These intermediate empty memory blocks can never be used and this is wastage of your CPU memory which is undesirable.

- Also, arrays have predefined sizes. LLs have no predefined size or restriction (well, can't be more than the memory size of course)

Any questions?

Traversing a Linked List

- Traversing means to start at some node of a Linked List and keep moving through the linked list, one node at a time (either until you reach the “NULL” pointer at the end of a Linked List or some other termination criterion is met).

Traversing a Linked List and printing the elements

- ```
  Node* temp = head;    //Define a new pointer that points to the same position as head
  while(temp !=NULL)    // Loop till you reach the end of the linked list
  {
        cout<<temp->value<<endl;
        temp = temp->next;    // Make temp point to the next node in LL
  }
  ```
  ```

Traversing a Linked List and counting nodes

- ```
  int count=0;
  Node* temp = head;    //Define a new pointer that points to the same position as head
  while(temp !=NULL)    // Loop till you reach the end of the linked list
  {
        count++;              //Increment the counter
        temp = temp->next;    // Make temp point to the next node in LL
  }
  cout<<"The given linked list has  "<<count<<"  nodes"<<endl;
  ```
  ```

Insertion in a Linked List

- Three different scenarios in insertion:
  a. Insertion at the beginning of the linked list
  b. Insertion in the middle of the linked list
  c. Insertion at the end of the linked list

Insertion in a Linked List

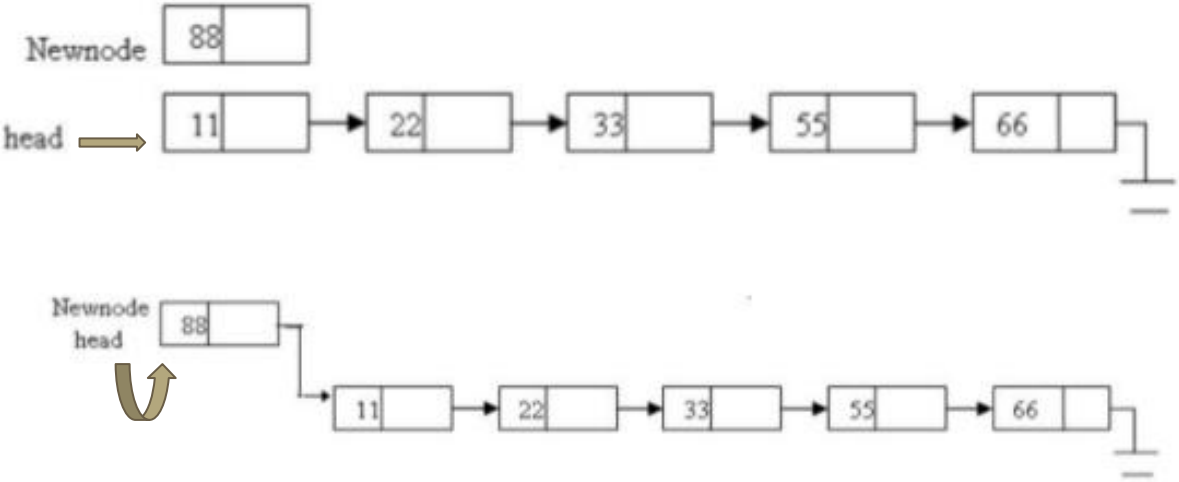
- Three different scenarios in insertion:
  a. Insertion at the beginning of the linked list
  b. Insertion in the middle of the linked list
  c. Insertion at the end of the linked list

- First step is to create a new node that you wish to insert. (but how?)

```
Node* newNode = new Node;
newNode->key = value;
newNode->next = NULL;
```

```
Node* newNode = new Node;
(*newNode).key = value;
(*newNode).next = NULL;
```

Insertion at the beginning of the linked list

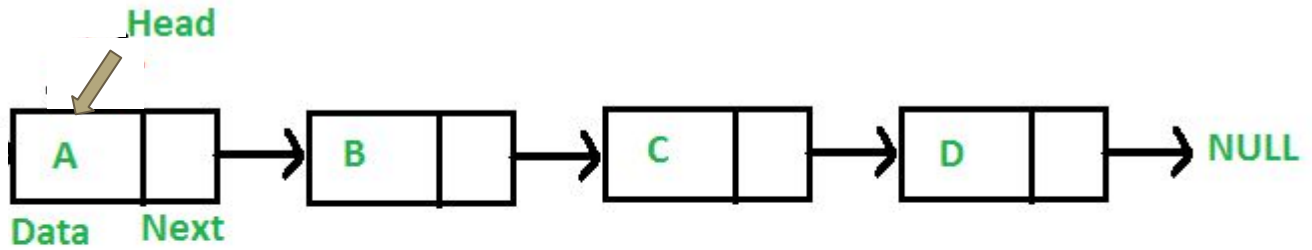


Insertion at the beginning of the linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Make it point to the current head of the LL
newNode->next = head;
//Make your head point to the new node
head = newNode;
```

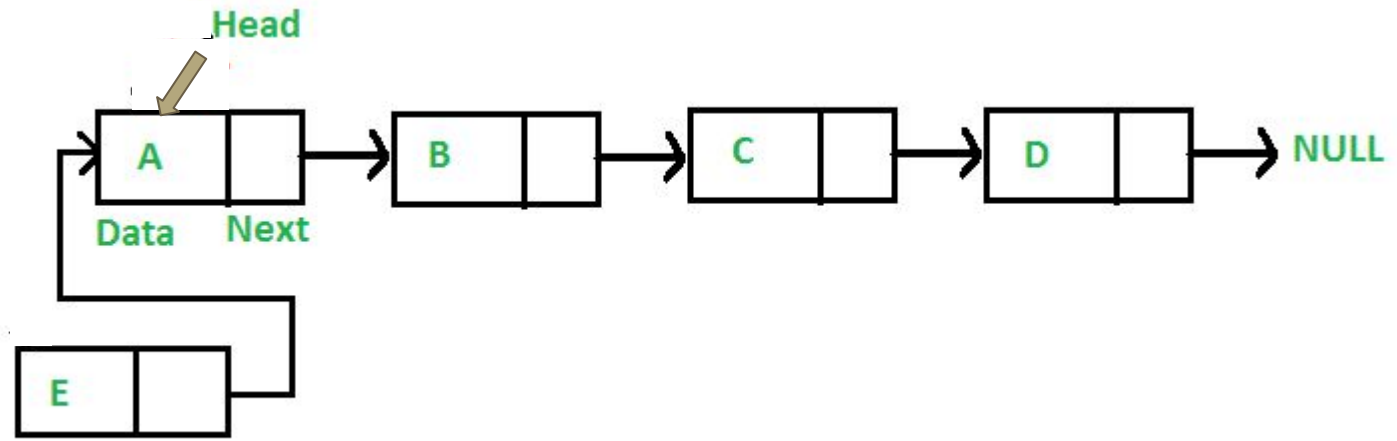
Insertion at the beginning of the linked list

STEP 1



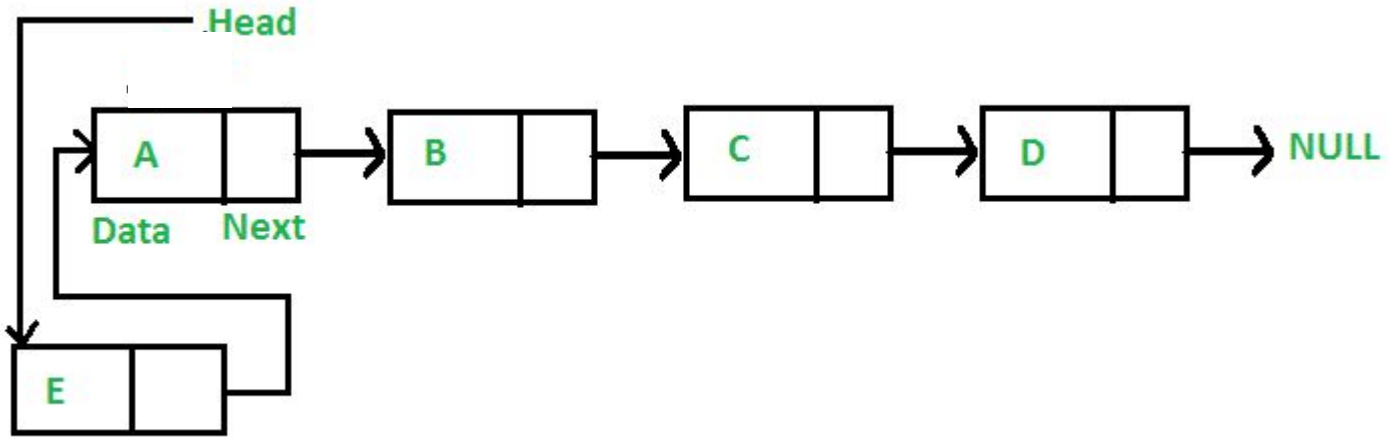
Insertion at the beginning of the linked list

STEP 2

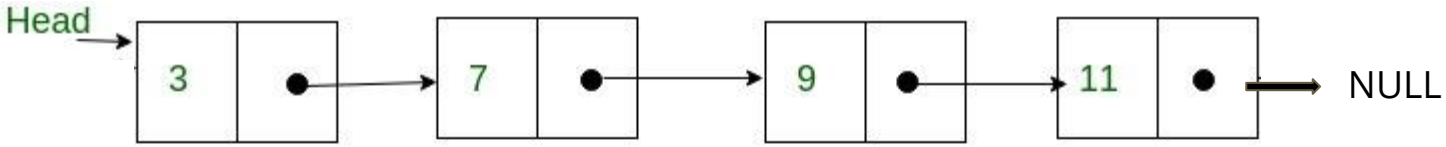


Insertion at the beginning of the linked list

STEP 3

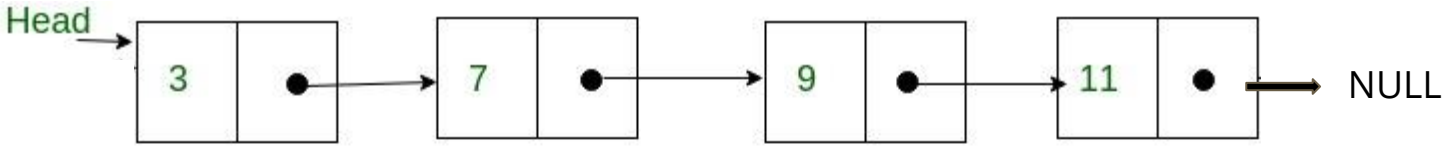


Insertion in the middle of a linked list

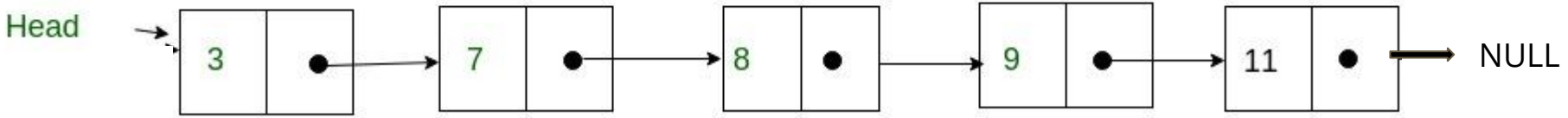


- Insert a node with value 8 at index 2. What does it mean?

Insertion in the middle of a linked list



After inserting 8, the above LL should be changed to the following

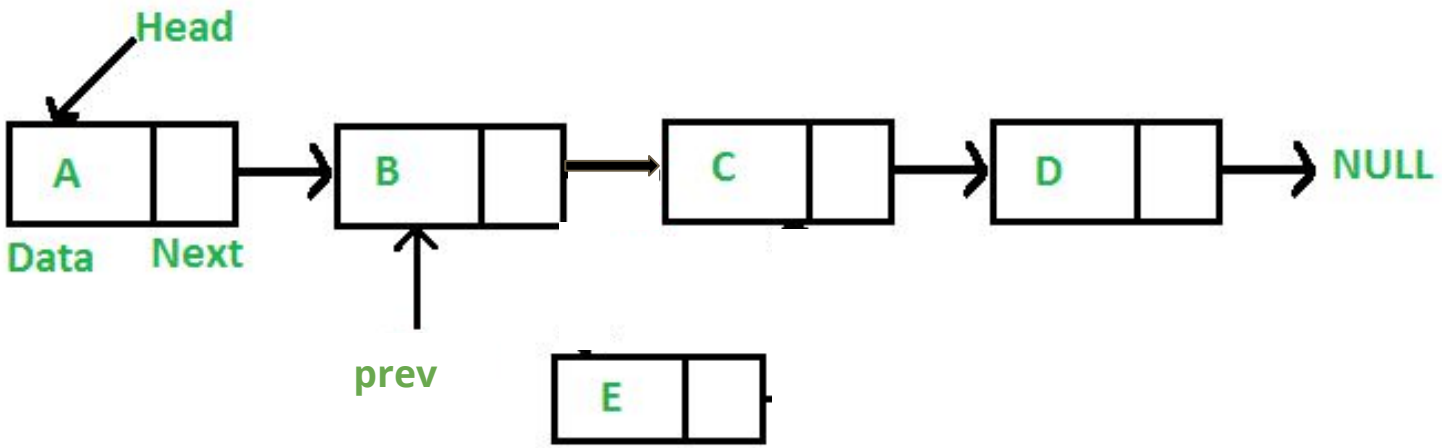


Insertion in the middle of a linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Assume who have the pointer pointing to the previous node. Call it
prev.
//Make your new node point to the prev pointer's next
newNode->next = prev->next;
//Make your prev point to the new node
prev->next = newNode;
```

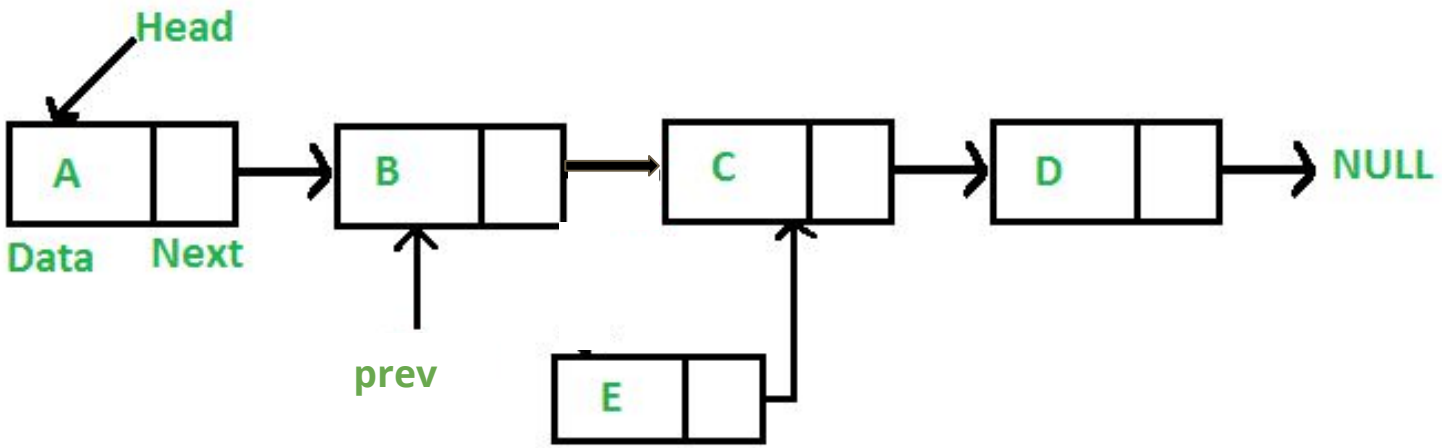
Insertion in the middle of a linked list

STEP 1



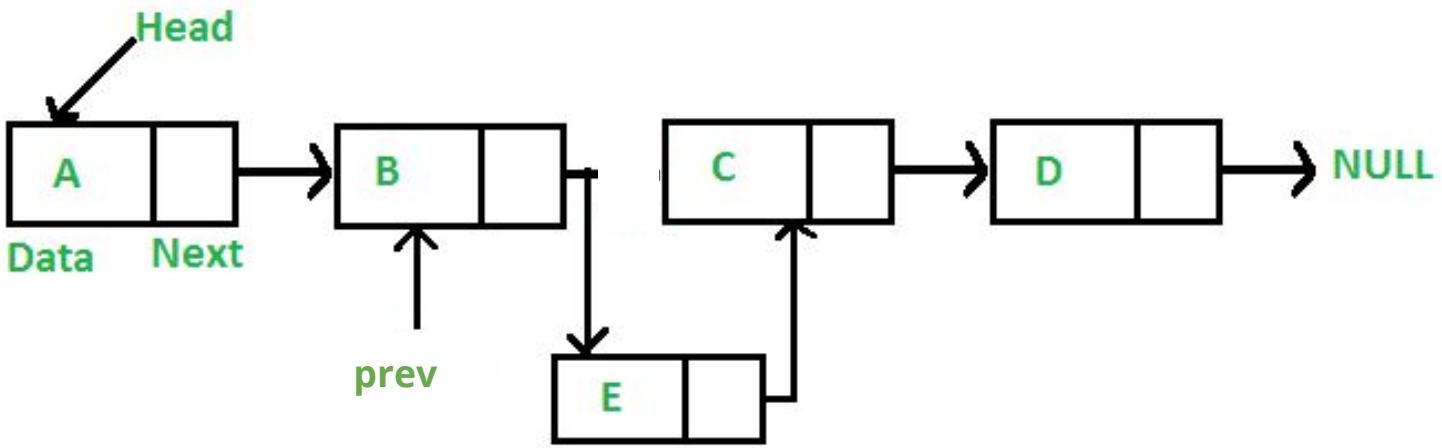
Insertion in the middle of a linked list

STEP 2



Insertion in the middle of a linked list

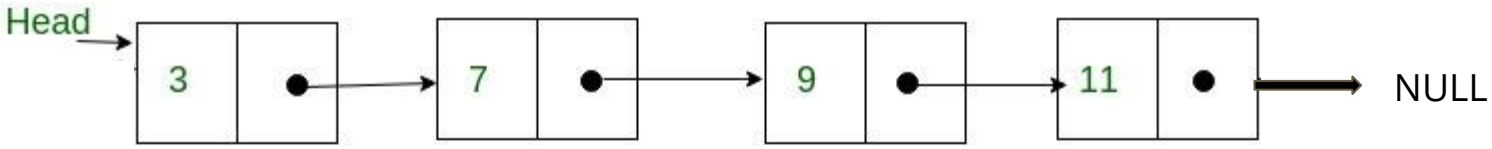
STEP 3



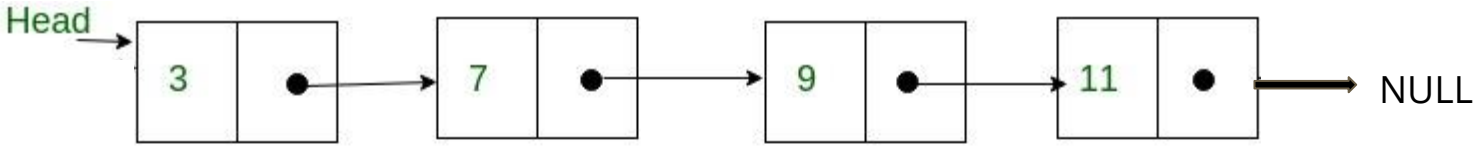
Insertion in the middle of a linked list

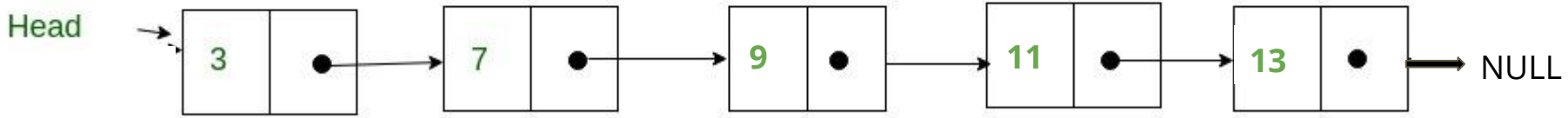
- In most of the cases you are either given the **“prev”** pointer or you have to traverse the array and find it (based on some condition).

Insertion at the end of a linked list



Insertion at the end of a linked list



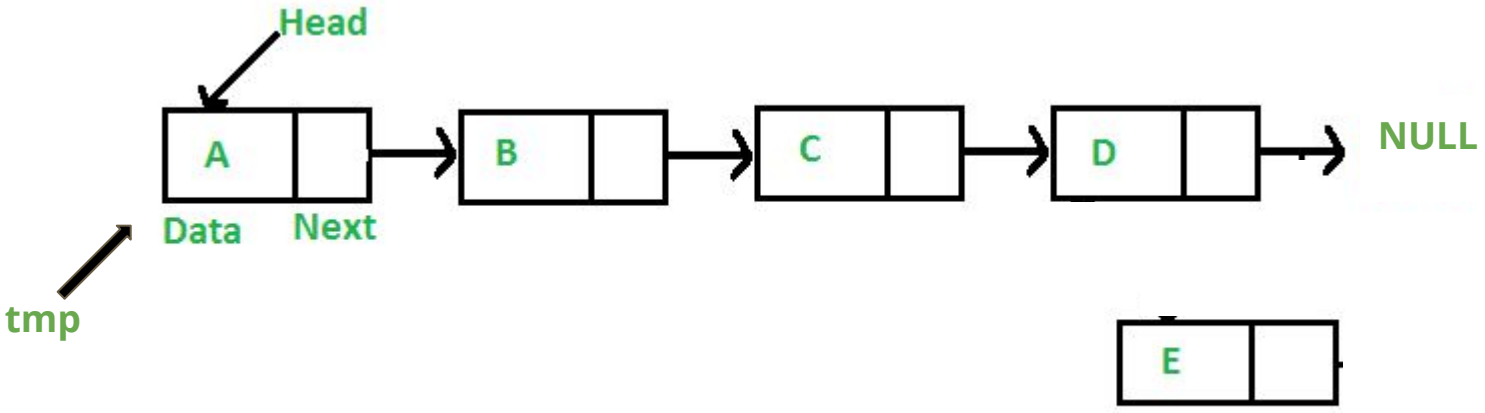


Insertion at the end of a linked list

```
//Create a new node
Node* newNode = new Node;
newNode->key = newKey;
//Traverse the linked list to reach the last node in the list
Node* tmp = head;
while( tmp->next != NULL){
        tmp = tmp->next;
}
//temp pointer now points to the last node in the LL
//Make your last node point to the new node
tmp->next = newNode;
//Make your newNode point to NULL because the last element in a LL
always points to NULL
newNode->next = NULL;
```

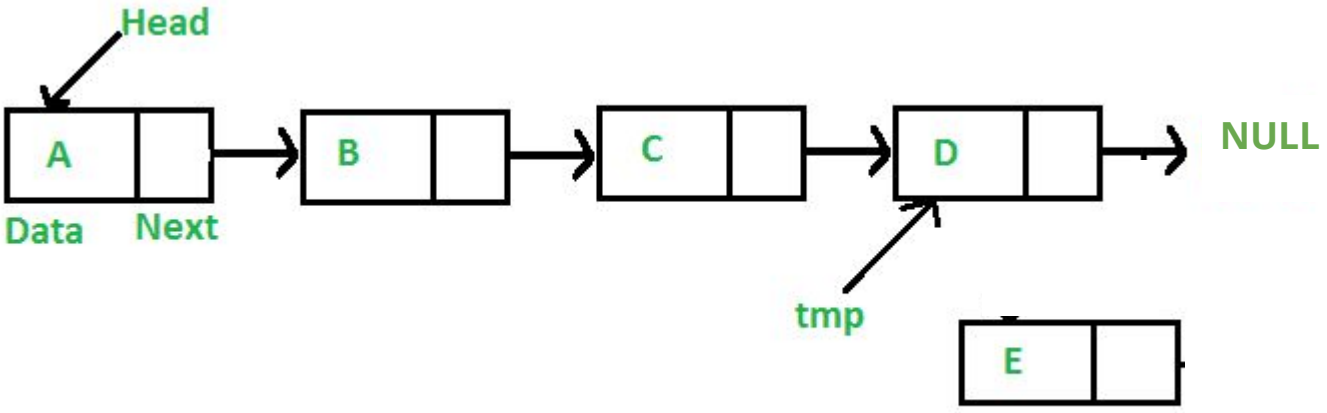
Insertion at the end of a linked list

STEP 1



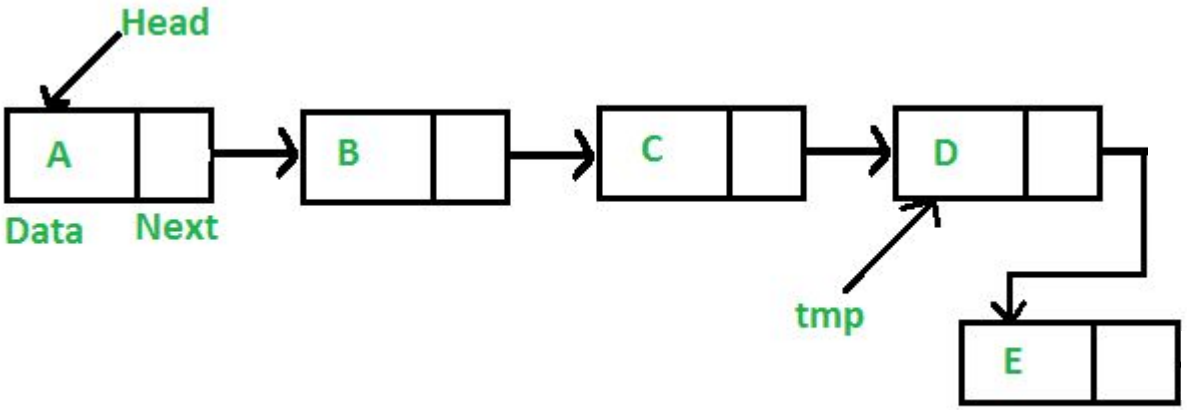
Insertion at the end of a linked list

STEP 2



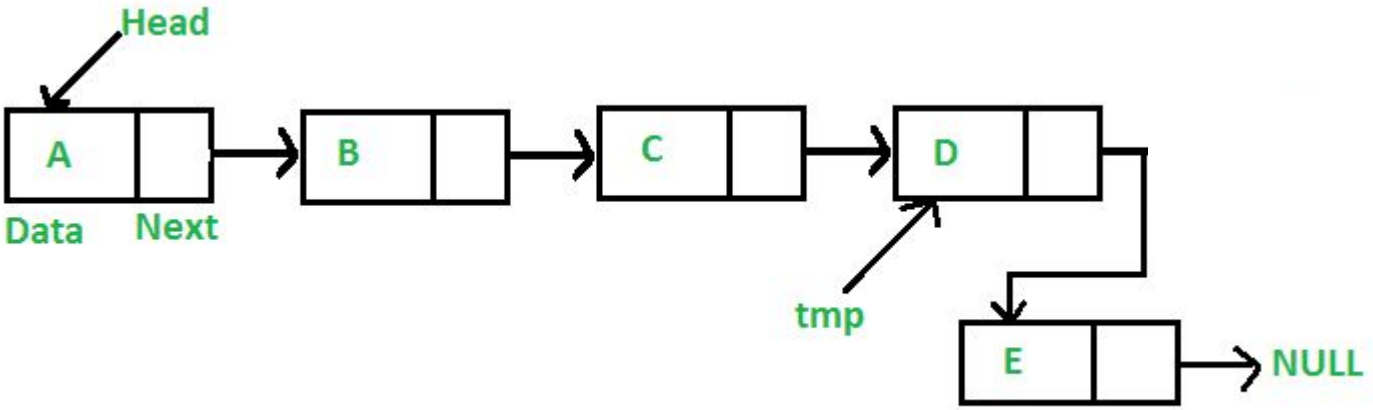
Insertion at the end of a linked list

STEP 3



Insertion at the end of a linked list

STEP 4



Insertion in a linked list

```
void insert(Node* prev, int newKey){

//Check if head is Null i.e list is empty
        if(head == NULL){
                head = new Node;
                head->key = newKey;
                head->next = NULL;
        }

  // if list is not empty, look for prev and
append our node there
        else if(prev == NULL) {
                Node* newNode = new Node;
                newNode->key = newKey;
                newNode->next = head;
                head = newNode;
  }
  else{
  Node* newNode = new Node;
  newNode->key = newKey;
newNode->next = prev->next;
prev->next = newNode;
}
  }
```

Any questions?

Deletion in a Linked List

- Three different scenarios in deletion:
  a. Delete from the beginning of the linked list
  b. Delete from the middle of the linked list
  c. Delete at the end of the linked list

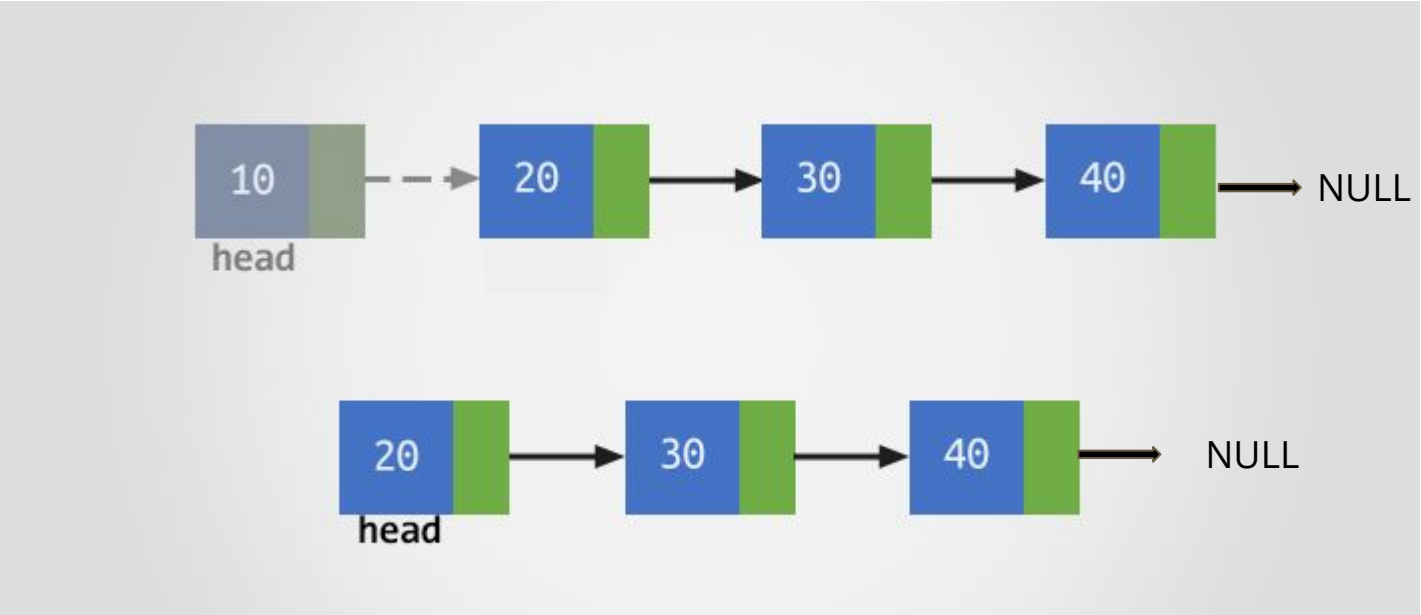
Deletion in a Linked List

- First the position at which we want to delete must be found or the node that we want to delete must be found. Call it **“to_be_deleted”**

- After that,
  a. find the node just before the node that we wish to delete. Call it **“prev”**
  b. find the node just after the node that we wish to delete. Call it **“after”**
  c. Modify “prev->next” to point to the “after” node.
     (This step is performed to ensure that there is no break in your linked list)
     prev->next = after;

- Delete the memory used by the node you want to delete.
  delete to_be_deleted;

Deletion at the beginning of a linked list

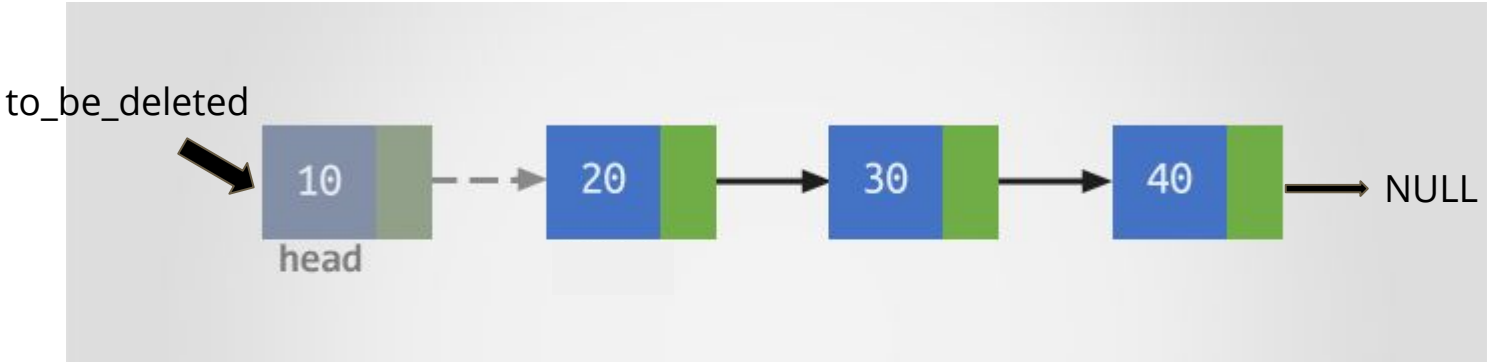


Deletion at the beginning of a linked list

```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//Make your head point to next node of the LL (this is the 2nd node in LL )
head = head->next;
//delete your to_be_deleted node
delete to_be_deleted;
```

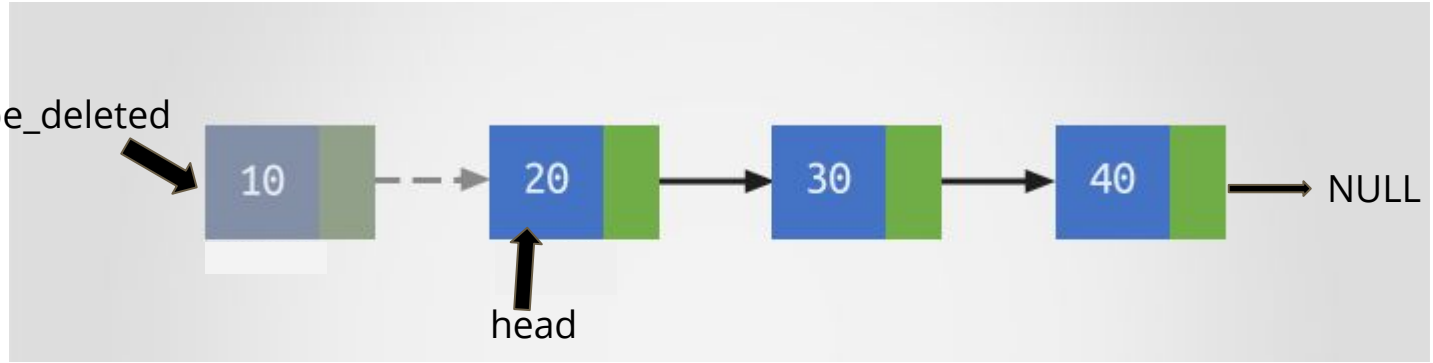
Deletion at the beginning of a linked list

STEP 1



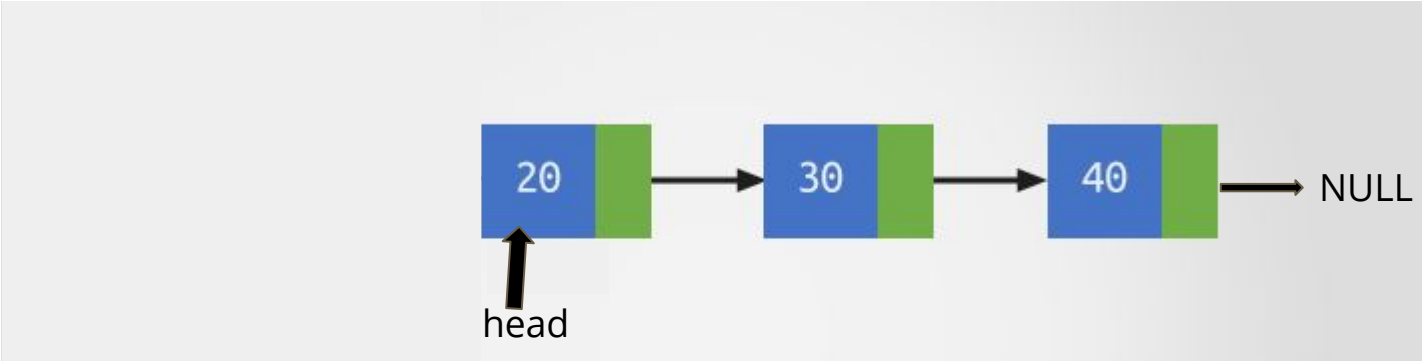
Deletion at the beginning of a linked list

STEP 2

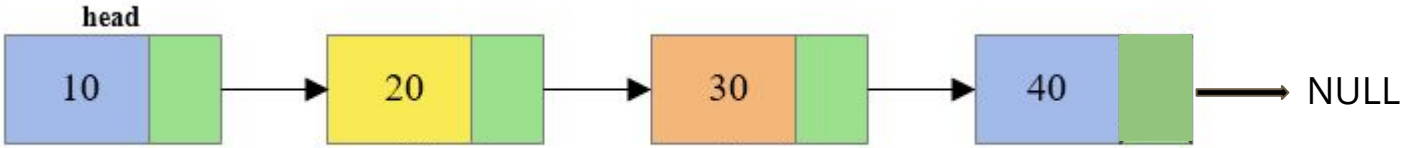


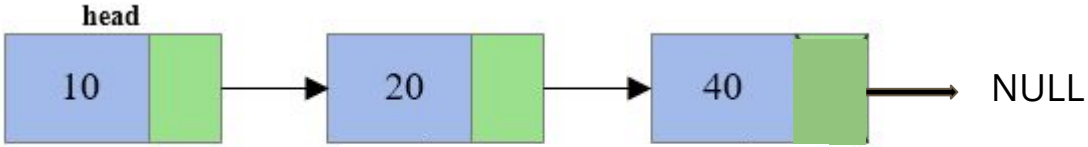
Deletion at the beginning of a linked list

STEP 3



Deletion in the middle of a linked list

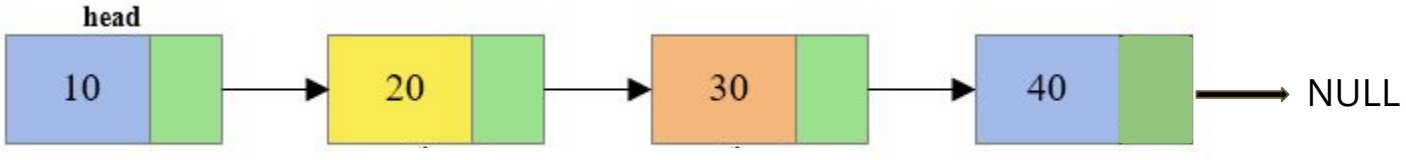




Deletion in the middle of a linked list

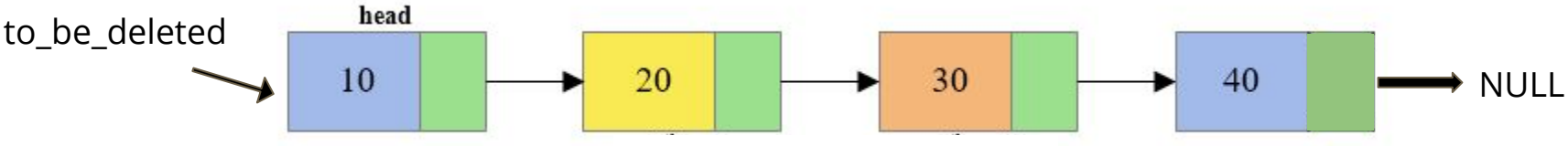
```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//pointer to access the node previous to node that will be deleted
Node* prev = NULL
//Suppose you have been asked to delete the node at index n
//Traverse the linked list till you reach the nth node in the LL
int index=0;  //to check if you have reached the correct index in the LL
while(index!=n && to_be_deleted->next !=NULL){
        prev = to_be_deleted;
        to_be_deleted = to_be_deleted->next;
        index++;
}
//Make your prev point to the node right after your “to_be_deleted” node
prev->next = to_be_deleted->next;
//Delete your “to_be_deleted” node
delete to_be_deleted;
```

Deletion in the middle of a linked list



Deletion in the middle of a linked list

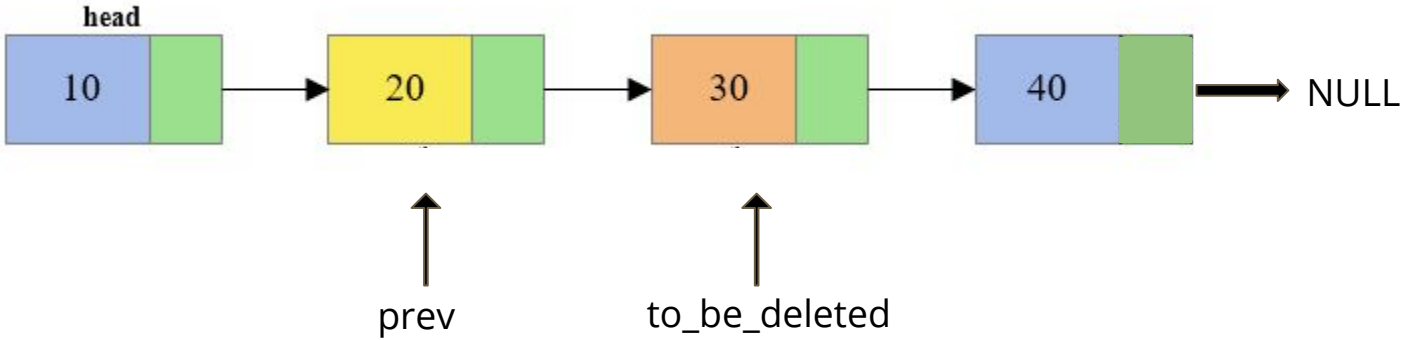
STEP 1



prev = NULL

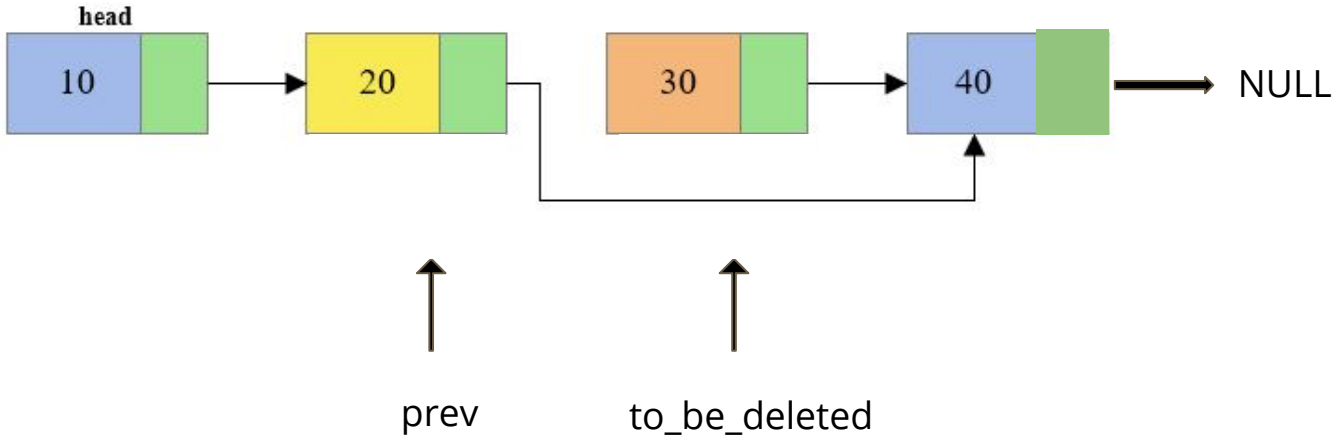
Deletion in the middle of a linked list

STEP 2



Deletion in the middle of a linked list

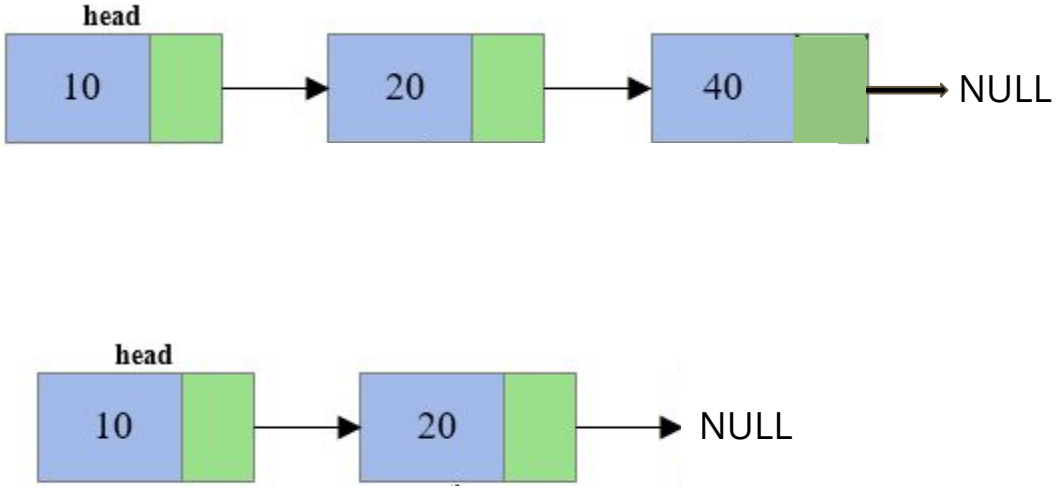
STEP 3



Deletion in the middle of a linked list

STEP 4

Deletion at the end of a linked list

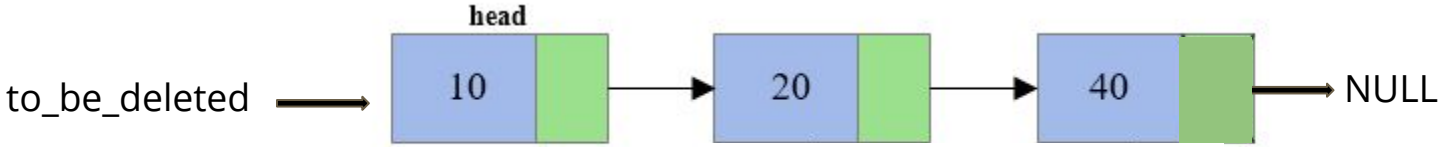


Deletion at the end of a Linked List

```
//Create a new pointer and make it point to the first node in the LL
Node* to_be_deleted = head;
//pointer to access the node previous to node that will be deleted
Node* prev = NULL
//Traverse the linked list till you reach the last node in the LL
while( to_be_deleted->next !=NULL){
        prev = to_be_deleted;
        to_be_deleted = to_be_deleted->next;
}
//Make your prev point to NULL
prev->next = NULL;
//Delete your “to_be_deleted” node
delete to_be_deleted;
```

Deletion at the end of a linked list

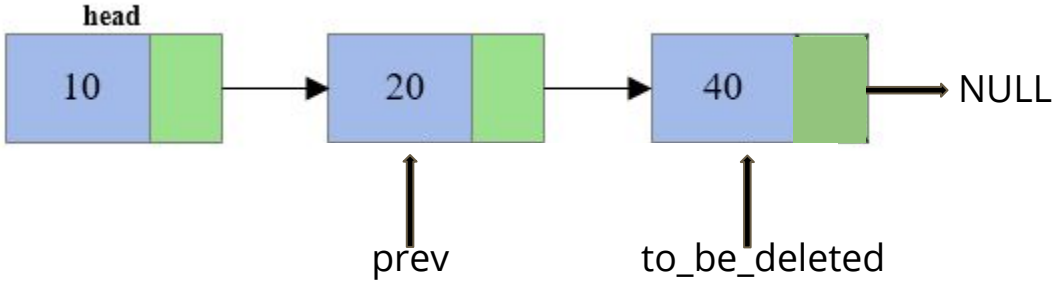
STEP 1



prev = NULL

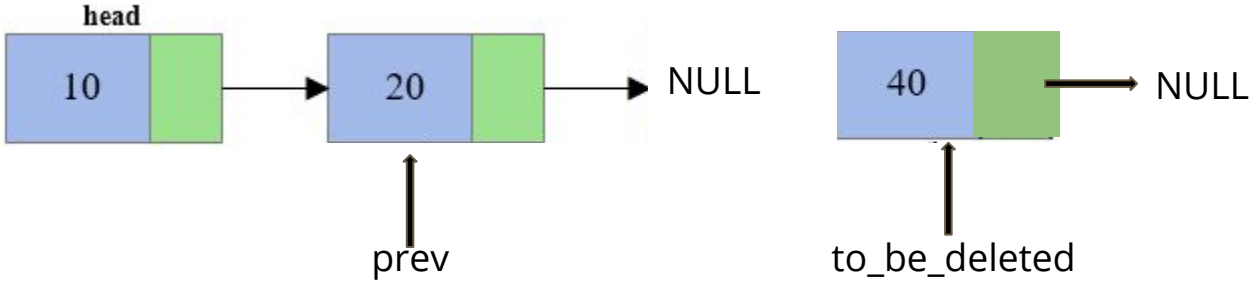
Deletion at the end of a linked list

STEP 2



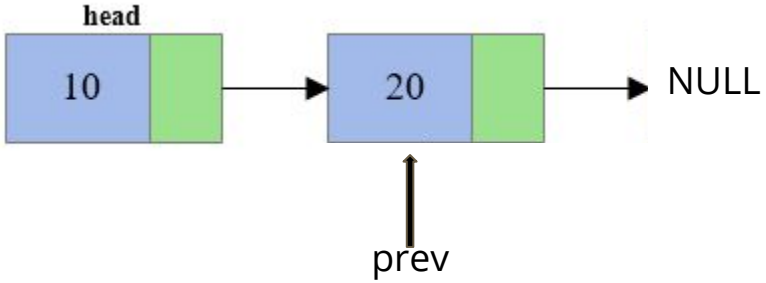
Deletion at the end of a linked list

STEP 3



Deletion at the end of a linked list

STEP 4



Any questions?

Few common errors

- Always check if your Linked list is currently empty or not.
  - How to do that?

Few common errors

- Always check if your Linked list is currently empty or not.
  - How to do that?
  - Check if head == NULL
    - If True, that means it is empty

Few common errors

- Always check if your Linked list is currently empty or not.
  - How to do that?
  - Check if head == NULL
    - If True, that means it is empty

- **REMEMBER: “head” pointer should always point to the first node in the linked list ALL THE TIME.**
  If there are no entries in the LL, then head is equal to NULL.

Exercise (Silver Problem)

- This problem is mandatory.
- You will be implementing the deleteAtIndex(int n) function in LinkedList.cpp
- Compile both main.cpp and LinkedList.cpp together for successful compilation.
  - g++ -std=c++11 main.cpp LinkedList.cpp -o rec4
  - ./rec4

Exercise (Gold Problem)

- This problem is not mandatory but you are highly encouraged to solve it.
- You will be implementing the swapFirstAndLast() function in LinkedList.cpp
- Once again, compile both main.cpp and LinkedList.cpp together for successful compilation.
  - g++ -std=c++11 main.cpp LinkedList.cpp -o rec4
  - ./rec4

Expected Output

```
himanshu@Mercury:~/Documents/Padhai/Spring 2020/CSCI 2270/Recitations_Himanshu/Recitation#4$
himanshu@Mercury:~/Documents/Padhai/Spring 2020/CSCI 2270/Recitations_Himanshu/Recitation#4$
Adding nodes to List:
2
-1 -> 2
-1 -> 2 -> -7
-1 -> 2 -> -7 -> 10
-1 -> 2 -> -7 -> 10 -> 3
-1 -> 2 -> -7 -> 10 -> 3 -> 5
-1 -> 2 -> -7 -> 10 -> 3 -> 5 -> -4

Running delete function.
Deleting node at index: 3
-1 -> 2 -> -7 -> 3 -> 5 -> -4

Deleting at index: 0
2 -> -7 -> 3 -> 5 -> -4

Swapping First and last nodes
-4 -> -7 -> 3 -> 5 -> 2

Swapping a Linked List with 2 nodes
-1 -> 2

Swapping First and last nodes
2 -> -1
```

Exercise (Gold Problem)

- **Useful Hints:**
  - You first need to traverse till the last node of the linked list.
  - You might want to keep track of the previous node of the last node as well. Call it “prev”
  - Now what?
    - Make your last node point to second node in the LL.
    - Modify your “prev” node to point to the first node in the LL.
    - Modify “head” to point to the correct node in the swapped linked list.
    - Ensure that the last node in the LL is pointing to NULL.

- **Edge cases:**
  - Treat the case when linked list has just 2 elements separately. Hints mentioned above can get you stuck in an infinite loop in this case.