```
video * ptr;
```


```
ptr = new video;
```

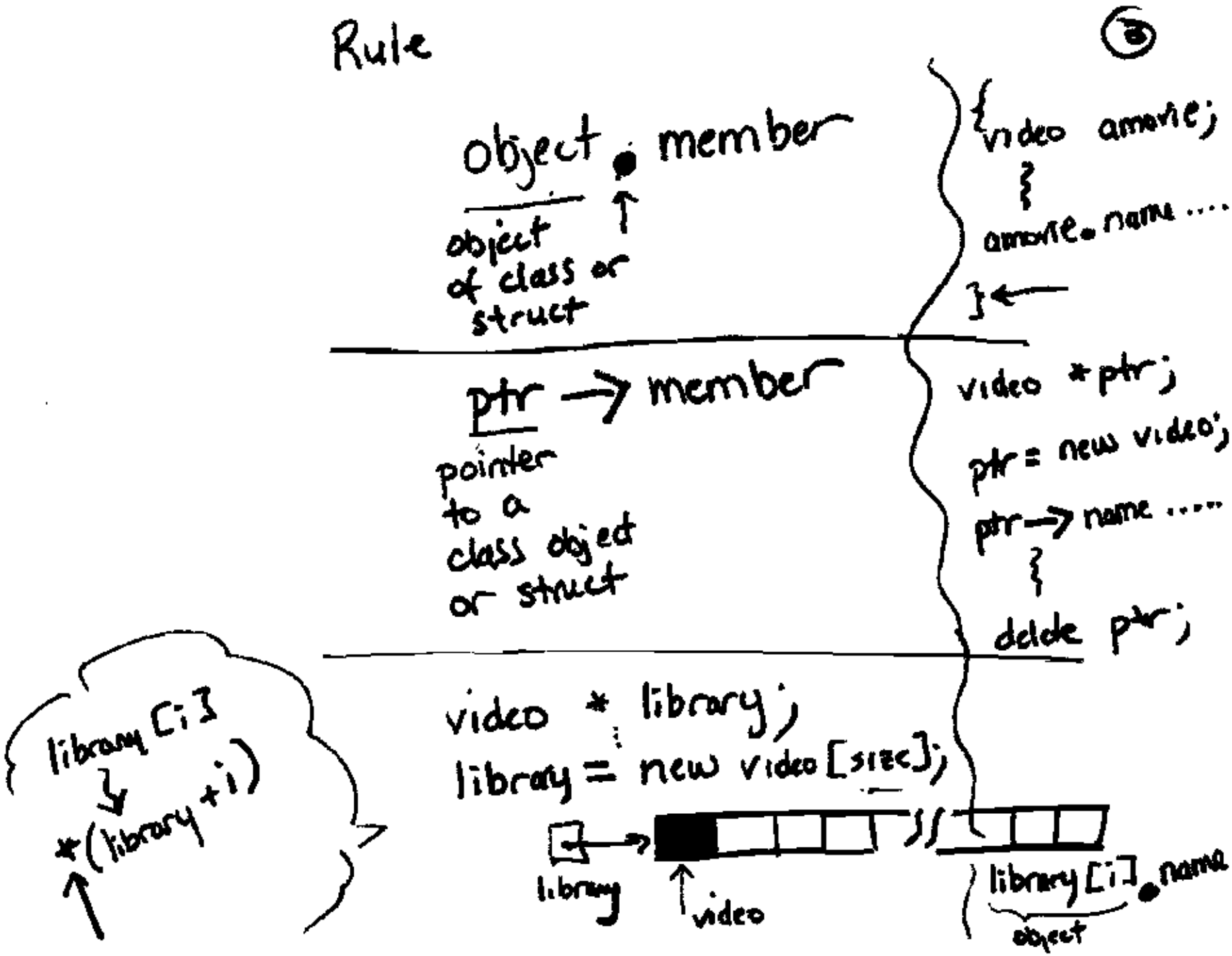

```
(*ptr).name = new char[strlen(movie)+1];
```

Direct Member Access operator

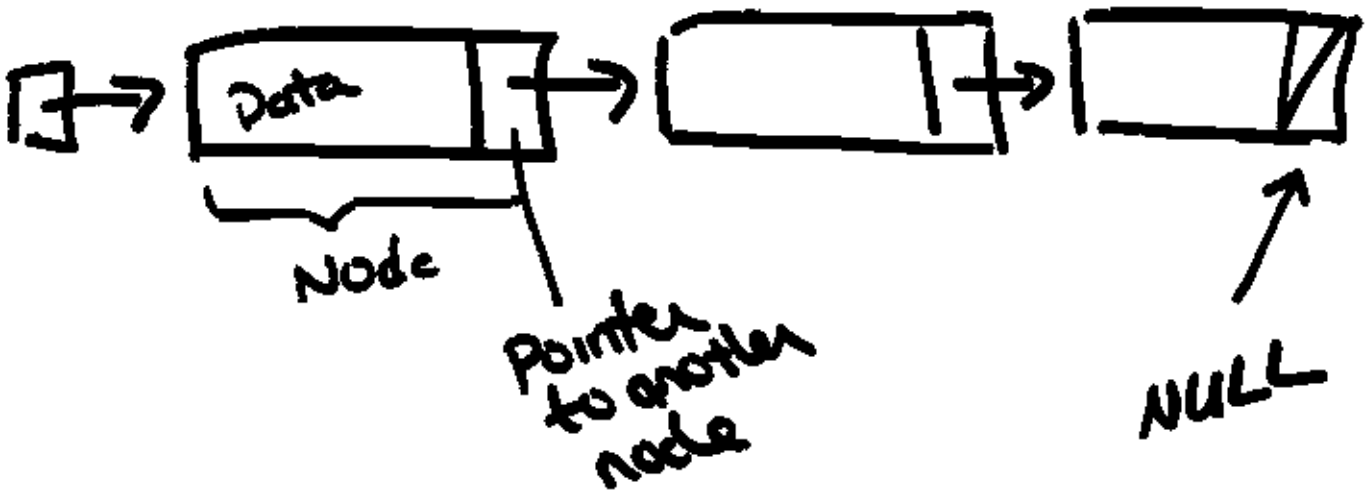
```
ptr->name =
```

↑ indirect member access operator

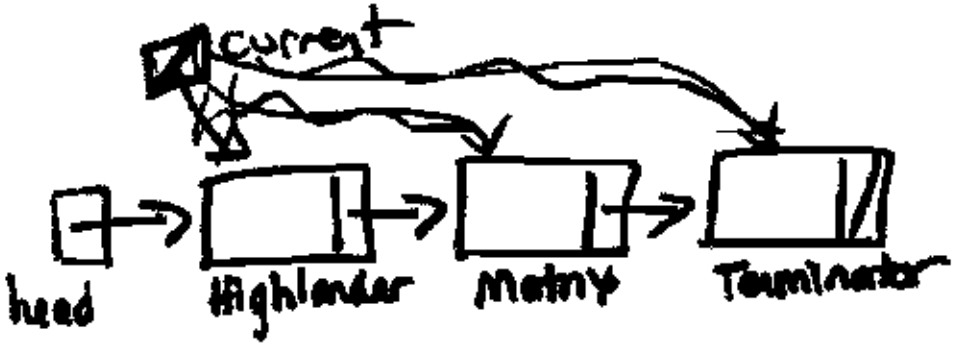
→


Rule
object . member
object of class or struct
ptr -> member
pointer to a class object or struct
{video amovie;
amovie.name ....
}
video *ptr;
ptr = new video;
ptr -> name ....
delete ptr;
video * library;
library = new video [size];
library
video
library[i].name
object
library[i]
*(library+i)

Linear Linked List (LLL)




```
Struct Node
{
    Video amovie;
    Node * next;
};
```

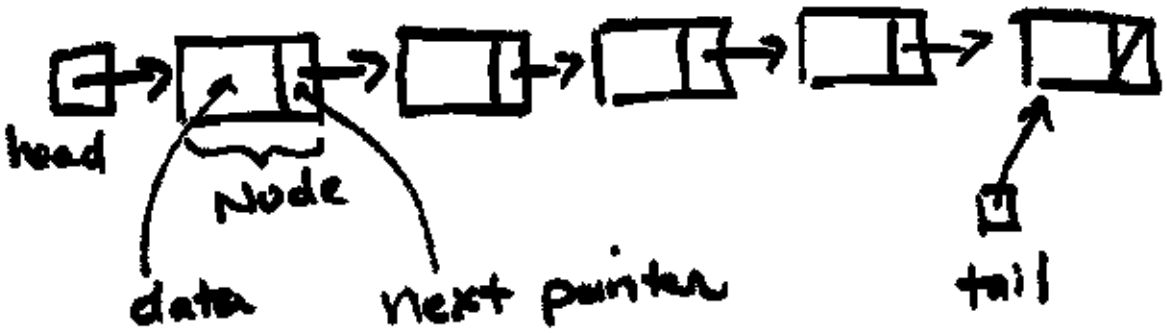


Display the Contents Traverse

```
node * current = head; // both pointers
                       // point to the
                       // same place
while (current)
{                      // while (current != NULL)
    current->movie.display();
    current = current -> next; //traverse
}
```

Linear Linked List

LLL

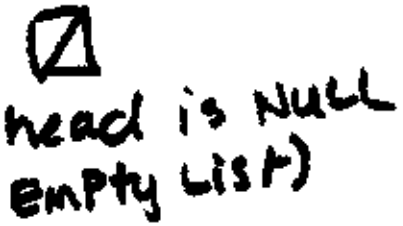


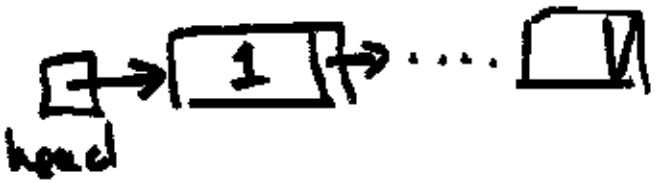
Insert

- Insert at beginning
- Insert at end without a tail pointer
- " " " with " "
- Insert in sorted order

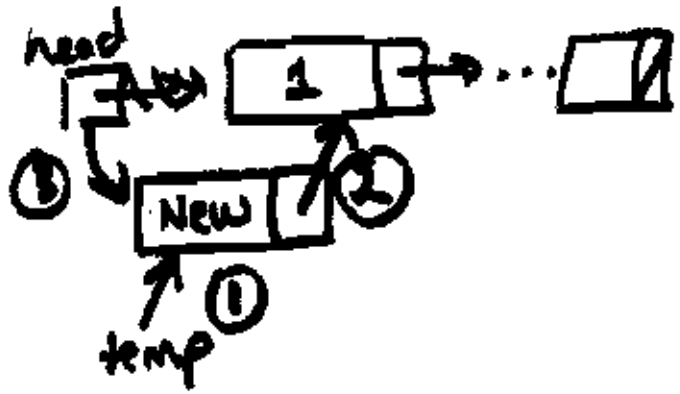
Insert at Beginning

Before

1. 

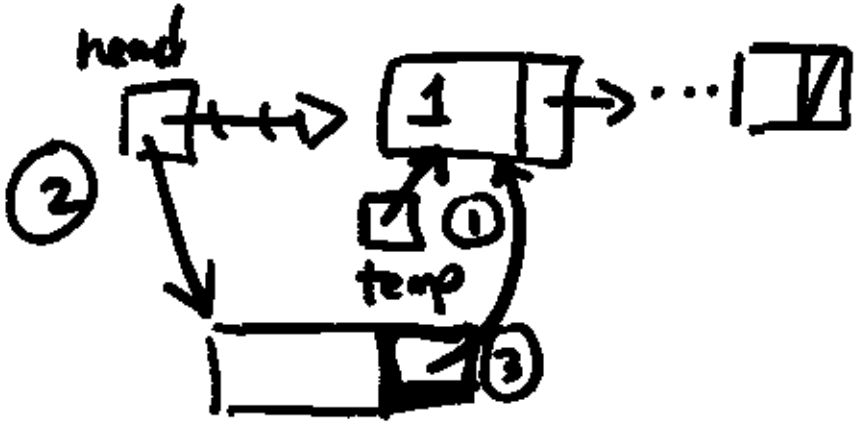
2. 

After



OR...

```
node *temp;
temp = head;
head = new node;
head->next = temp;
```



```
if ( head == NULL)  // case 1
                    // if ( !head )
{
    head = new node;
    head->next = NULL;
    // store the data
    // head->movie.read();
    // tail = head;
}
else  // case 2
{   node * temp;
    temp = new node;
    // store the data
    // temp->movie.read();
    temp->next = head;
    head = temp;
}
```

step 1

step 2

Grab the address in head's memory

```
struct node
{   video movie;
    node * next;
};
node * head = NULL;
```

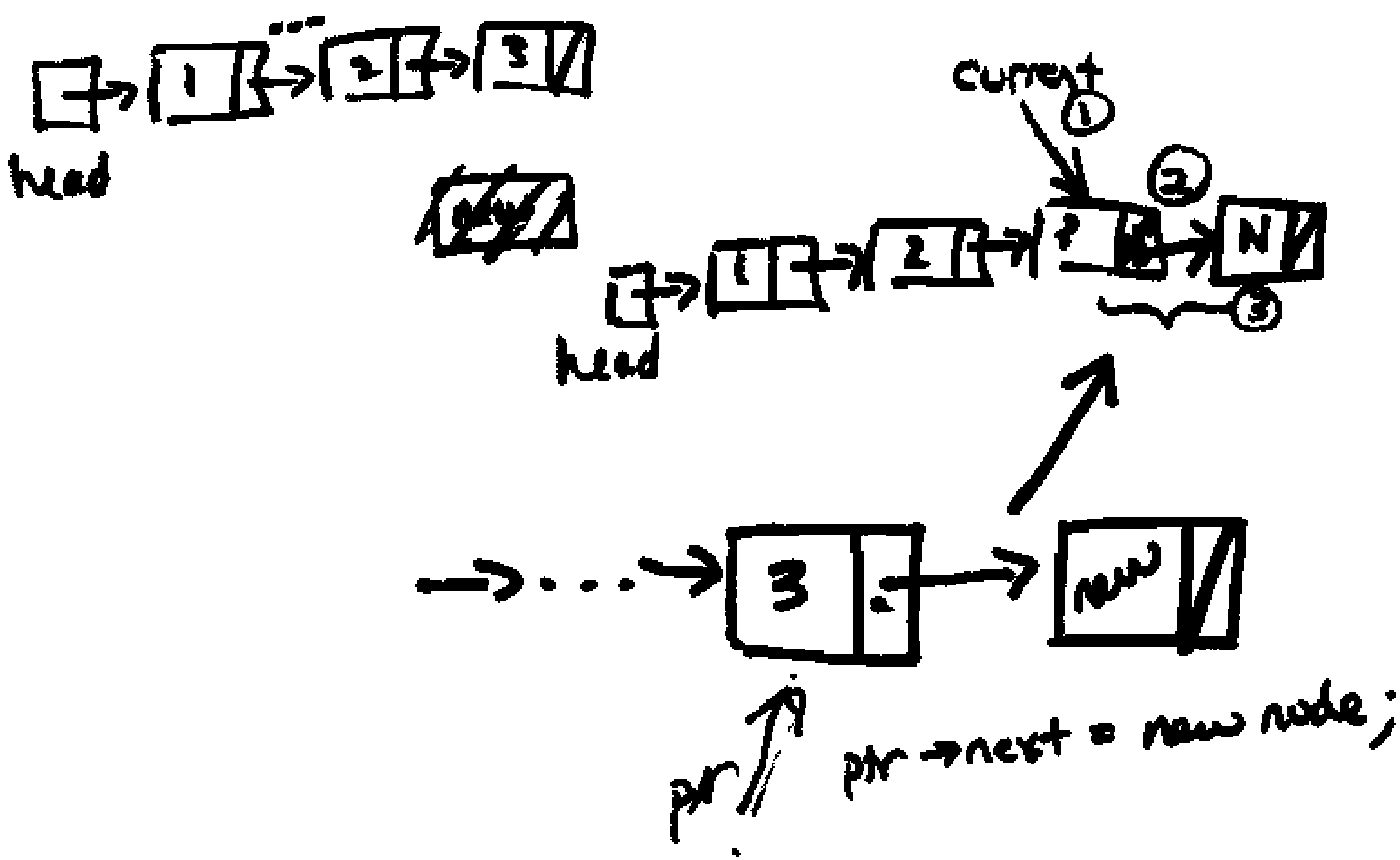
"Append"

Add at the end (NO tail ptr) ⑤

1.

Before | After

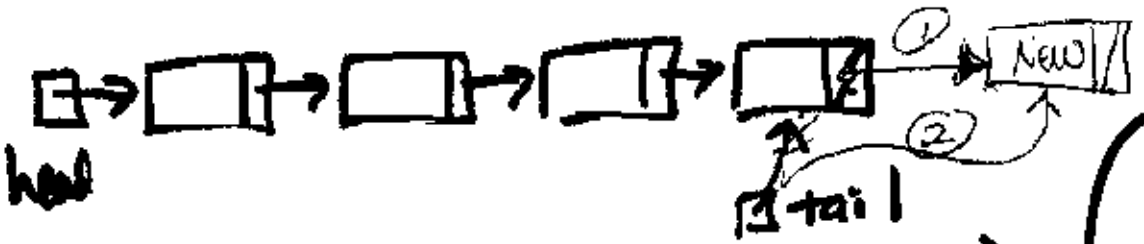
2.



```
if (!head)   // Case 1.   //if(head == NULL)  (6)
{   head = new node;
    head -> movie.read();
    head -> next = NULL;
}
else   // case 2.
{
(1) { node *current = head;
      while ( current->next ) //while current->next != NULL
      {     current = current -> next;
      }
    }
(2)   current->next = new node;
      (*curent).next
(3) { current = current->next; // Traversing to the new node
      current -> movie.read();
      current -> next = NULL;
    }
}
```

Insert at end <u>with</u> a tail pointr

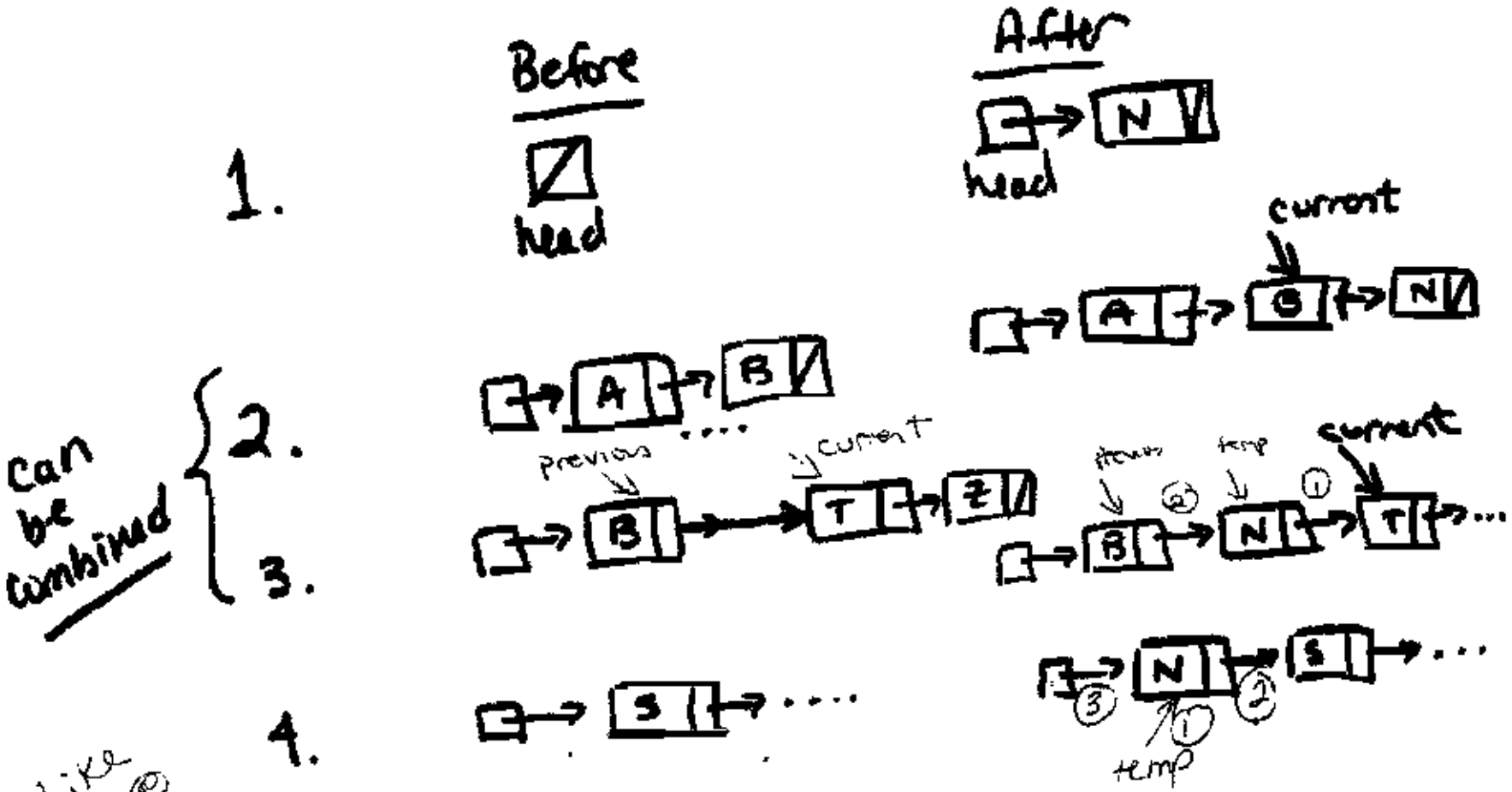
(12)




```
if (tail) //if(tail != NULL)
{
        tail->next = new node;      ①
        tail = tail->next;          // tail = temp;  ②
        //save data
        tail->next = NULL;
}
```

temp

Insert in Sorted Order



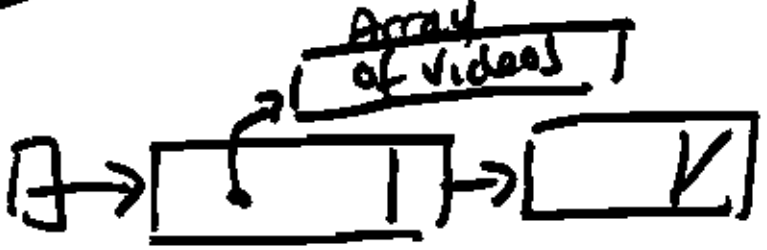
IF

Alternatively

```
Struct node
{ video * movie;
  node * next;
};
```

```
temp = new node;
temp -> move -> read();
```

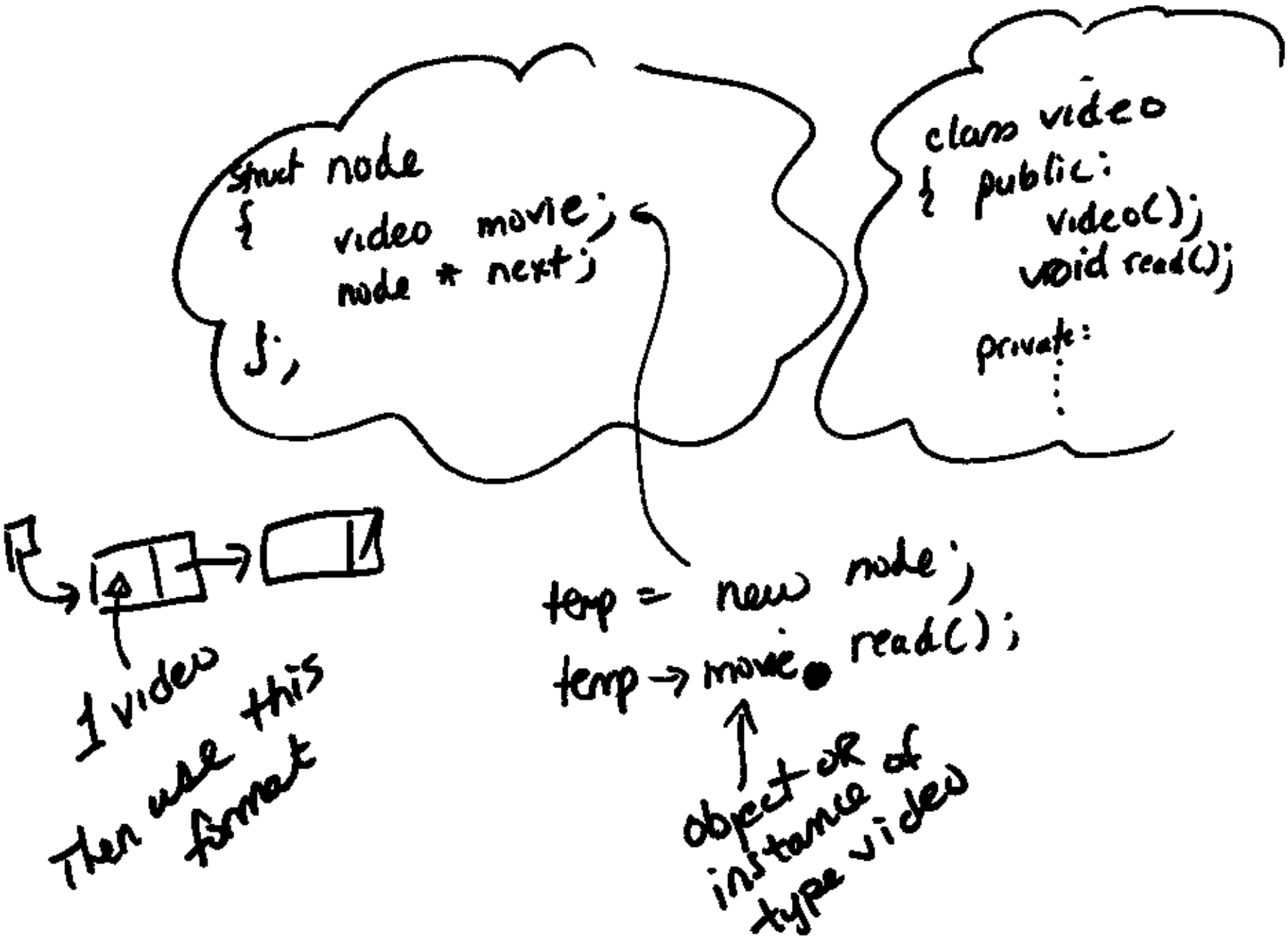
↑ is a pointer to a video



①

② if memory for the video is also used elsewhere....

T1
T2
Binary

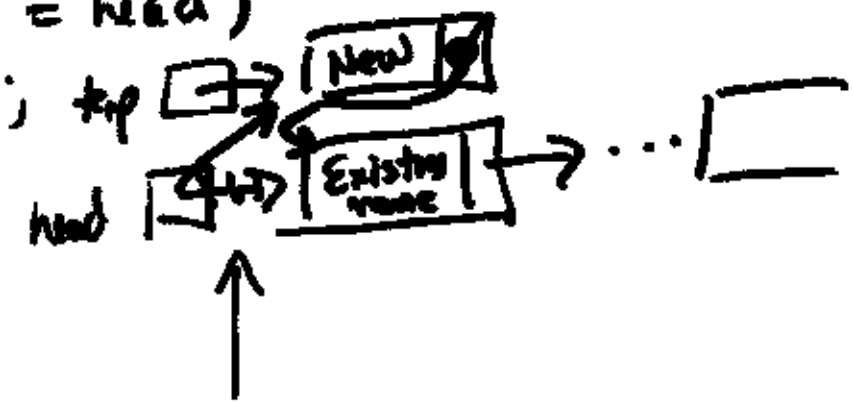

struct node
{
video movie;
node * next;
};
class video
{ public:
video();
void read();
private:
temp = new node;
temp → movie. read();
object or instance of type video
1 video
Then use this format

// sorted!

InsertCHead();

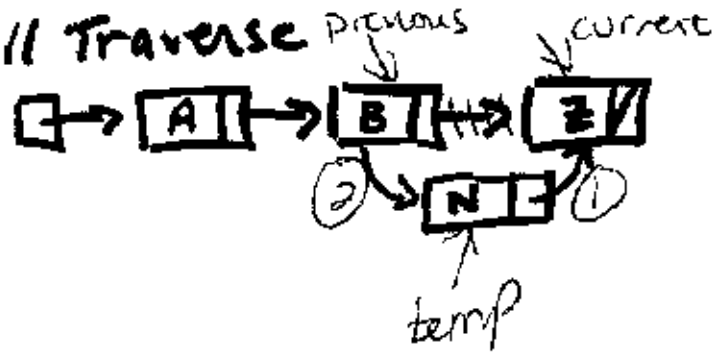
```
void Insert ( node *&head)
{
    node * temp = new node;  //Adding a node
    temp->movie.read();
    temp->next = NULL;

    // case 1
    if ( !head)  // EMPTY LIST
        head = temp;
    else // case 4
    if ( head->movie.compare(temp->movie) )
                // True if temp's movie is less head's movie
                // if (head->data > temp->data)
    {
        temp->next = head;
        head = temp;
    }
    else
```



```
(
if ( strcmp ( temp->movie.title, head->movie.title) < 0)
                //          <
```

{ // Traverse




```
// case 2, 3
node * current = head->next;
node * previous = head;
while( current && current->data < temp->data)
{
    previous = current;
    current = current->next;
}
// positioned correctly
(1) temp->next = current;
(2) previous->next = temp;
```

while (current != NULL && ...

```
{   // case 2, 3
    node * current = head->next;
    node * previous = head;
    while ( current &&
            temp->movie.compare(current->movie))
            // True if current's movie is less
            // than temp's movie
            // if ( temp's data > current's data)

    {
        previous = current;
        current = current->next;

    }

    temp->next = current;
    previous->next = temp;

}
```

Example

Add: Matrix
Add: Highlander
Add: Terminator
Add: Spiderman

Assume an Empty List to begin with

Example

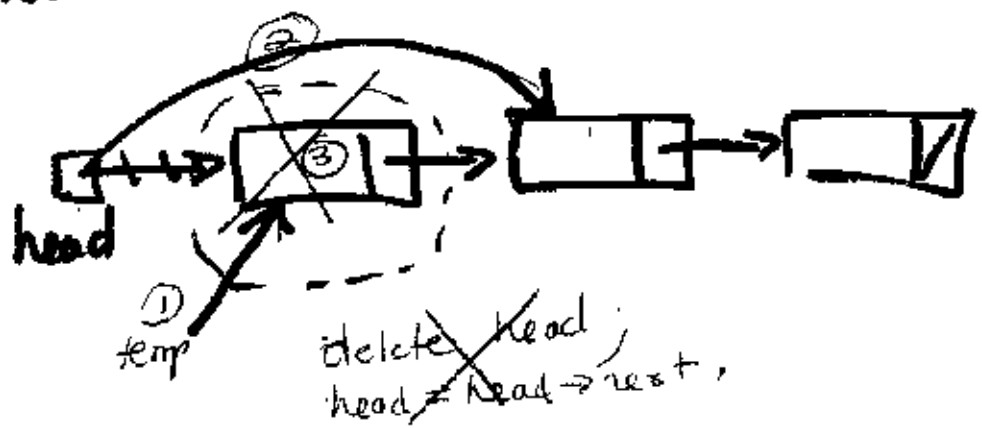
Add: matrix
Add: Highlander
Add: Terminator
Add: Serenity

Assume Empty List to start

Removal of Data from a LLL

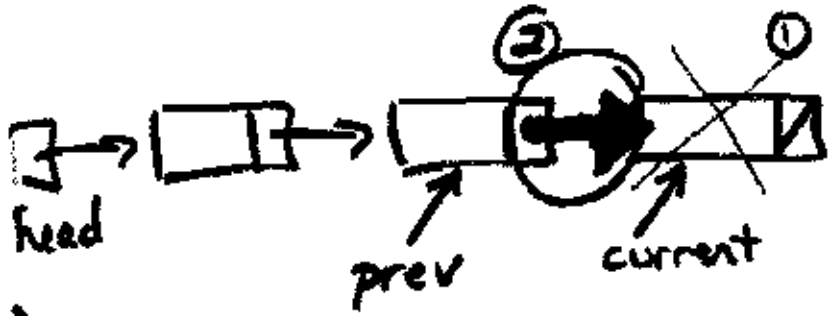
- Remove the first item
- Remove the last item
- Remove something in the middle
- Remove all

Remove first item




```
node * temp;
① temp = head;
② head = head->next;
③ delete temp;
```

Remove last item



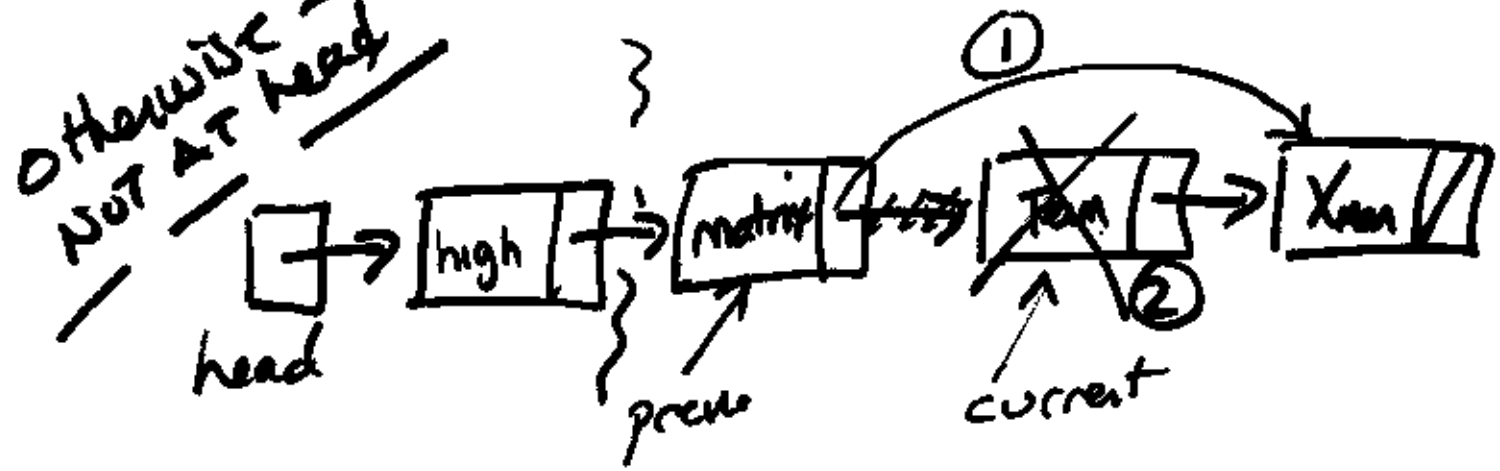

```
if(head)
{ current = head;
  previous = NULL;
   while (current->next)
   { previous = current;
     current = current->next;
   }
   delete current;
   previous->next = NULL;

}
```

```
delete current;   (1)
prev->next = NULL; (2)

if (!head->next)
{ // one node;
   delete head;
   head = NULL;

}
else
{
```

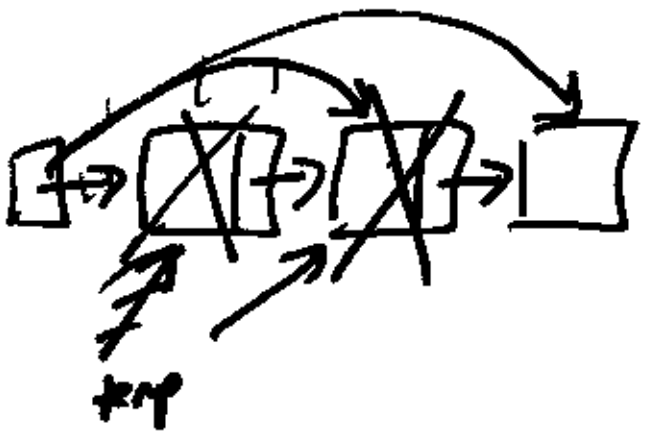



```
node * previous = head;
node * current = head->next;
while (current && current->data != match)
{
    previous = current;
    current = current->next;
}
if (current == NULL)
    cout << " NO Match Found ";
else
{
    previous->next = current->next;
    delete current;
}
```

if current IS Null There IS NO match

Remove All


```
node * temp;
while (head)
{     temp = head;
      head = head -> next;
      delete temp;
}
```



correct