



Types of Linked List - Part I

Course on C-Programming & Data Structures: GATE - 2024 & 2025

Data Structure: Linked List 2

By: Vishvadeep Gothi



Hello!

I am Vishvadeep Gothi

I am here because I love to teach

Traversing in Linked List

```
p = start  
while (p)  
{  
    process p → data  
    p = p → link  
}
```

Number of Elements in Linked List

Counting elements in a linked list is a common operation. There are several ways to approach this problem.

One way is to traverse the list and keep a count of the nodes visited. This requires traversing the entire list, resulting in a time complexity of $O(n)$, where n is the number of elements in the list.

Another way is to use a recursive approach. If the list is empty, the count is zero. Otherwise, the count is one plus the count of the rest of the list. This results in a time complexity of $O(n)$ for the same reason as the first approach.

A third way is to use an auxiliary data structure like an array or a stack to store the elements of the list. Once the list is stored, the count can be determined by simply counting the elements in the array or stack. This approach has a time complexity of $O(n)$ for the initial traversal of the list, but subsequent counts are $O(1)$.

The choice of approach depends on the specific requirements of the application. If the list is small and the count is needed only once, the recursive approach might be the most straightforward. If the list is large and the count is needed multiple times, using an auxiliary data structure might be more efficient. If the list is very large and memory is a concern, traversing the list might be the best option.

Address of Last Node in Linked List

```
p = start  
if (p == NULL) return NULL;  
while (p->link)  
{  
    —  
}  
}
```

Question

What would be the output after the following code segment is executed on a valid NULL terminated singly linked list.

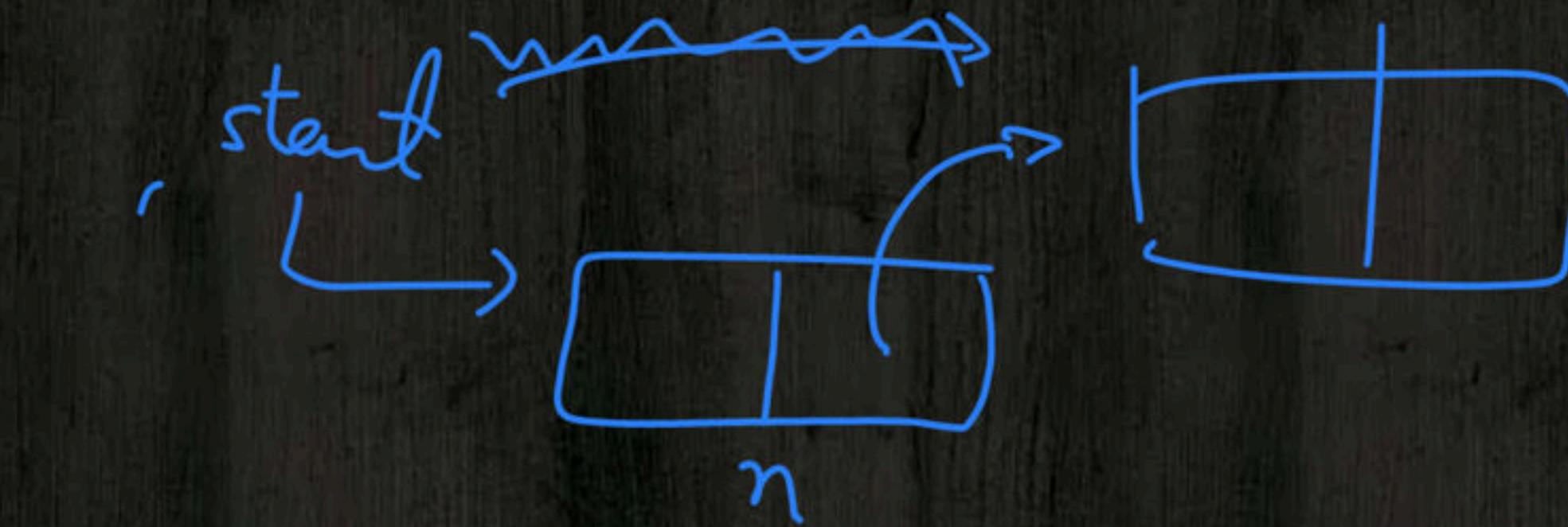
```
struct node *p;  
p = start ;  
while(p → link → link)  
{  
    p = p→ link;  
}  
printf("%c", p → data );
```

Insertion in Linked List

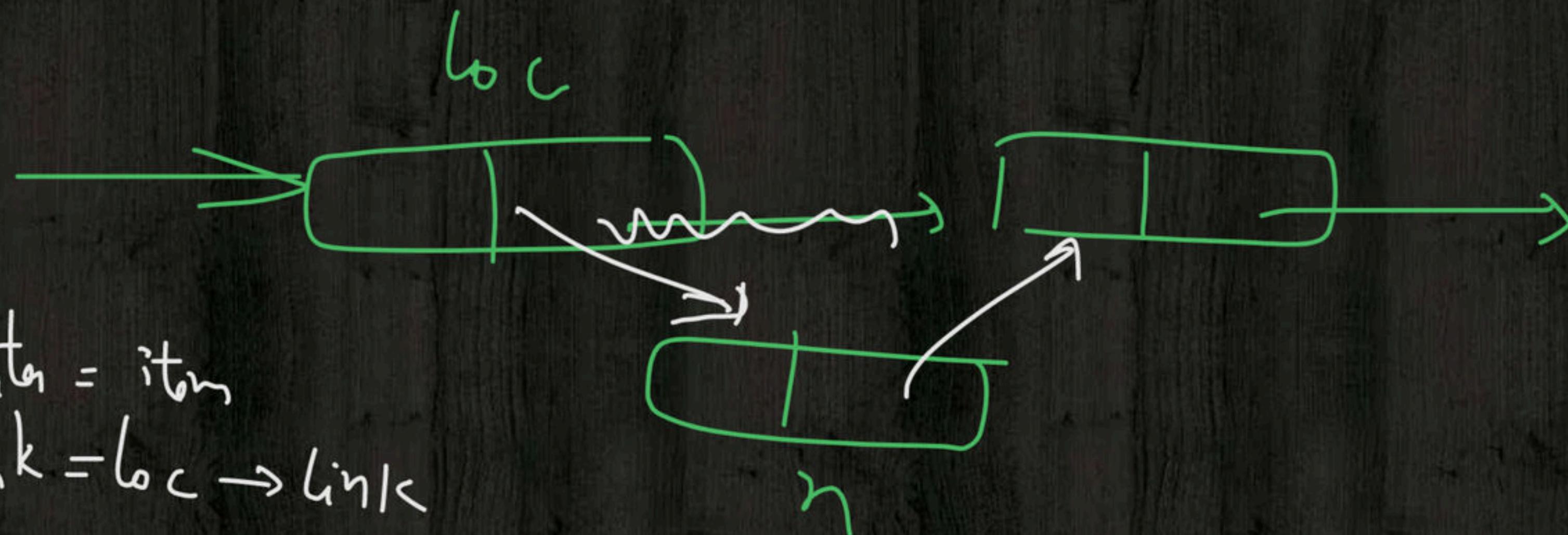
1. Insertion at beginning
2. Insertion after given node
3. Insertion at the end

Insertion at beginning

$n \rightarrow \text{data} = \text{item}$
 $n \rightarrow \text{link} = \text{start}$
 $\text{start} = n$



Insertion After Given Node



$n \rightarrow data = item$

$n \rightarrow link = loc \rightarrow link$

$loc \rightarrow link = n$

Insertion At the End

$p = start$

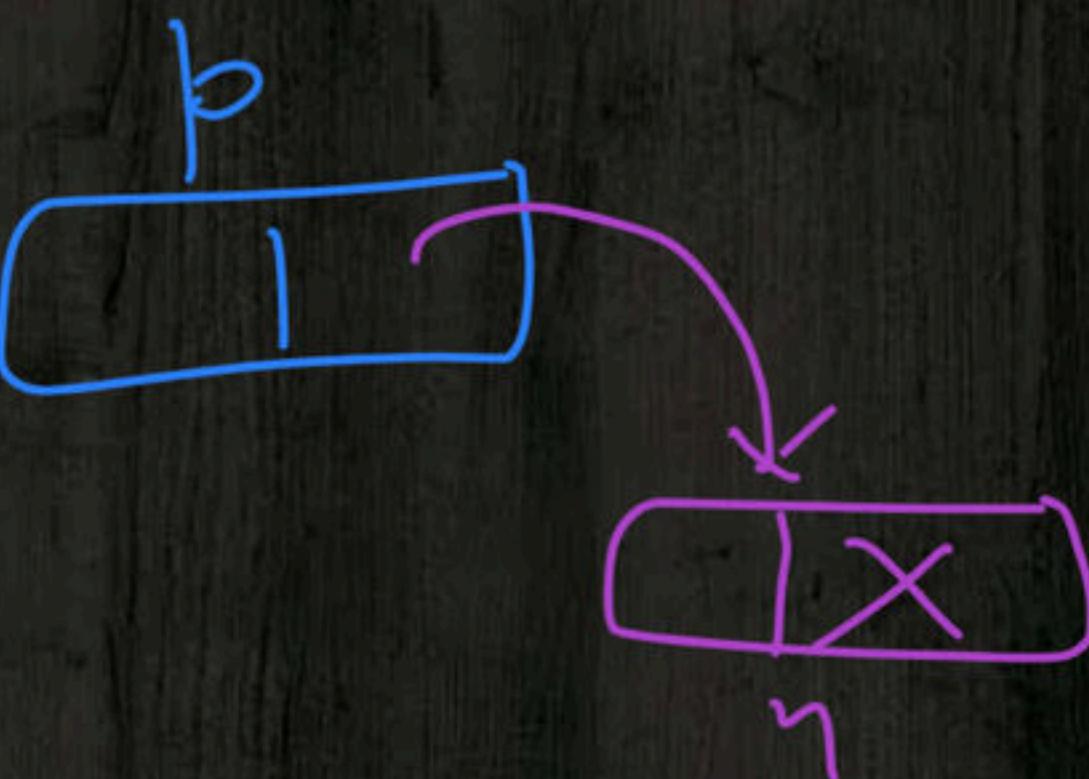
```
while ( $p \rightarrow link$ )  
{  
     $p = p \rightarrow link;$   
}
```

$h \rightarrow data = item$

$h \rightarrow link = NULL$

$p \rightarrow link = h$

$\Theta(n)$



if add of last node is given



Insertion at the end :-

$n \rightarrow \text{data} = \text{item}$

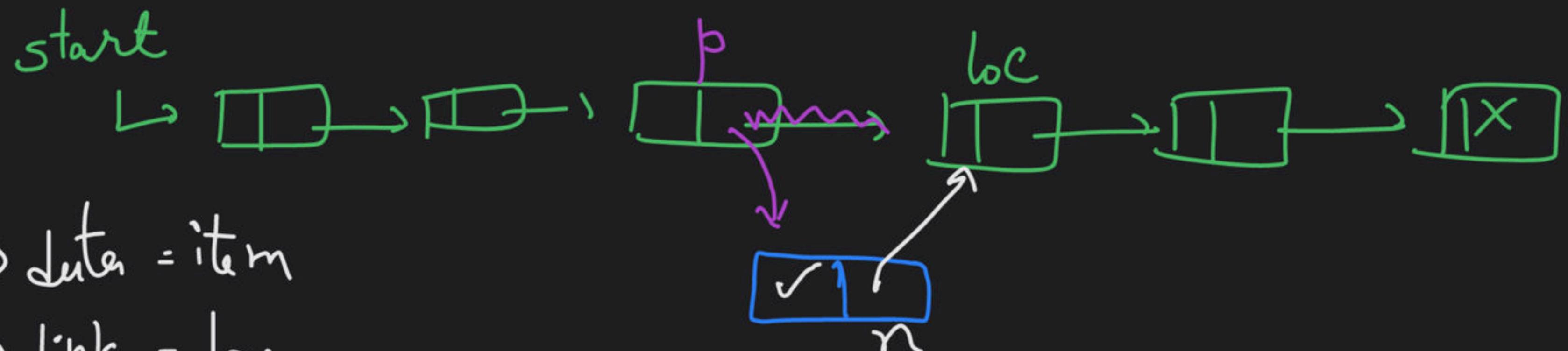
$n \rightarrow \text{link} = \text{NULL}$

$\Theta(1)$

$\text{last} \rightarrow \text{link} = n$

$\text{last} = n$

Insertion before a given node:-



$n \rightarrow \text{data} = \text{item}$

$n \rightarrow \text{link} = \text{loc}$

start node *p = start

while ($p \rightarrow \text{link} \neq \text{loc}$)

{
 $p = p \rightarrow \text{link}$

}

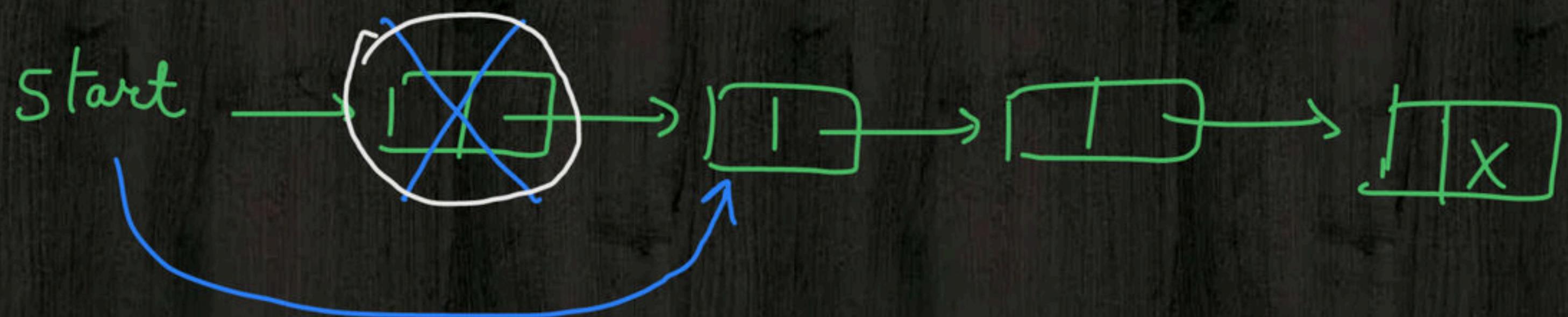
$p \rightarrow \text{link} = n$

R.T. Complexity = $O(n)$

Deletion in Linked List

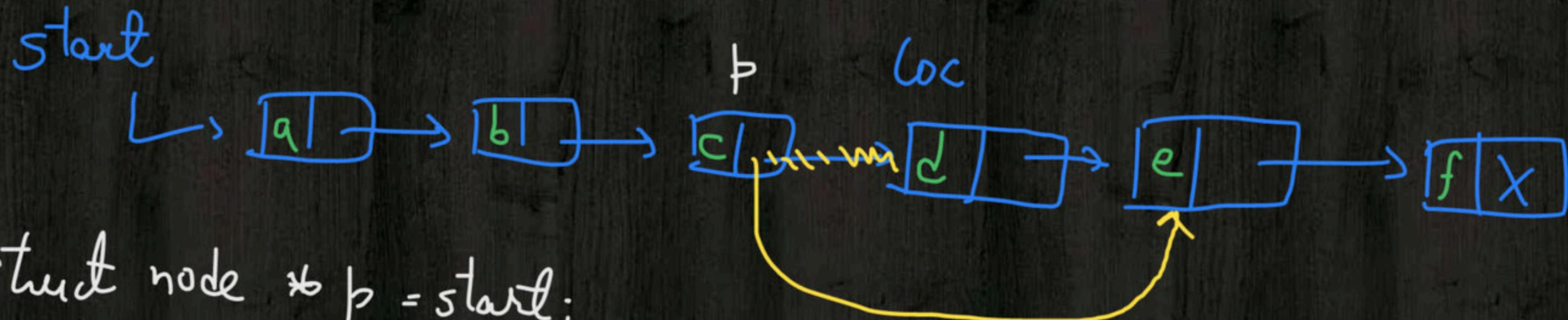
1. Deletion at beginning
2. Deletion of given node
3. Deletion at the end

Deletion At Beginning



```
struct node *p = start;           Θ(1)  
start = start->link;  
free(p);
```

Deletion Of a Given Node



start node $\& p = start;$

while ($p \rightarrow link \neq loc$)

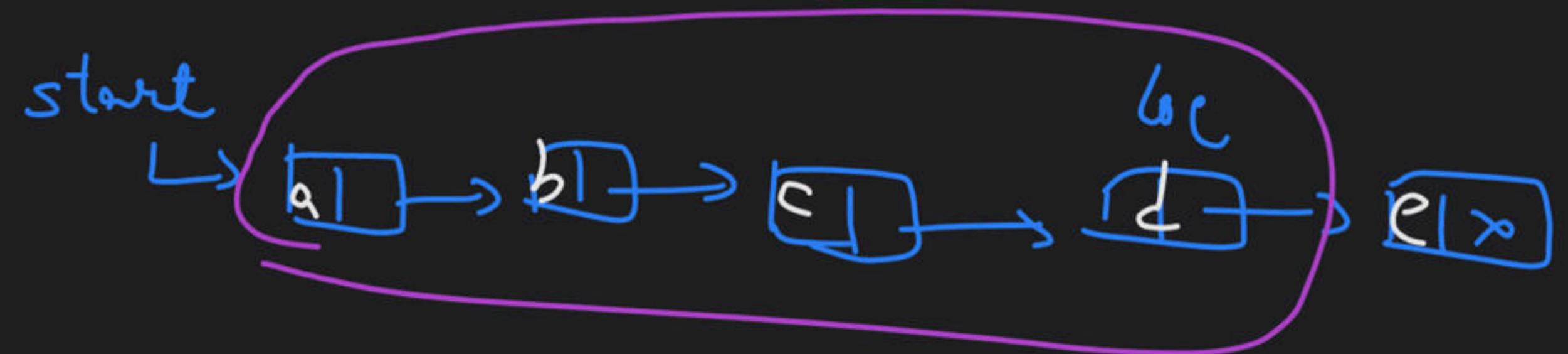
{
 $p = p \rightarrow link;$
}

$p \rightarrow link = loc \rightarrow link$

free(loc)

R. T. C. $\Rightarrow O(n)$

Invoked : ①

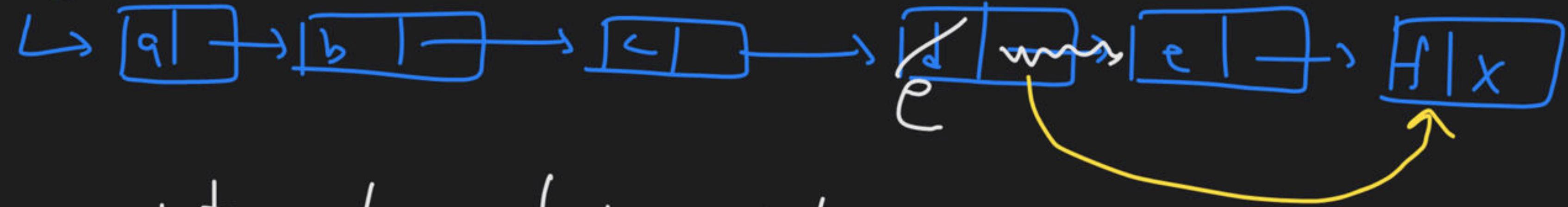


start = loc → link



Invalid 2 :-

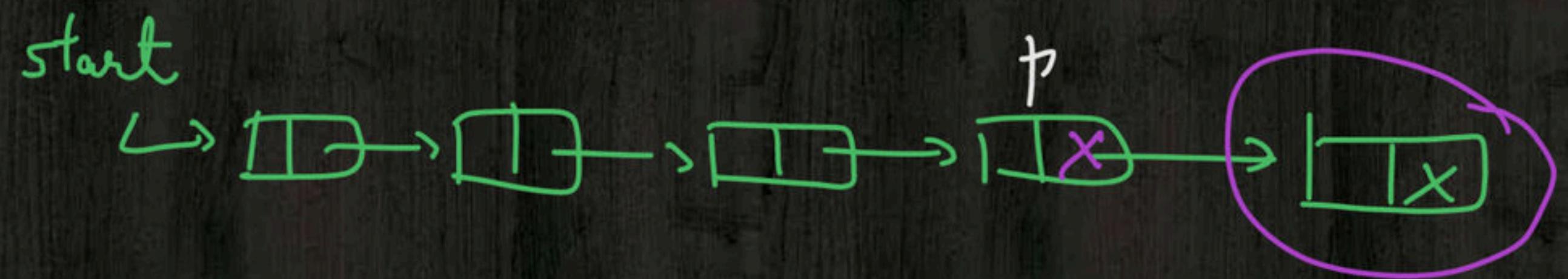
start



$loc \rightarrow \text{data} = loc \rightarrow \text{link} \rightarrow \text{data}$

$loc \rightarrow \text{link} = loc \rightarrow \text{link} \rightarrow \text{link}$

Deletion At the End



```
struct node *p = start;
```

```
while (p->link->link)
```

```
{
```

```
    p = p->link;
```

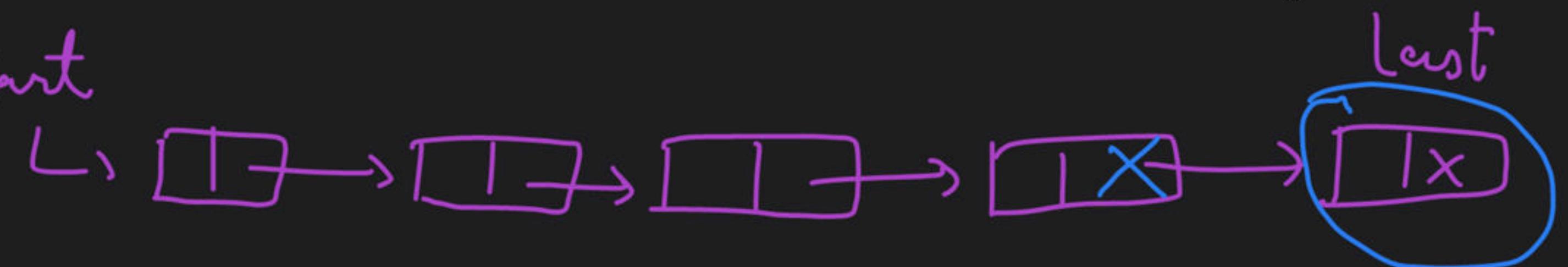
```
}
```

```
p->link = NULL;
```

R.T. Complexity = $\Theta(n)$

Deletⁿ of last node, when address of last node is given:-

start



R. I. Complexity = $\Theta(n)$

Searching in linked-list :-

Successful search \Rightarrow return add. of node where element found

Unsuccessful search \Rightarrow return NULL

linear search in LL :-

struct node *p = start;

while (p)

{

if ($p \rightarrow \text{data} == \text{item}$)

return p;

p = p \rightarrow link;

}

return p or return NULL;

R.F. complexity = $O(n)$

Binary search in LL :-

start



B.S. in LL can not be as efficient as it is on array because in LL, reaching to mid node is not possible in constant time.

sets:-

$$S_1 = \{a, b, c, d\}$$

$$S_2 = \{c, d, e, f\}$$

Union

$$S_1 \cup S_2$$

$$\Rightarrow \{a, b, c, d, e, f\}$$

Intersection

$$S_1 \cap S_2$$

$$\{c, d\}$$

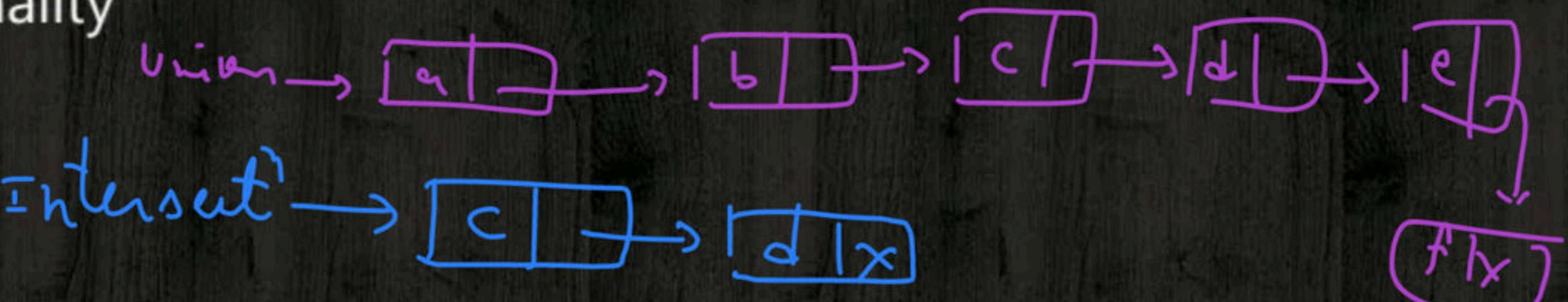
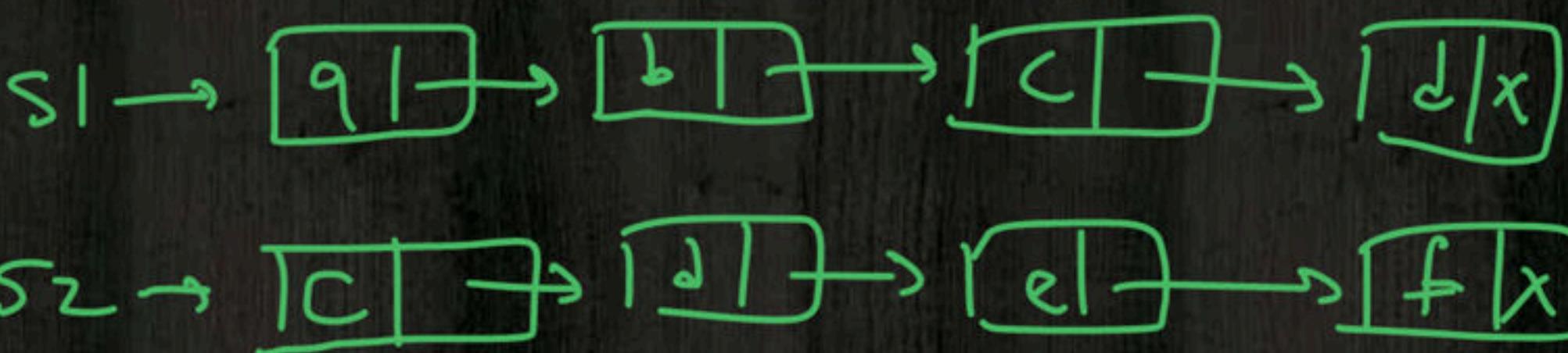
Question 1 GATE-2004

Suppose each set is represented as a linked list with elements in arbitrary order. Which of the operations among union, intersection, membership and cardinality will be the slowest?

- (A) Union only
- (B) Intersection, membership
- (C) Membership, cardinality
- (D) Union, intersection

$$S_1 = \{a, b, c, d\}$$

$$S_2 = \{c, d, e, f\}$$



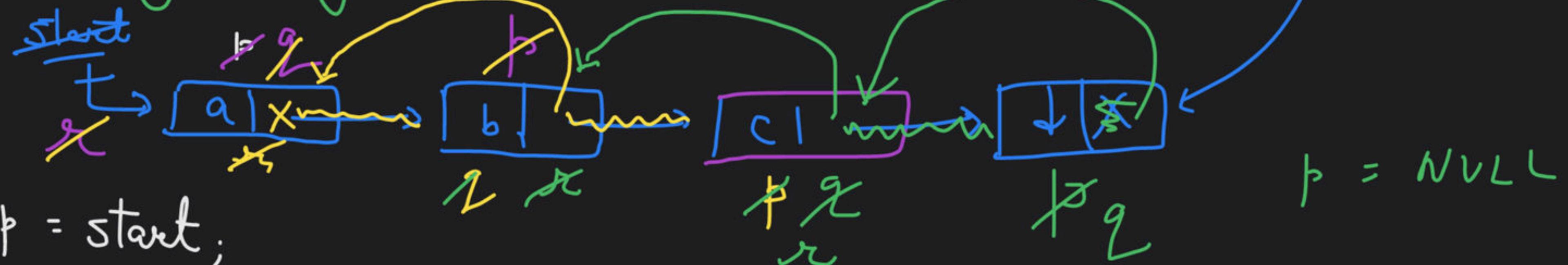
Membership \Rightarrow Searching $\Rightarrow \Theta(n)$

Cardinality \Rightarrow finding no. of elements $\Rightarrow \Theta(n)$

Union $\Rightarrow \Rightarrow \Theta(n^2)$

Intersection $\Rightarrow \Rightarrow \Theta(n^2)$

Reversing a given linked-list :-



r;

while (p)

{

r = q;

q = b;

b = p->link;

q->link = r; q = start = q;

R.T. Complexity = $\Theta(n)$

Space - II = $\Theta(1)$

Application of Linked List

- ◆ To store polynomials
 - Univariate
 - Bivariate

Disadvantage of Singly Linked List

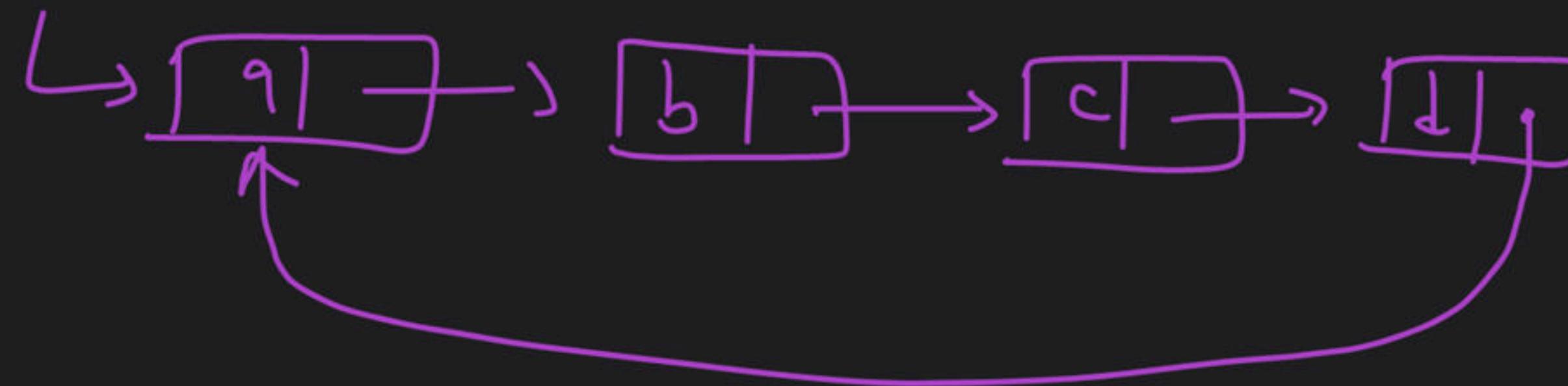
- ◆ The link part of last node is not utilized
- ◆ The address of predecessor is not known
- ◆ Stepping backward is not possible

→ circular list

→ doubly linked list

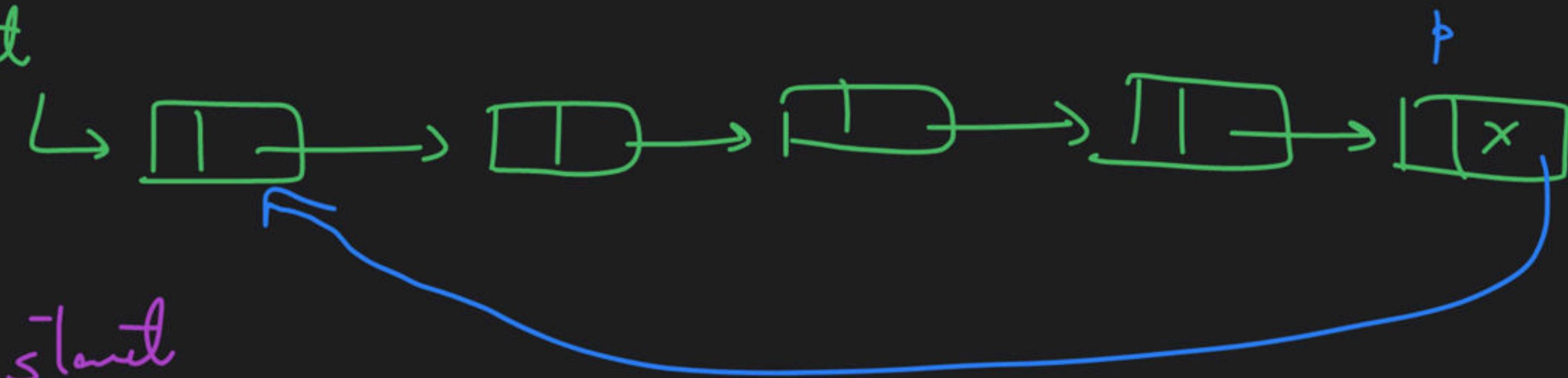
Circular List:-

Start



Convert a NULL terminated list into a circular list:-

Start



$p = \text{start}$

while ($p \rightarrow \text{link}$)

{
 $p = p \rightarrow \text{link}$;
}

$p \rightarrow \text{link} = \text{start}$;

R.T. Complexity = $\Theta(n)$

Header List

- ◆ Contains special first node called **Header node**.
- ◆ Header node contains some summery information.

Types of Header List

1. Grounded
2. Circular

Doubly Linked List

A doubly linked list is a linear data structure consisting of a sequence of nodes. Each node contains two pointers: one pointing to the previous node and another pointing to the next node.

The first node in the list is called the head, and the last node is called the tail. The head pointer is typically stored in a separate variable called the head pointer.

To insert a new node into the list, we need to update the pointers of the adjacent nodes. We can do this by setting the next pointer of the previous node to the new node and the previous pointer of the new node to the previous node.

To delete a node from the list, we need to update the pointers of the adjacent nodes. We can do this by setting the next pointer of the previous node to the next node and the previous pointer of the next node to the previous node.

Doubly linked lists are useful for implementing stacks and queues, as well as for performing bidirectional traversals of the list.

The time complexity for inserting or deleting a node in a singly linked list is O(n), where n is the number of nodes in the list. In contrast, the time complexity for inserting or deleting a node in a doubly linked list is O(1), as we only need to update the pointers of the adjacent nodes.

Doubly linked lists are also useful for implementing circular lists, where the last node's next pointer points back to the first node. This allows for bidirectional traversal of the list, as well as for efficiently finding the previous node in a single step.

In conclusion, doubly linked lists are a powerful data structure for implementing linear data structures with efficient insertion and deletion operations. They are particularly useful for implementing stacks, queues, and circular lists, as well as for performing bidirectional traversals of the list.

Insertion in Doubly Linked List

Deletion in Doubly Linked List

Question 1

- ◆ How many links updated when insertion and deletion done in doubly linked list?

Question 2

The following code is given to reverse the doubly linked list. Fill the blank?

```
void reverse(Dnode *head)
{
    struct Dnode *p = head;
    while(p != head)
    {
        swap(p → forw, p → back);


---


    }
    swap( head → forw, head → back)
}
```



DPP

Question 1

Given a singly linked list, check whether there is any loop in the list or not?

Question 2

Find the intersection point of a Y-shaped List?

Question 3

Write an algorithm to reverse a singly linked list?



Happy Learning