
# PROGRAMMING AND DATA STRUCTURES

## Traversing the List (Linked List Vs Arrays)

# Linked List

## Counting the elements

```
void count_of_nodes(struct node *head) {
    int count = 0;
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        count++;
        ptr = ptr->link;
    }
    printf("%d", count);
}
```

Time complexity: O(n)

## Printing the Data

```
void print_data(struct node *head) {
    if(head == NULL)
        printf("Linked List is empty");
    struct node *ptr = NULL;
    ptr = head;
    while(ptr != NULL) {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }
}
```

Time complexity: O(n)

# Array

## Counting the elements

```
#include <stdio.h>

int main() {
    int arr[] = {45, 98, 3};
    int n;
    n = sizeof(arr)/sizeof(int);
    printf("%d", n);
    return 0;
}
```

Time complexity: O(1)

## Printing the Data

```
#include <stdio.h>

int main() {
    int arr[] = {45, 98, 3};
    int n, i;
    n = sizeof(arr)/sizeof(int);
    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Time complexity: O(n)

# Summary

## Linked List

Counting the elements: O(n)

Printing the data: O(n)

## Array

Counting the elements: O(1)

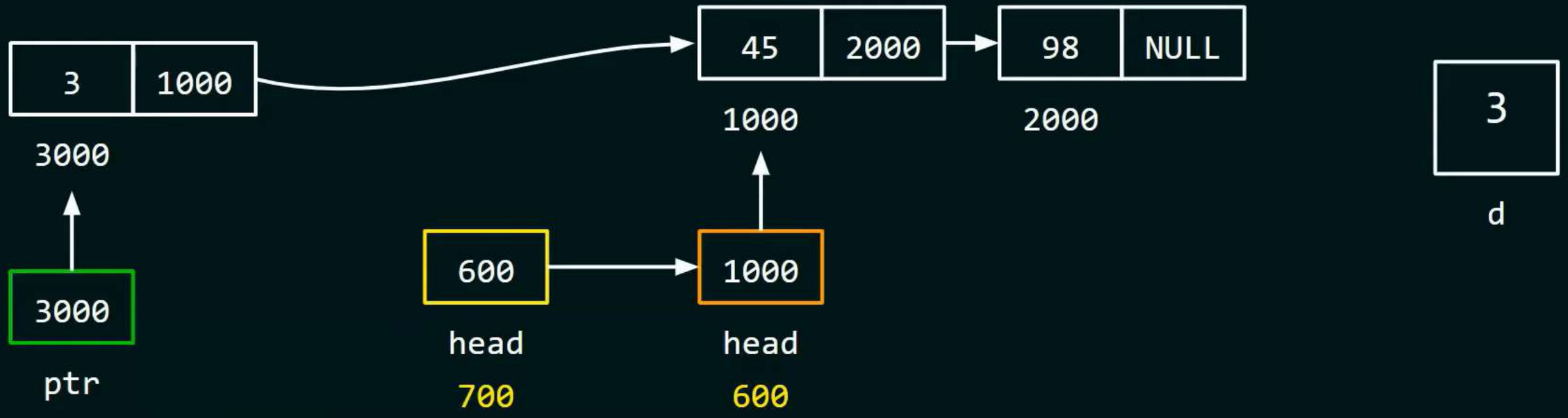
Printing the data: O(n)


# Programming and Data Structures

## Inserting Data at the Beginning of the List 2.0

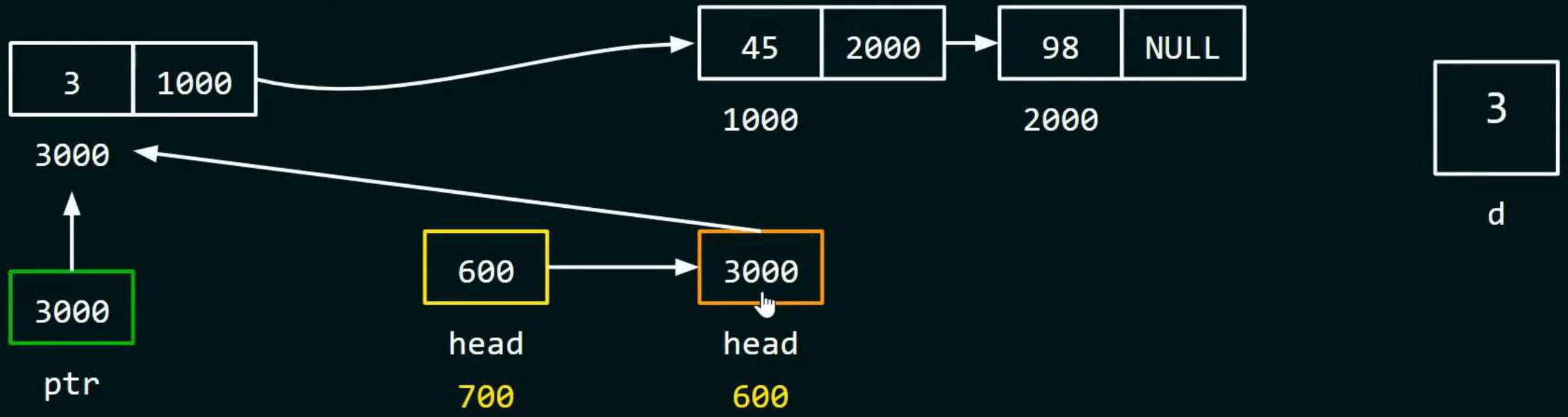
# Adding the node at the beginning of the list

```
void add_beg(struct node **head, int d)
{
    struct node *ptr = malloc(sizeof(struct node));
    ptr->data = d;
    ptr->link = NULL;

    ptr->link = *head;
    *head = ptr;
}
```



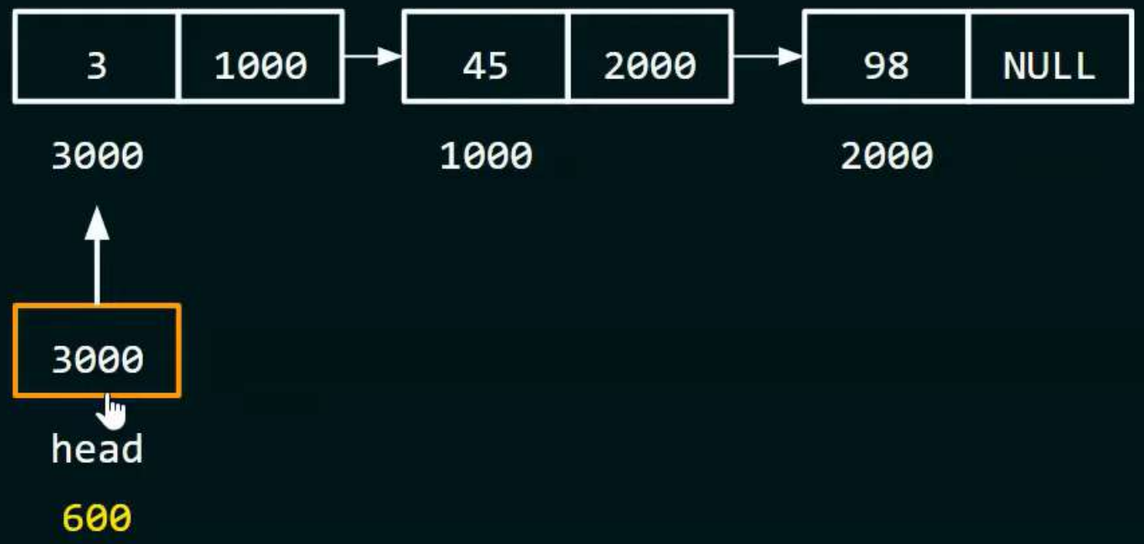
# Adding the node at the beginning of the list

```
void add_beg(struct node **head, int d)
{
    struct node *ptr = malloc(sizeof(struct node));
    ptr->data = d;
    ptr->link = NULL;

    ptr->link = *head;
    *head = ptr;
}
```



# Program – Main Function

```c
#include <stdio.h>
#include <stdlib.h>

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




```c
int main() {
    struct node *head = malloc(sizeof(struct node));
    head->data = 45;
    head->link = NULL;

    struct node *ptr = malloc(sizeof(struct node));
    ptr->data = 98;
    ptr->link = NULL;

    head->link = ptr;

    int data = 3;

    add_beg(&head, data);
    ptr = head;
    while(ptr != NULL)
    {
        printf("%d ", ptr->data);
        ptr = ptr->link;
    }

     return 0;
}
```


# Programming and Data Structures

## Inserting Data at the Beginning (Linked List vs Array)

VERSION 1.0

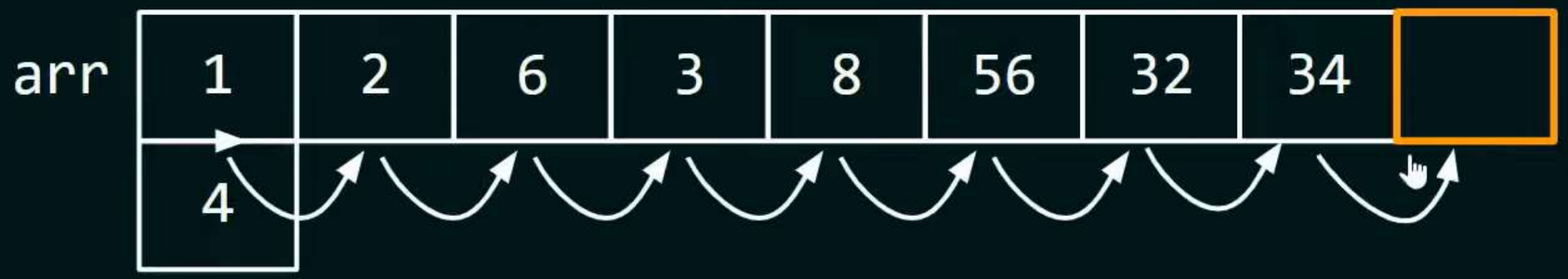
```c
struct node* add_beg(struct node* head, int d)
{
    struct node *ptr = malloc(sizeof(struct node));
    ptr->data = d;
    ptr->link = NULL;

    ptr->link = head;
    head = ptr;
    return head;
}
```

VERSION 2.0

```c
void add_beg(struct node **head, int d)
{
    struct node *ptr = malloc(sizeof(struct node));
    ptr->data = d;
    ptr->link = NULL;

    ptr->link = *head;
    *head = ptr;
}
```

TIME COMPLEXITY: O(1)

# Example



# Example


If there are n elements in the array, then n shifts are required.

Time complexity: O(n)

# Program

```
int main() {
    int arr[10], data = 10, i, n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);

    n = add_beg(arr, n, data);

    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Let say, a user has entered the value of n as 4

And, the values entered by the user are: 1, 2, 3, 4


# Add at the beginning of the array

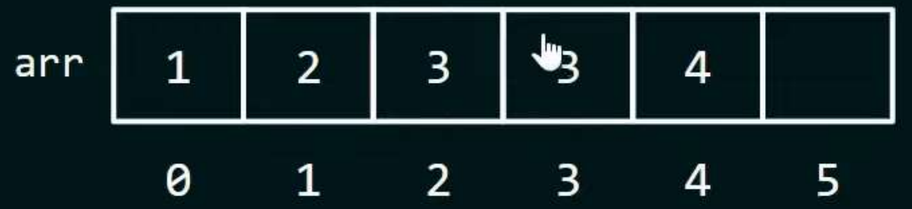
```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[i+1] = arr[i];
    }
    arr[0] = data;
    return n+1;
}
```


# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[4] = arr[3];
    }
    arr[0] = data;
    return n+1;
}
```


# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[3] = arr[2];
    }
    arr[0] = data;
    return n+1;
}
```

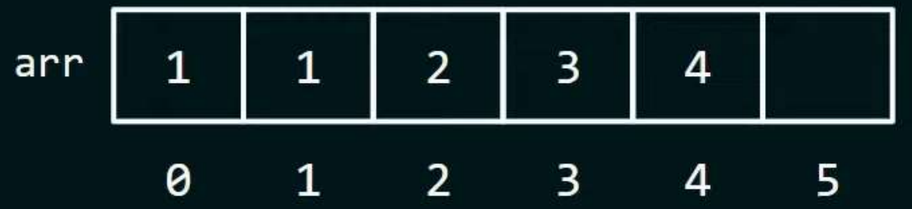



# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[1] = arr[0];
    }
    arr[0] = data;
    return n+1;
}
```


# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[i+1] = arr[i];
    }
    arr[0] = data;
    return n+1;
}
```

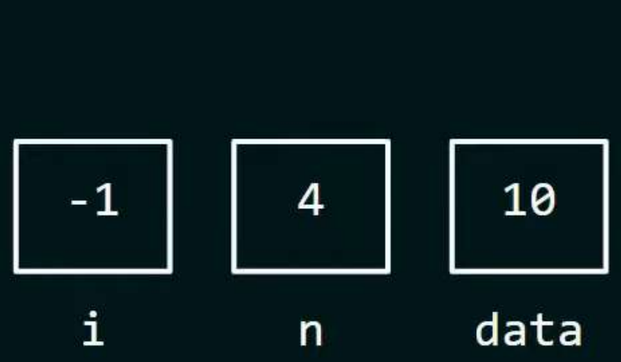



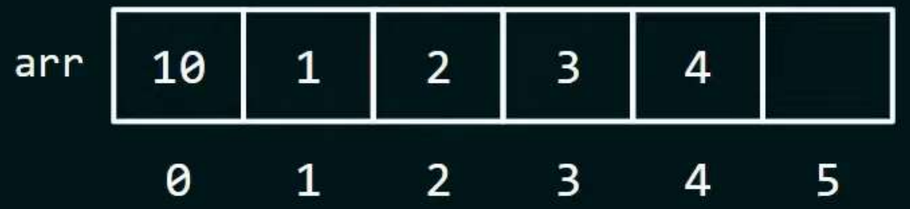
# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[i+1] = arr[i];
    }
    arr[0] = data;
    return n+1;
}
```


# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[i+1] = arr[i];
    }
    arr[0] = data;
    return 5;
}
```



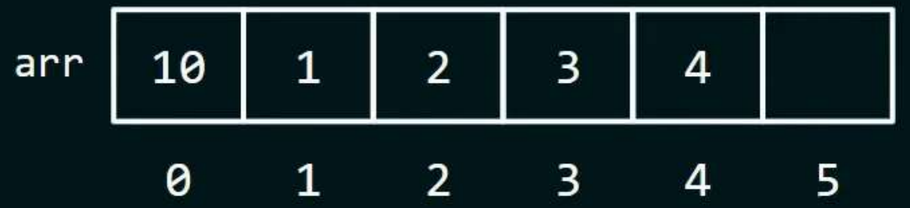



# Program

```
int main() {
    int arr[10], data = 10, i, n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);

    n = add_beg(arr, n, data);

    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Let say, a user has entered the value of n as 4

And, the values entered by the user are: 1, 2, 3, 4

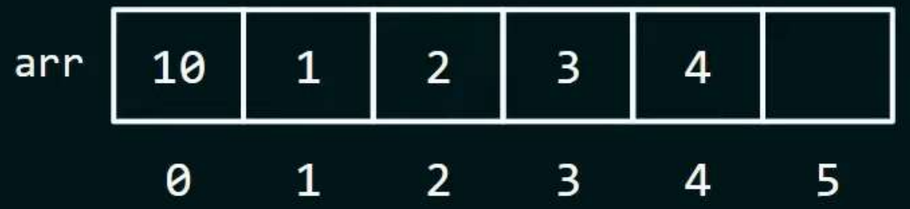



# Program

```
int main() {
    int arr[10], data = 10, i, n;
    printf("Enter the number of elements: ");
    scanf("%d", &n);
    printf("Enter the elements: ");
    for(i=0; i<n; i++)
        scanf("%d", &arr[i]);

    n = add_beg(arr, n, data);

    for(i=0; i<n; i++)
        printf("%d ", arr[i]);
    return 0;
}
```

Let say, a user has entered the value of n as 4

And, the values entered by the user are: 1, 2, 3, 4




Output: 10 1 2 3 4

# Add at the beginning of the array

```
int add_beg(int arr[], int n, int data)
{
    int i;
    for(i=n-1; i>=0; i--)
    {
        arr[i+1] = arr[i];
    }
    arr[0] = data;
    return n+1;
}
```

Considering that the array is not full.

Time complexity: O(n)

# When array is full

Copying the old array to the new array of size greater than the old array: O(n)

Adding at the beginning of the array: O(n)

Total time: n + n = 2n = O(n)

**Time complexity: O(n)**

# When array is full

Copying the old array to the new array of size greater than the old array from index 1: O(n)

Adding at the beginning of the array: O(1)

Total time: n + 1 = O(n)

**Time complexity: O(n)**

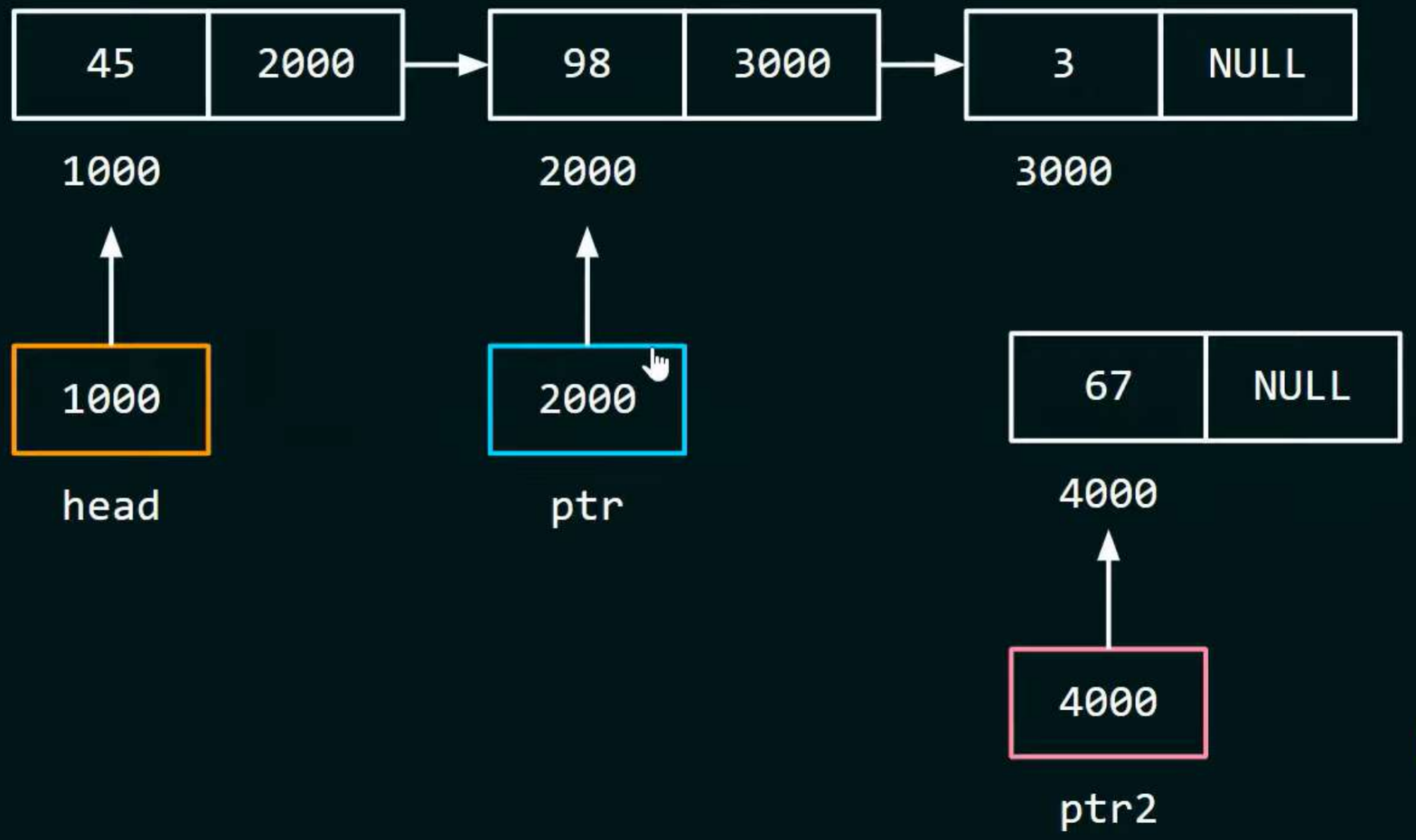
# Summary

| Linked List | | Array | |
|---|---|---|---|
| Add at beginning: | O(1) | Add at beginning: | O(n) |


# PROGRAMMING AND DATA STRUCTURES

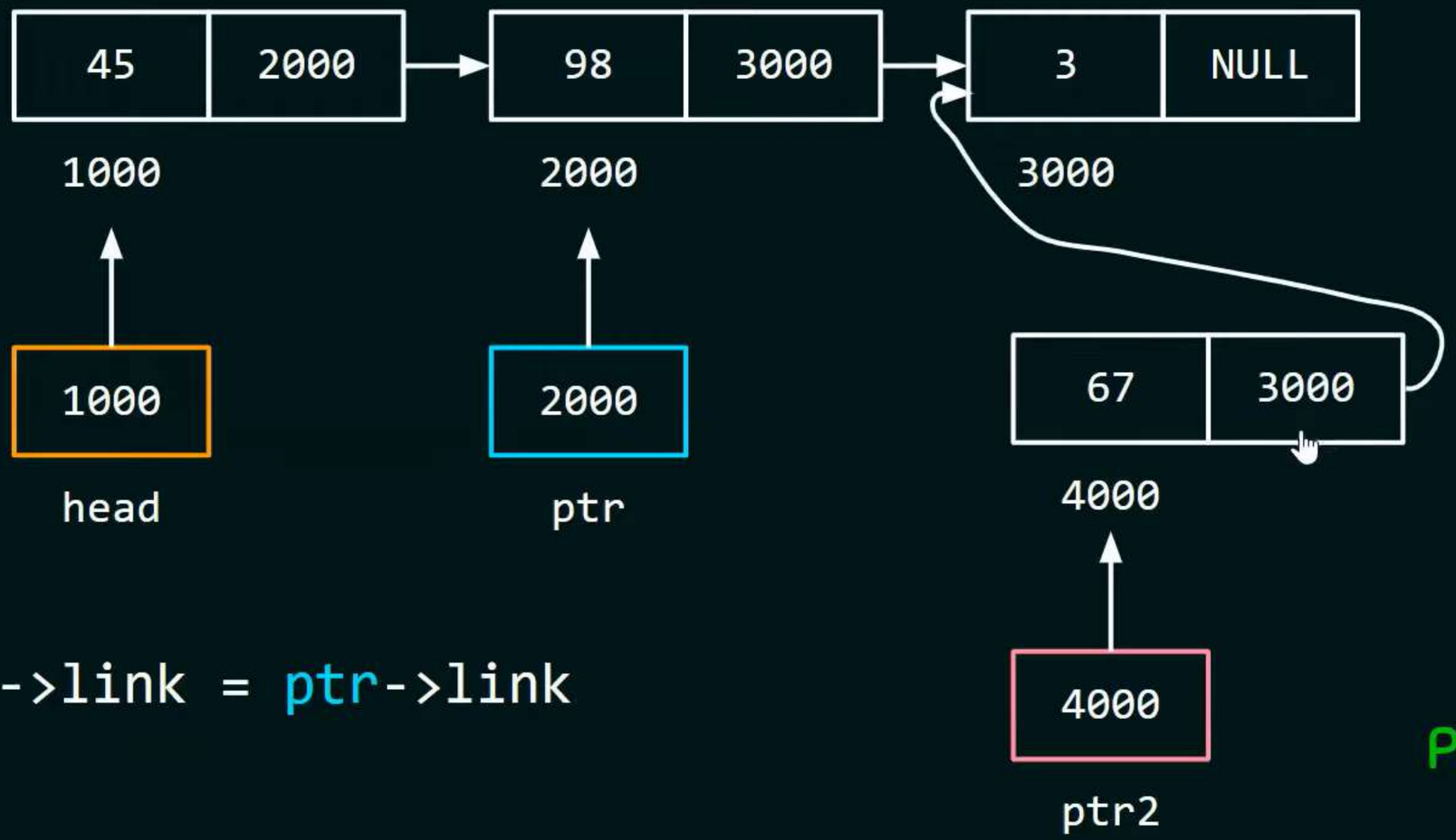
## Inserting a Node at Certain Position

**Assumption:** Position where the new node needs to be inserted is given.




Position: 3

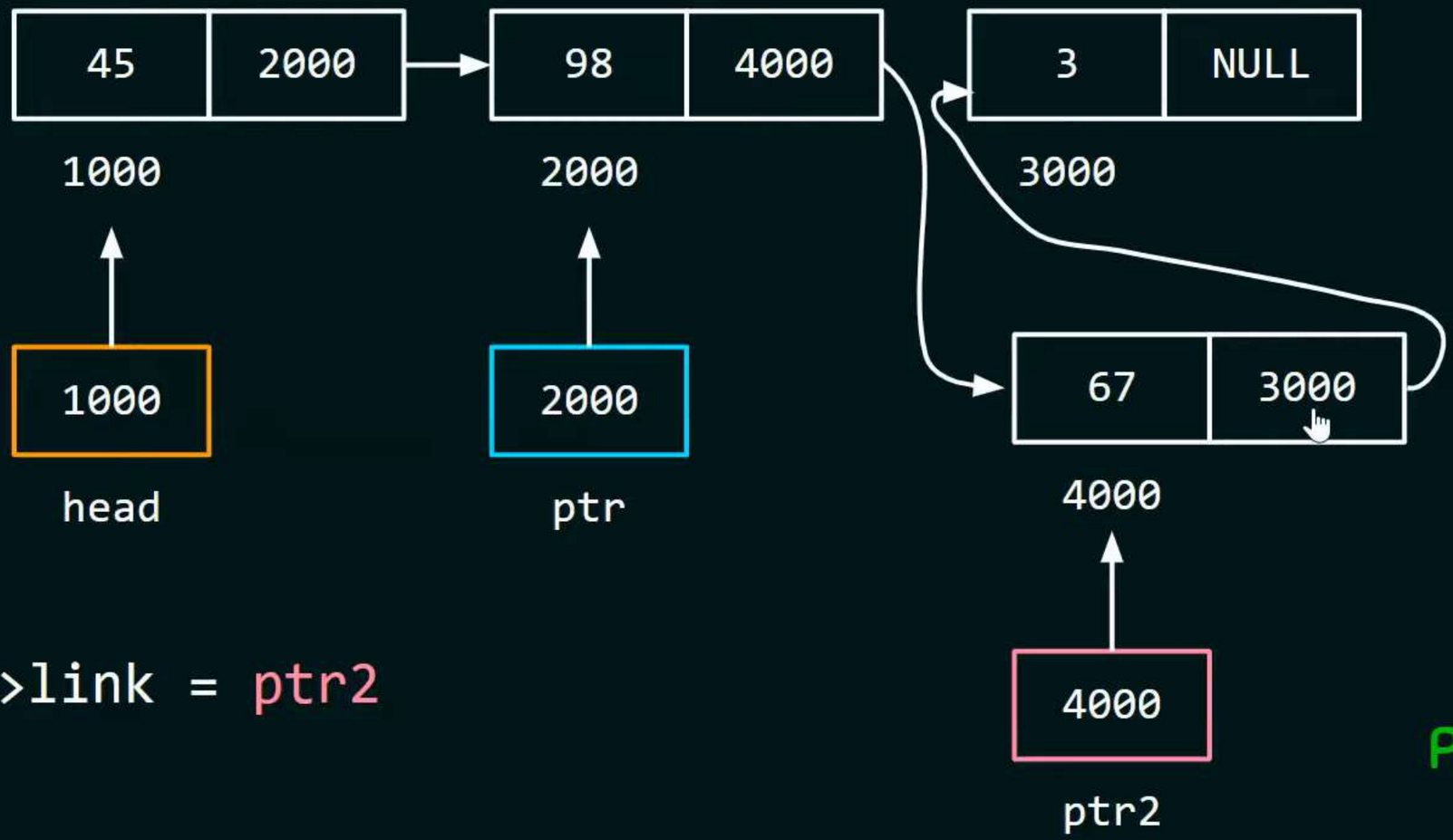
**Assumption**: Position where the new node needs to be inserted is given.




```
ptr2->link = ptr->link
```

**Position**: 3

**Assumption**: Position where the new node needs to be inserted is given.

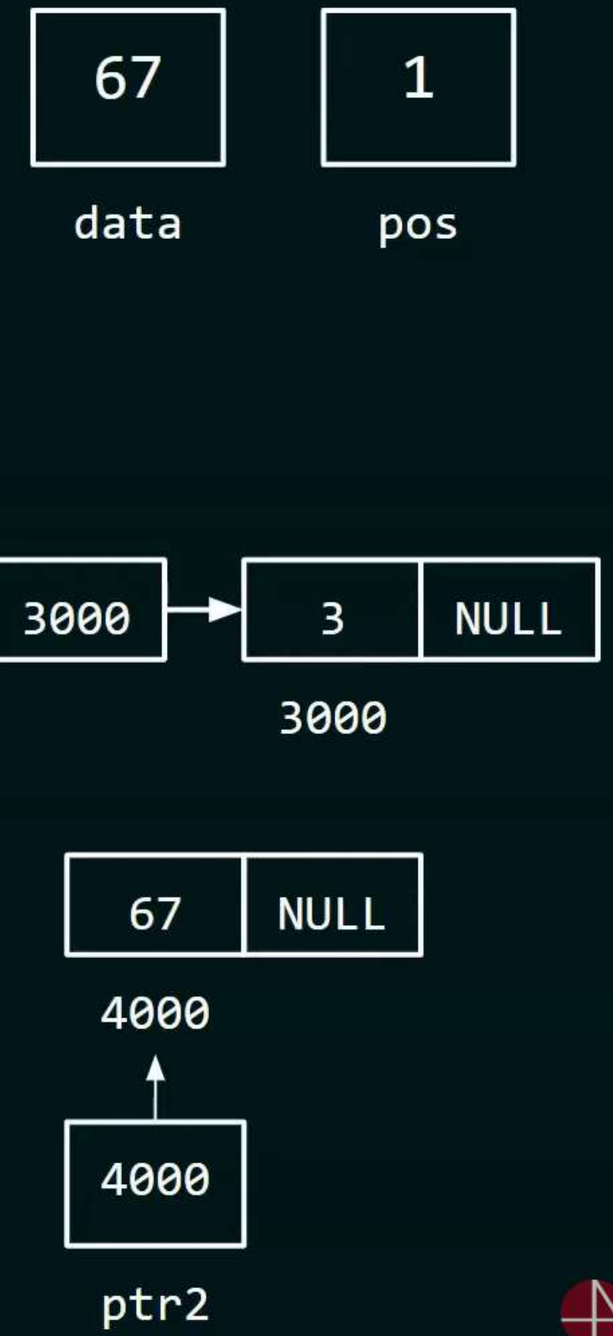



```
ptr->link = ptr2
```

**Position**: 3

# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

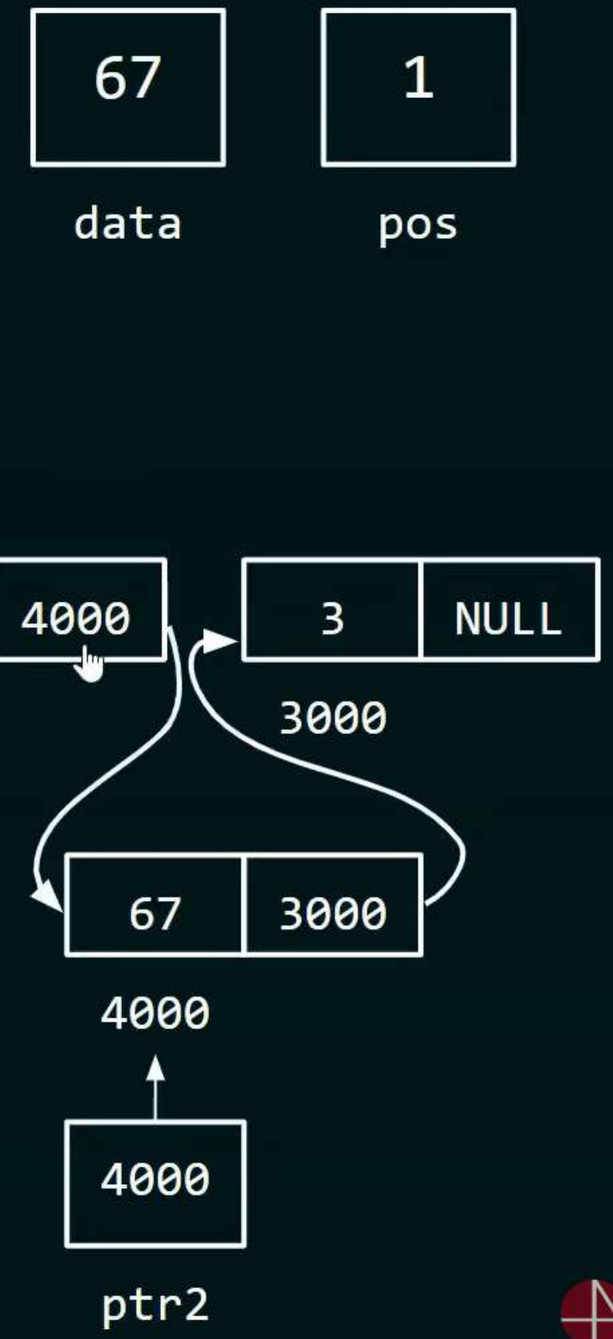


# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```




# PROGRAMMING AND DATA STRUCTURES

## Insertion at certain position (Linked List vs Array) (Part 1)

# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

Suppose, our target is to add the element at the end of the list.

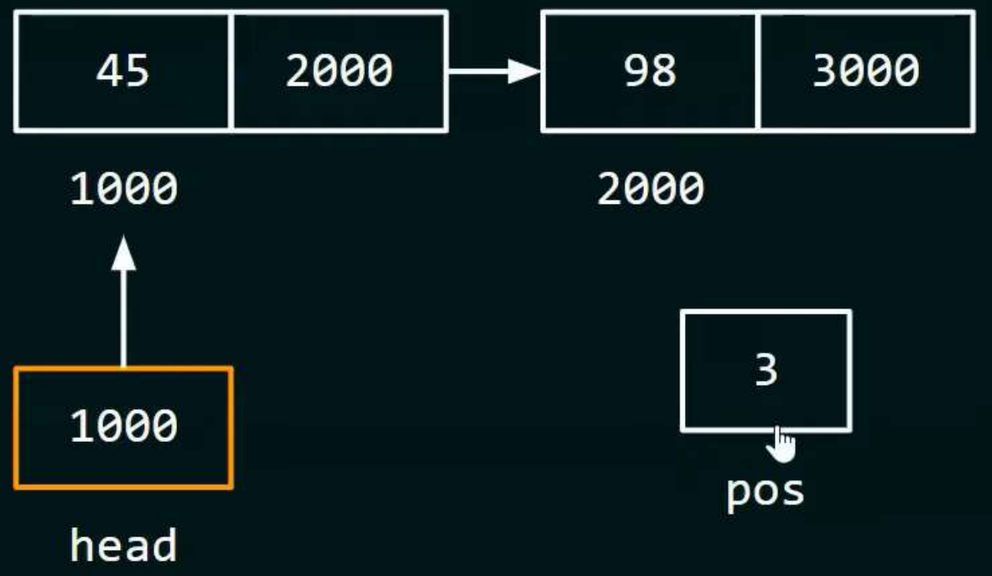
# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

If there 2 nodes in the list and we have to add a new node at 3rd position then while loop will run only 1 time.

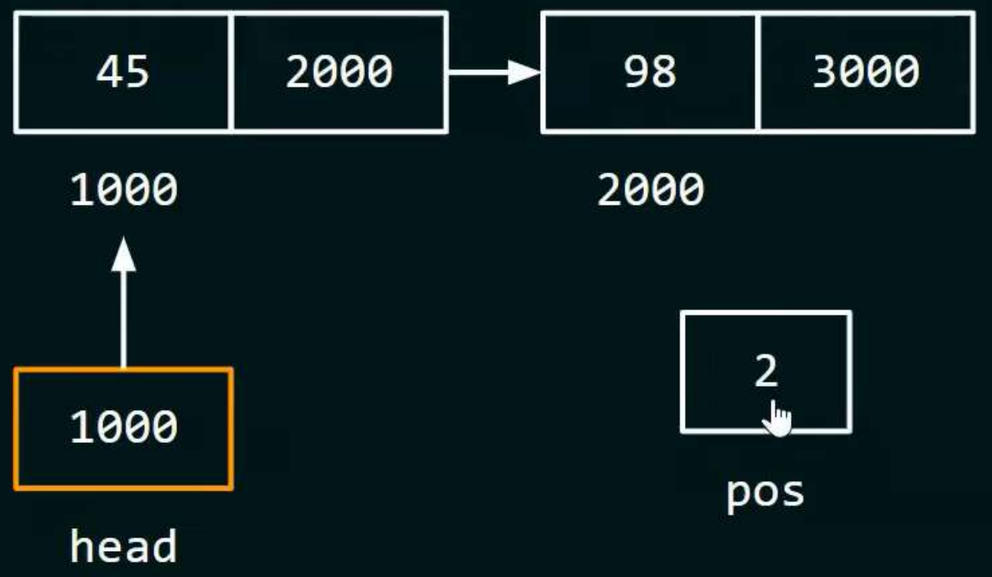
# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```



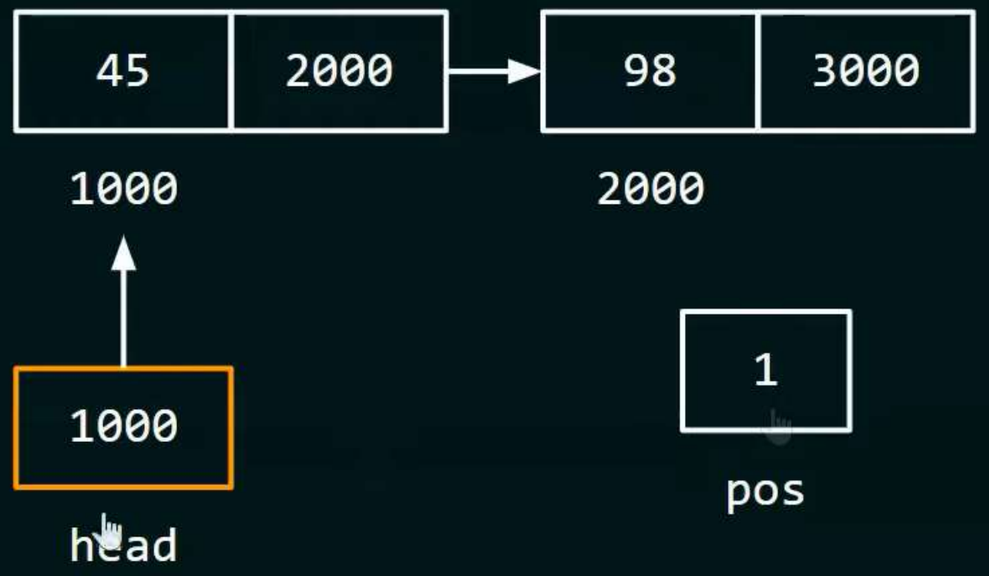
# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```



# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```



# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

If there 3 nodes in the list and we have to add a new node at 4th position then while loop will run only 2 times.

# Add at position function

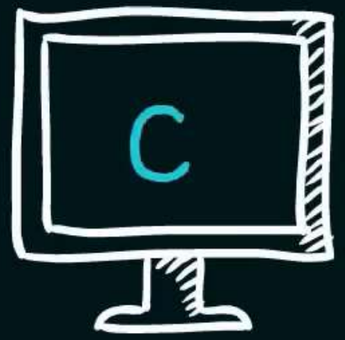
```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

If there n nodes in the list and we have to add a new node at (n+1)th position then while loop will run only (n-1) times.

# Add at position function

```
void add_at_pos(struct node* head, int data, int pos)
{
    struct node *ptr = head;
    struct node *ptr2 = malloc(sizeof(struct node));
    ptr2->data = data;
    ptr2->link = NULL;

    pos--;
    while(pos != 1)
    {
        ptr = ptr->link;
        pos--;

    }
    ptr2->link = ptr->link;
    ptr->link = ptr2;
}
```

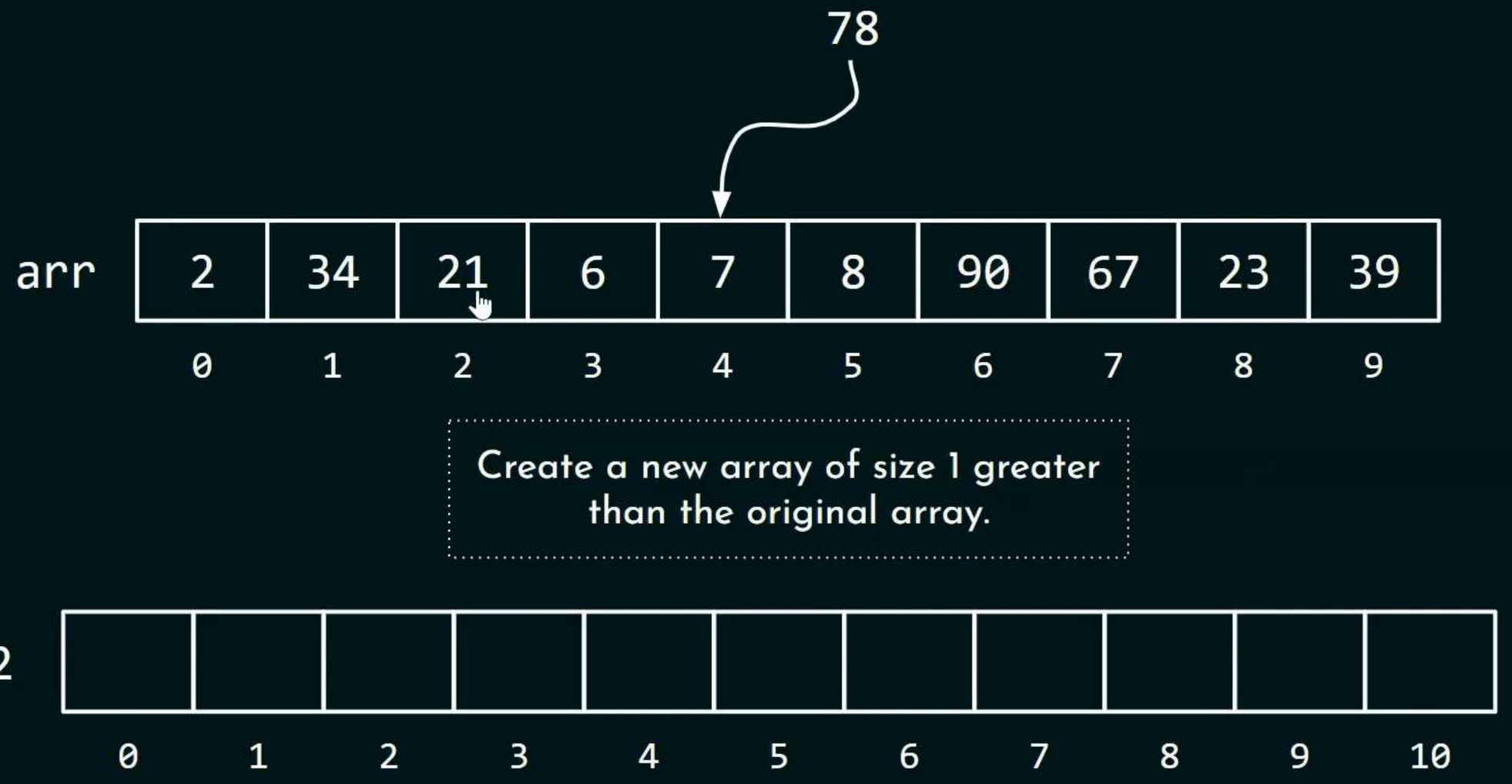
TIME COMPLEXITY: O(n)

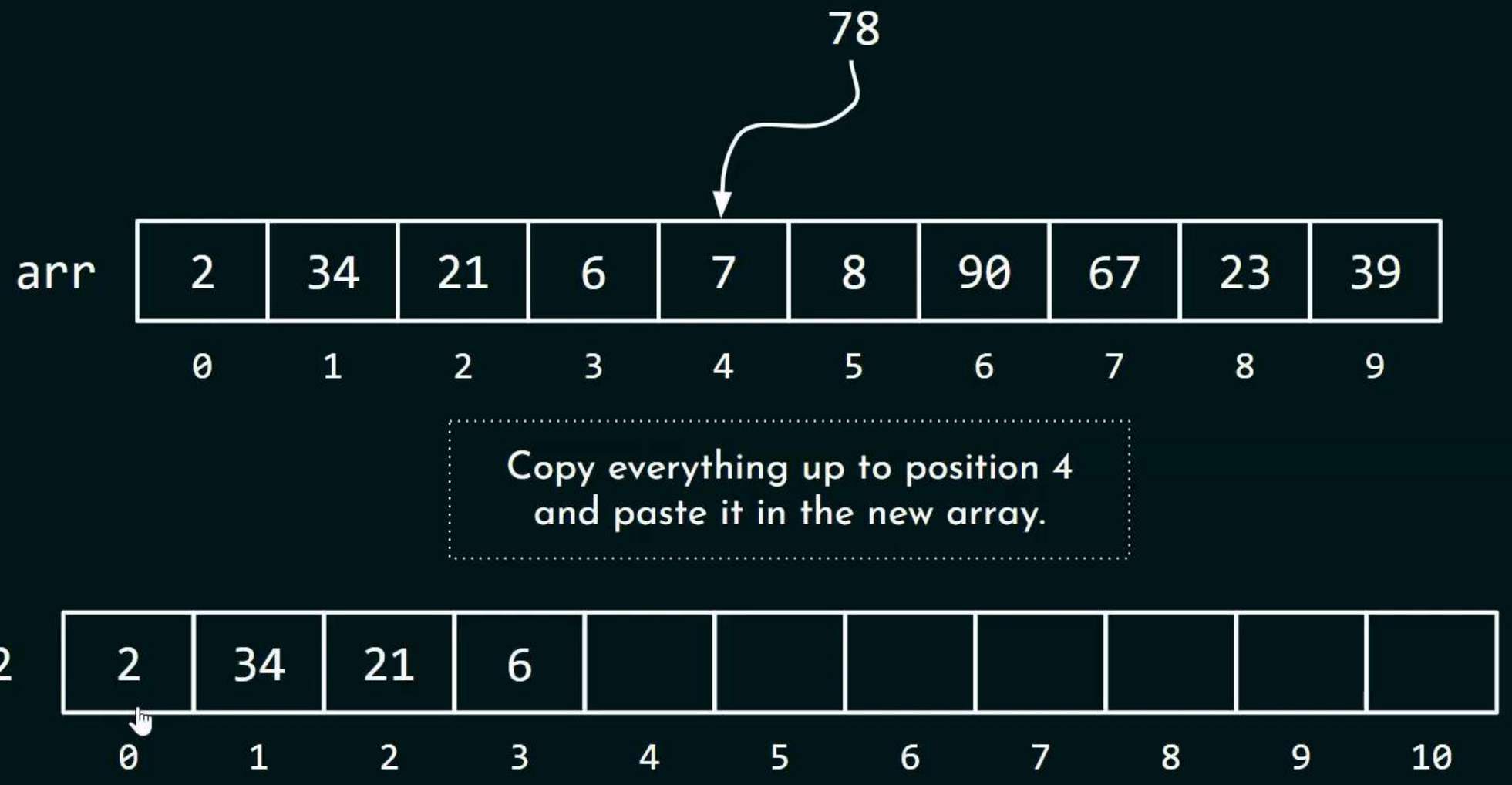


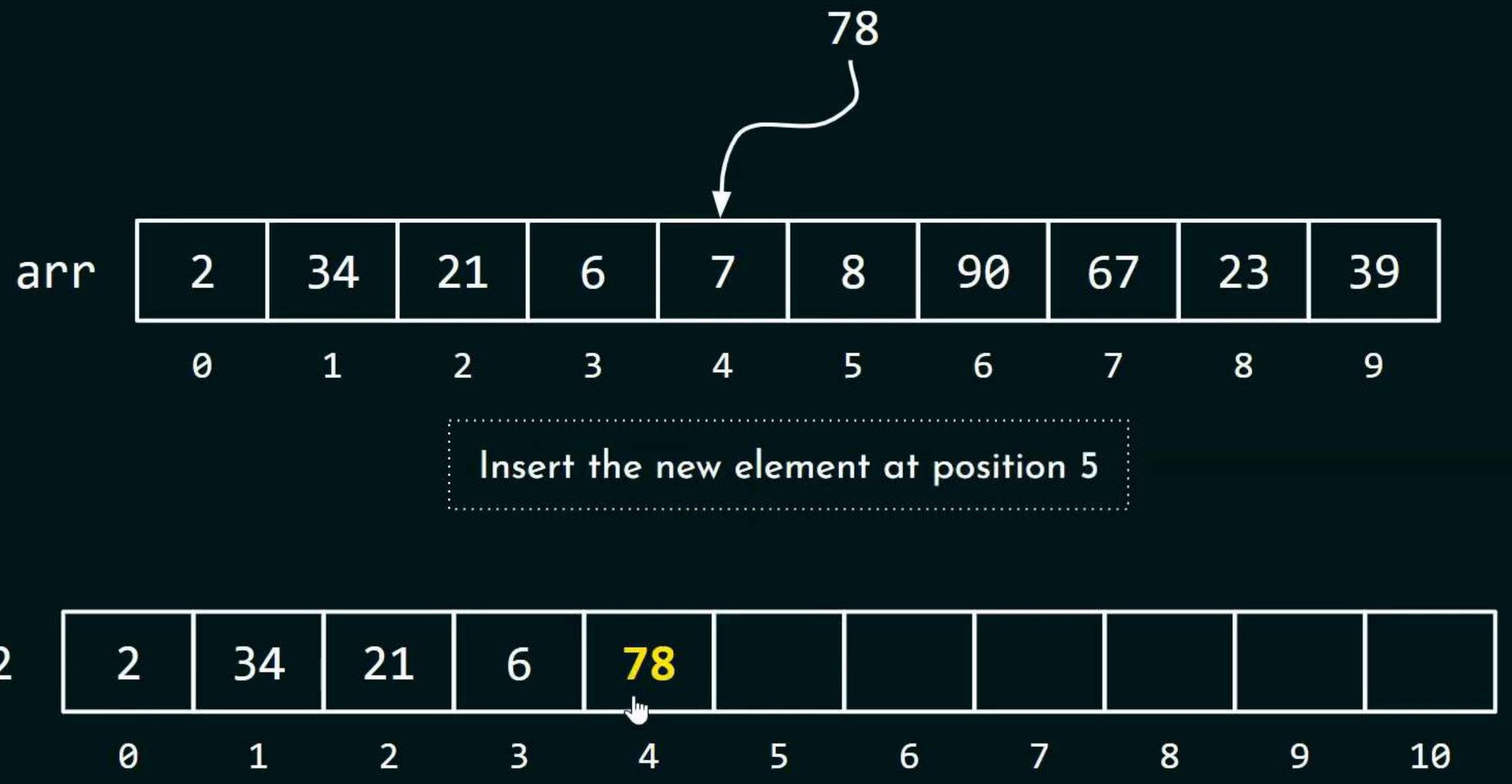

# Programming and Data Structures

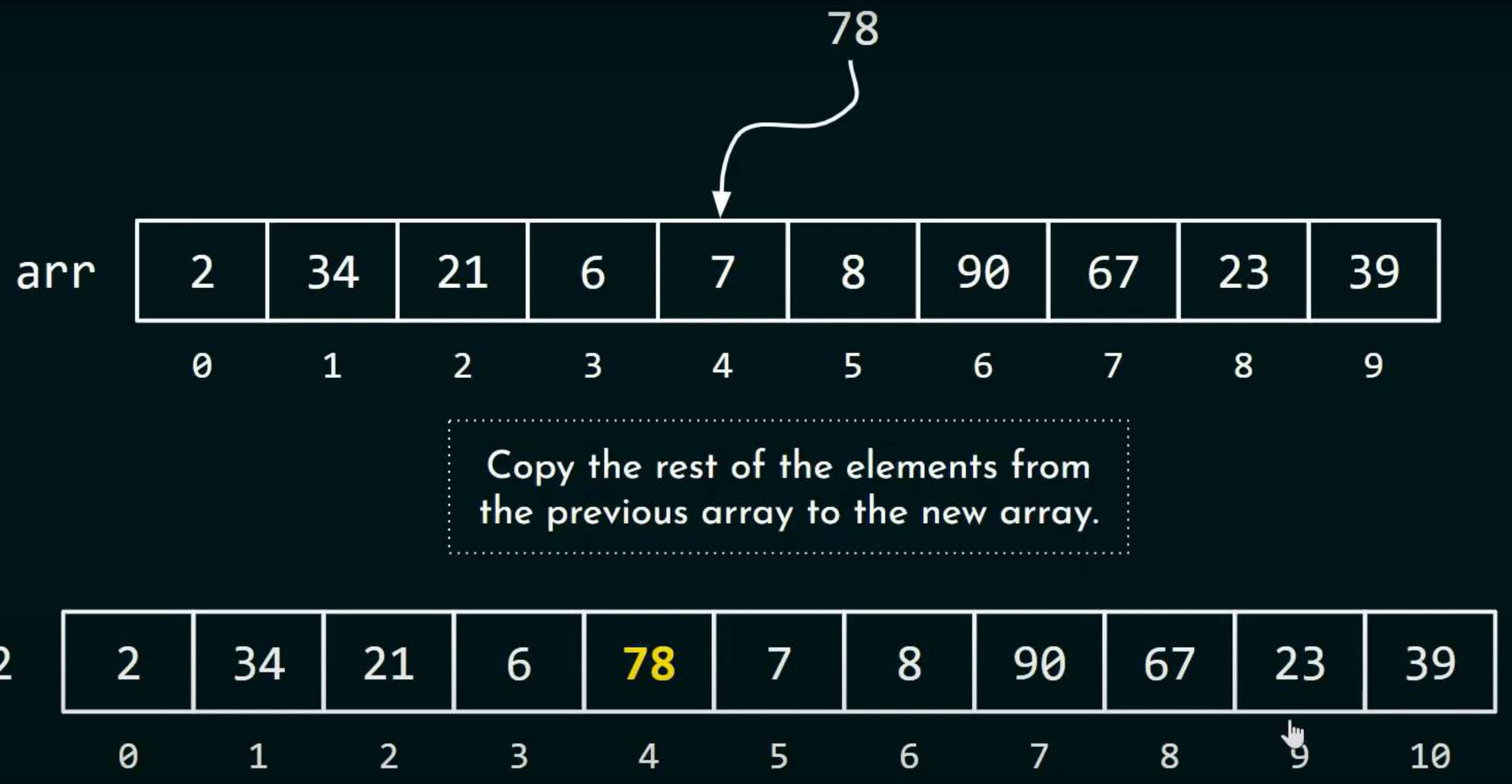
## Insertion at certain position (Linked List vs Array) (Part 2)


We have to insert a new element at position 5.





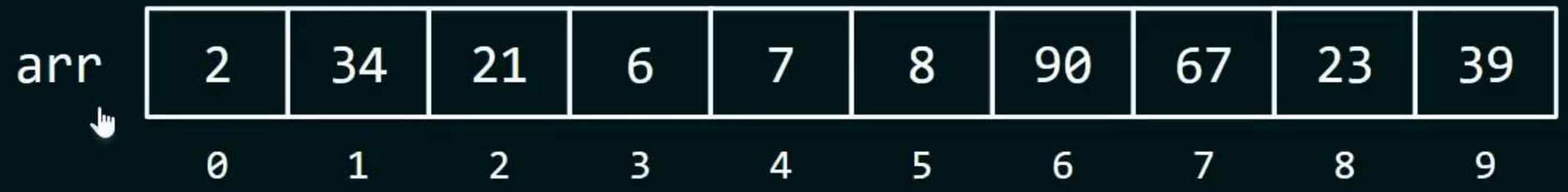



78
arr
2 34 21 6 7 8 90 67 23 39
0 1 2 3 4 5 6 7 8 9
Copy the rest of the elements from the previous array to the new array.
arr2
2 34 21 6 78 7 8 90 67 23 39
0 1 2 3 4 5 6 7 8 9 10

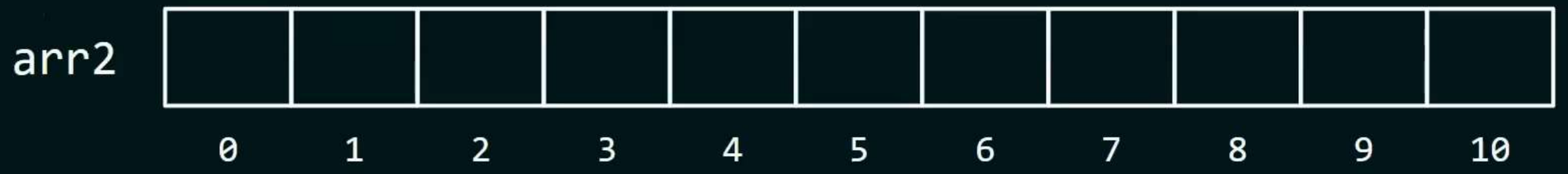
# Program

```
#include <stdio.h>

int main()
{
    int arr[] = {2, 34, 21, 6, 7, 8, 90, 67, 23, 39};
    int pos = 5, data = 78, i;
    int size = sizeof(arr)/sizeof(arr[0]);
    int arr2[size+1];
    add_at_pos(arr, arr2, size, data, pos);
    for(i=0; i<size+1; i++)
        printf("%d ", arr2[i]);
}
```







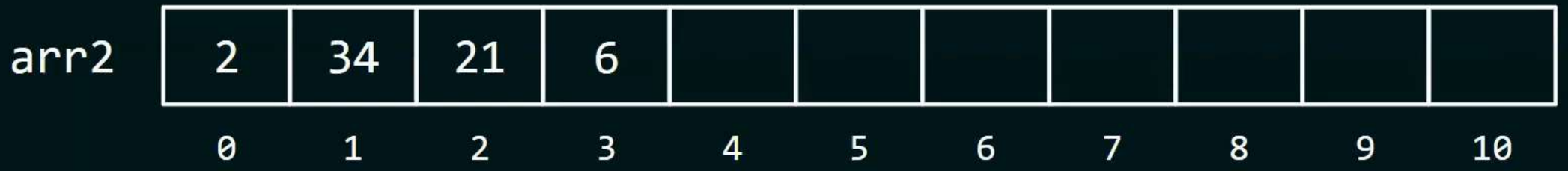


# Add at position

```
void add_at_pos(int arr[], int arr2[], int n, int data, int pos)
{
    int i;
    int index = pos-1;
    for(i=0; i<=index-1; i++)
        arr2[i] = arr[i];

    arr2[index] = data;
    int j;
    for(i=index+1, j=index; i<n+1, j<n; i++, j++)
        arr2[i] = arr[j];

}
```


index





# Add at position

```
void add_at_pos(int arr[], int arr2[], int n, int data, int pos)
{
    int i;
    int index = pos-1;
    for(i=0; i<=index-1; i++)
        arr2[i] = arr[i];

    arr2[index] = data;
    int j;
    for(i=index+1, j=index; i<n+1, j<n; i++, j++)
        arr2[i] = arr[j];

}
```



# Add at position

```
void add_at_pos(int arr[], int arr2[], int n, int data, int pos)
{
    int i;
    int index = pos-1;
    for(i=0; i<=index-1; i++)
        arr2[i] = arr[i];

    arr2[index] = data;
    int j;
    for(i=index+1, j=index; i<n+1, j<n; i++, j++)
        arr2[i] = arr[j];

}
```

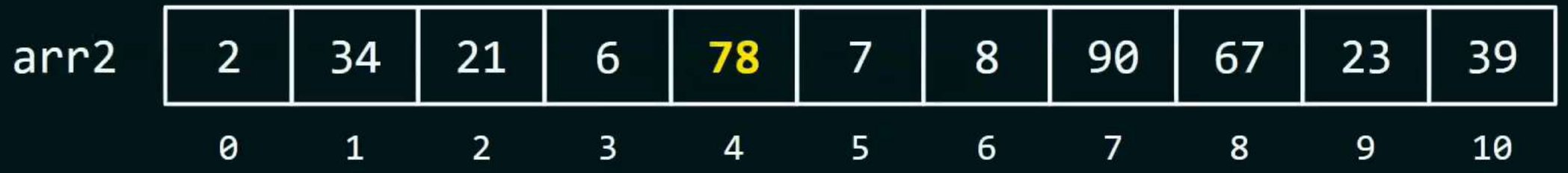

index

# Add at position

```
void add_at_pos(int arr[], int arr2[], int n, int data, int pos)
{
    int i;
    int index = pos-1;
    for(i=0; i<=index-1; i++)
        arr2[i] = arr[i];

    arr2[index] = data;
    int j;
    for(i=5, j=4; i<11, j<10; i++, j++)
        arr2[i] = arr[j];

}
```


index





# Program

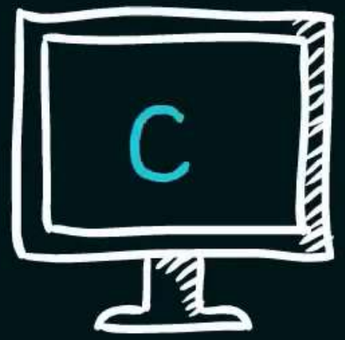
```
#include <stdio.h>

int main()
{
    int arr[] = {2, 34, 21, 6, 7, 8, 90, 67, 23, 39};
    int pos = 5, data = 78, i;
    int size = sizeof(arr)/sizeof(arr[0]);
    int arr2[size+1];
    add_at_pos(arr, arr2, size, data, pos);
    for(i=0; i<size+1; i++)
        printf("%d ", arr2[i]);
}
```

Output: 2 34 21 6 78 7 8 90 67 23 39



# Remember

- We can't return an array from a function because an array created in the function is local to that function.
- This is the reason why arr2[] has been passed as a parameter to a function.
- Passing arr2 (name of the array) is equivalent to passing the address of the first block of arr2. So the changes made in called function will be reflected in the caller function.




# PROGRAMMING AND DATA STRUCTURES

## Deletion at the end (Linked List vs Arrays)

# Function

```
void del_last(struct node *head)
{
    if(head == NULL)
        printf("List is already empty!");
    else if(head->link == NULL)
    {
        free(head);
        head = NULL;
    }
```

Time complexity: O(n)

```
else
    {
    struct node *temp = head;
    struct node *temp2 = head;
    while(temp->link != NULL)
    {
        temp2 = temp;
        temp = temp->link;
    }
    temp2->link = NULL;
    free(temp);
    temp = NULL;
    }
    return head;
}
```

# Program

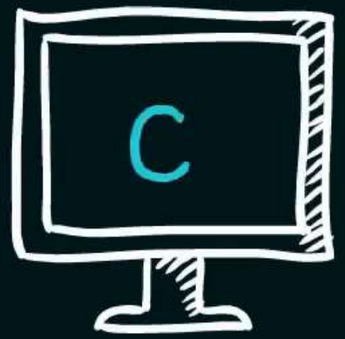
```
#include <stdio.h>

int main() {
     int arr[] = {23, 3, 45, 12, 67, 54, 6, 4};
     int size = sizeof(arr)/sizeof(arr[0]);
     int i;
     size = size - 1;
     for(i=0; i<size; i++)
         printf("%d ", arr[i]);
     return 0;
}
```

Time complexity: O(1)



# Summary

| Linked List | Array |
| --- | --- |
| Deletion at the end: O(n) | Deletion at the end: O(1) |




# PROGRAMMING AND DATA STRUCTURES

## Deletion in the Beginning (Linked List vs Arrays)

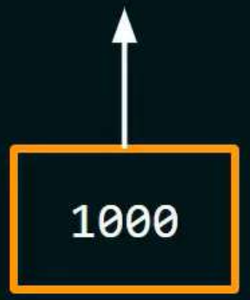
# Function

```
struct node* del_first(struct node *head)
{
    if(head == NULL)
        printf("List is already empty!");
    else
    {
    struct node *temp = head;
    head = head->link;
    free(temp);
    temp = NULL;
    }
    return head;
}
```

Time complexity: O(1)


# Programming and Data Structures

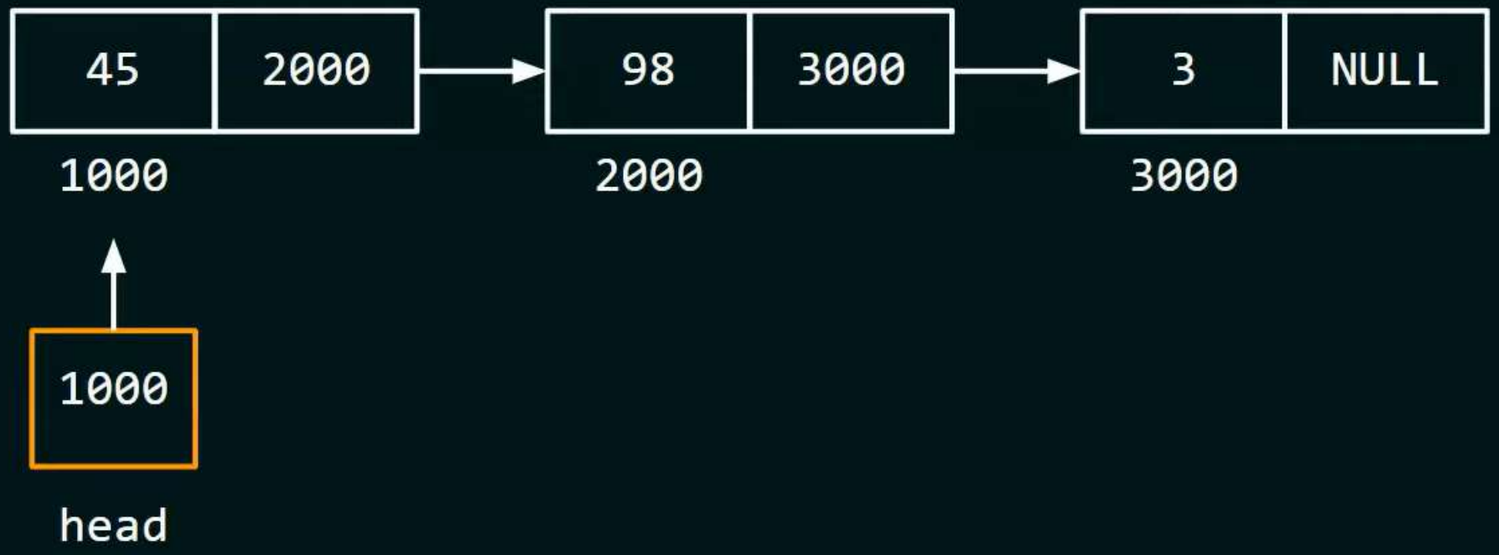
## Reverse a Single Linked List

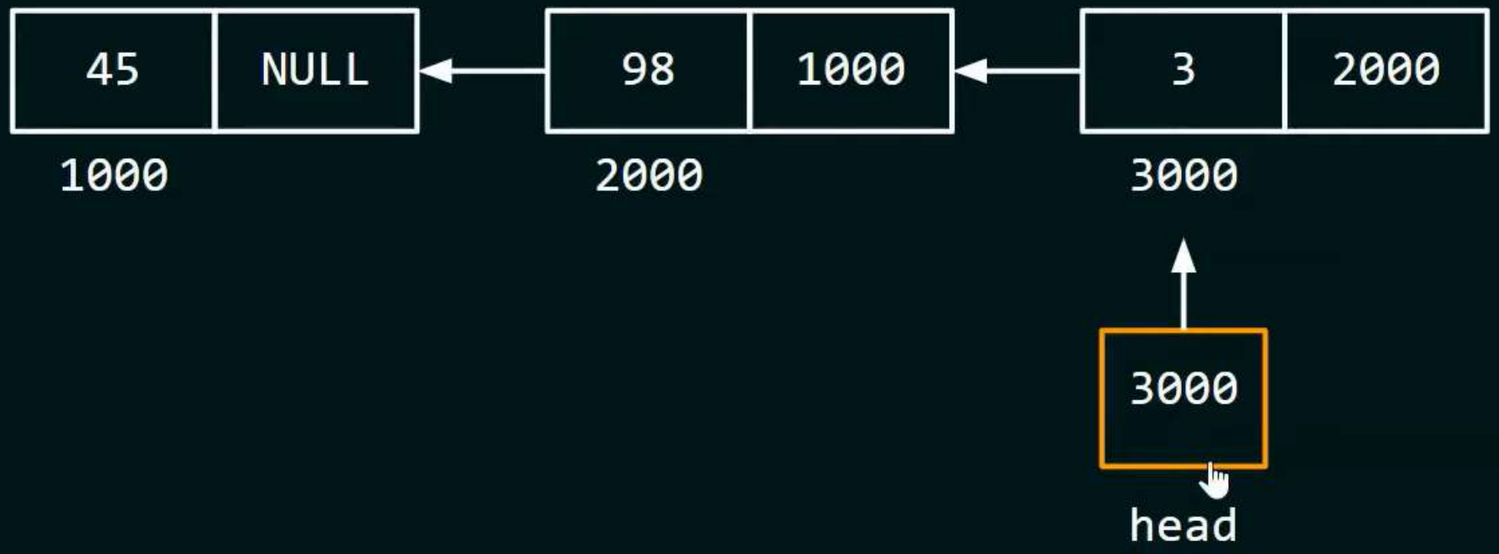

head

Our job is to make the last node of this list, the first node—second last node of this list, the second node, and so on.

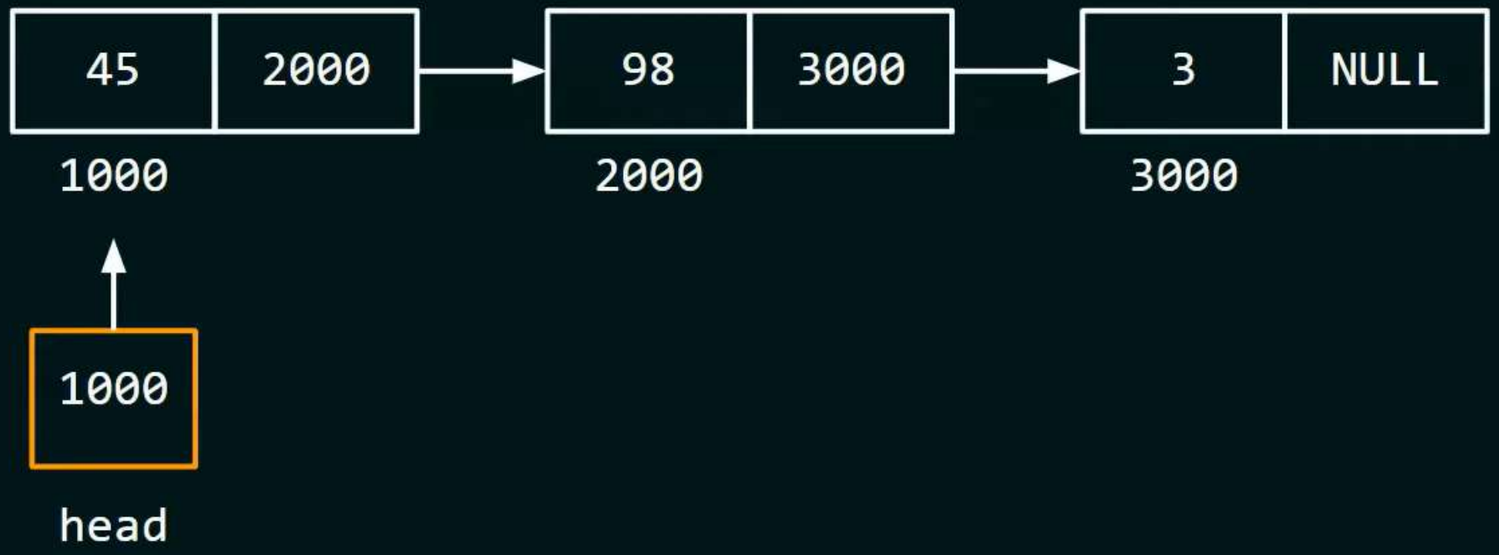
Old List



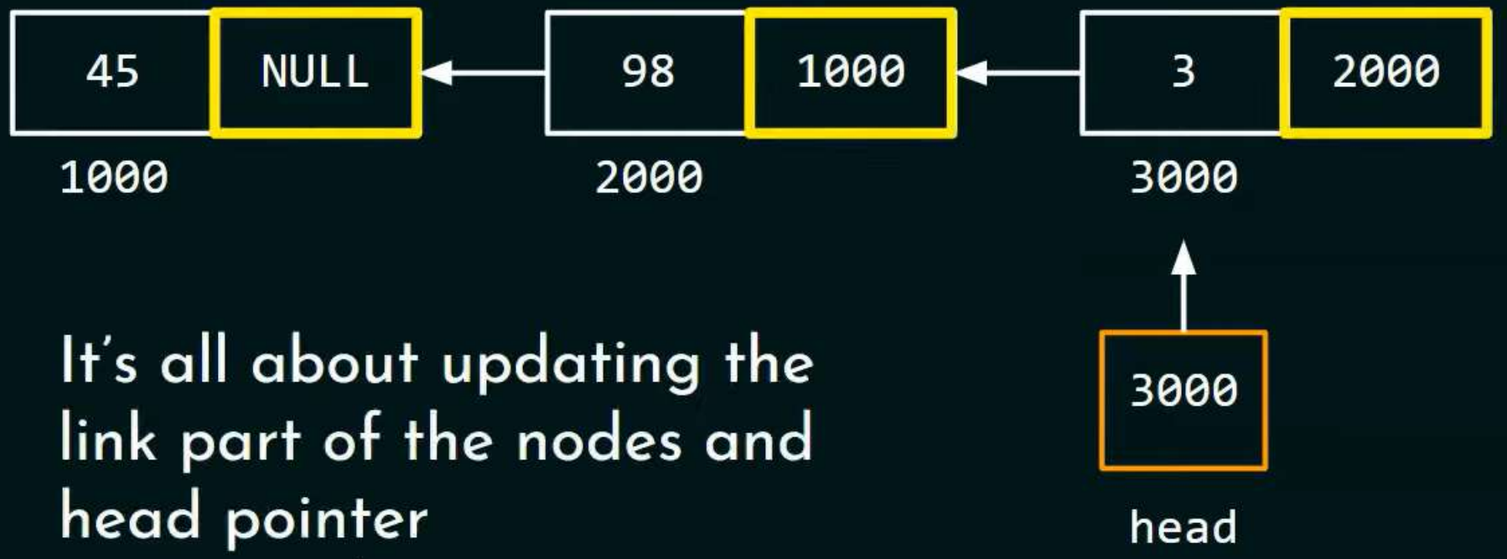

New List



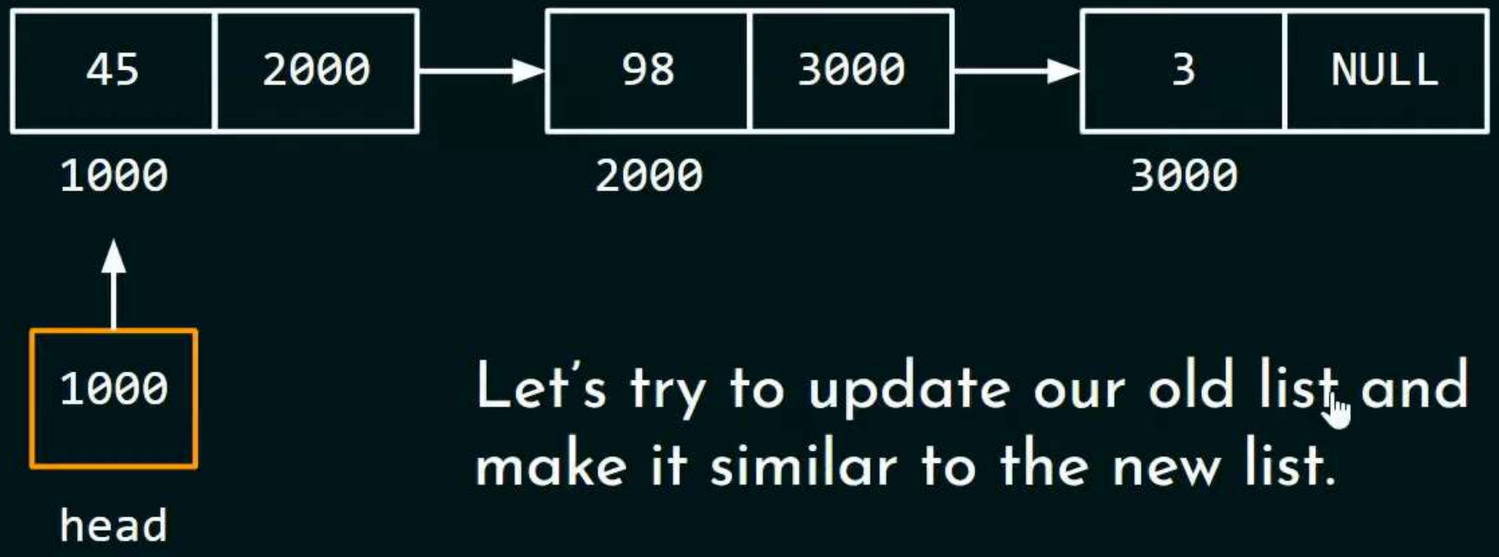
## Old List




## New List

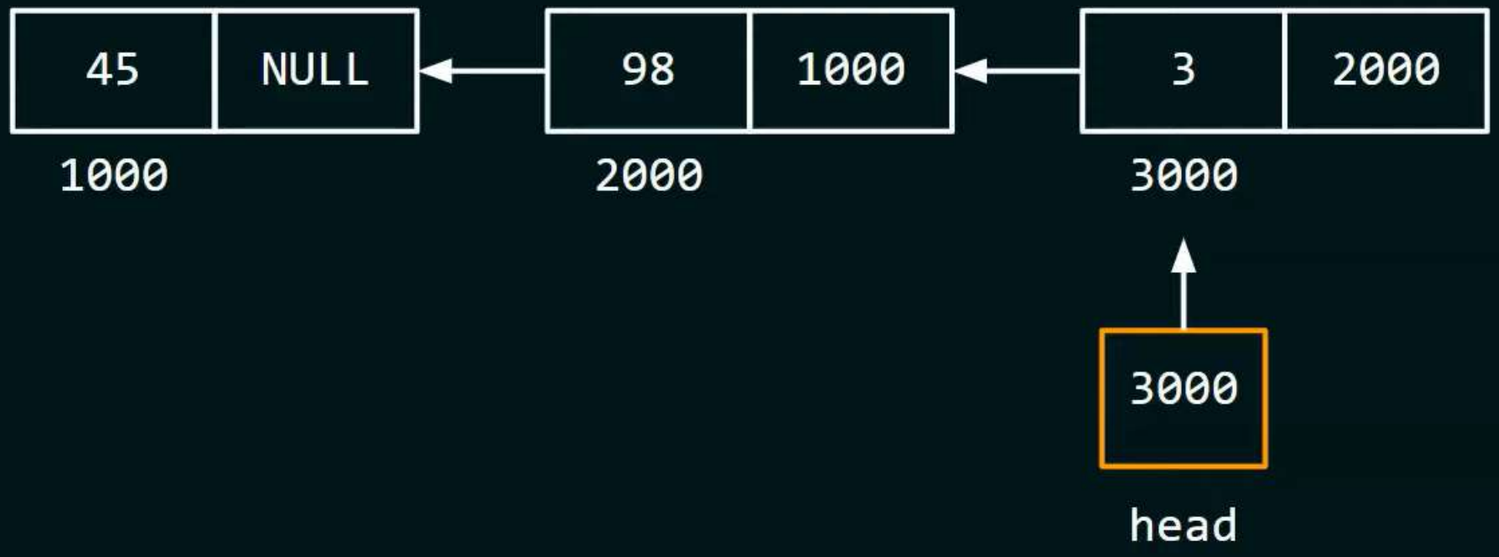



It's all about updating the link part of the nodes and head pointer




Old List

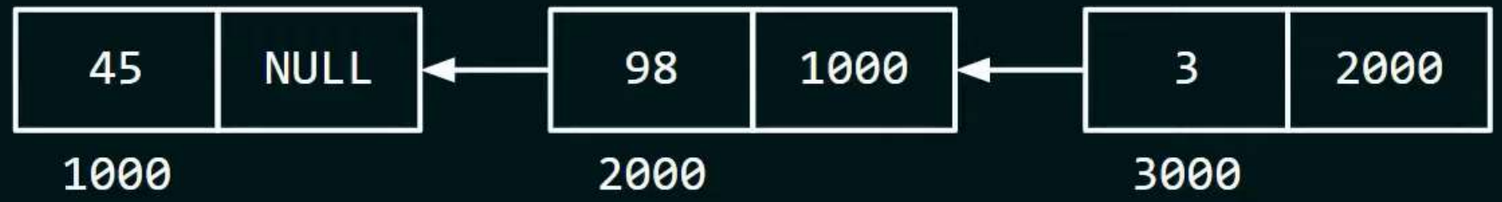
Let's try to update our old list and make it similar to the new list.



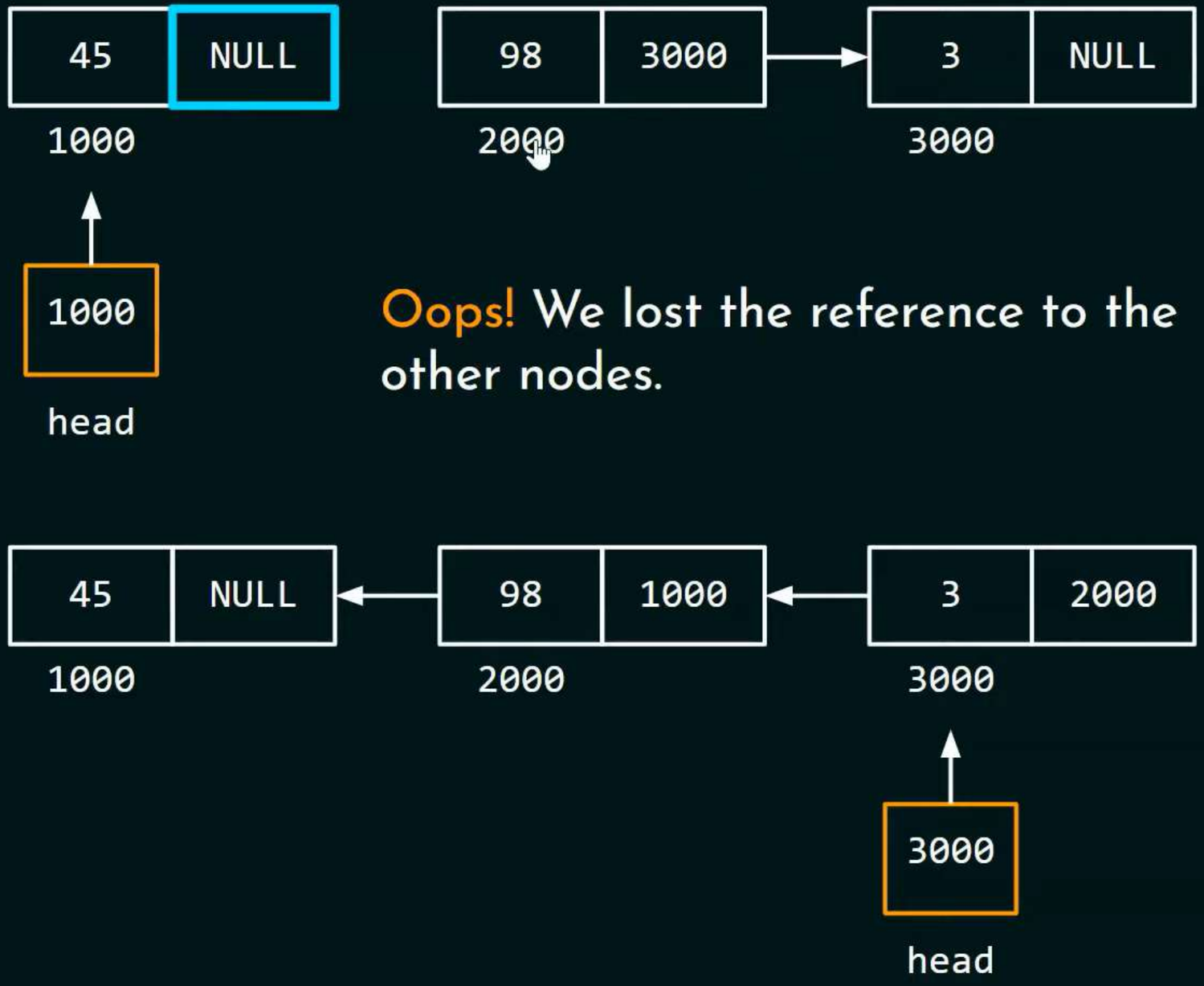

New List

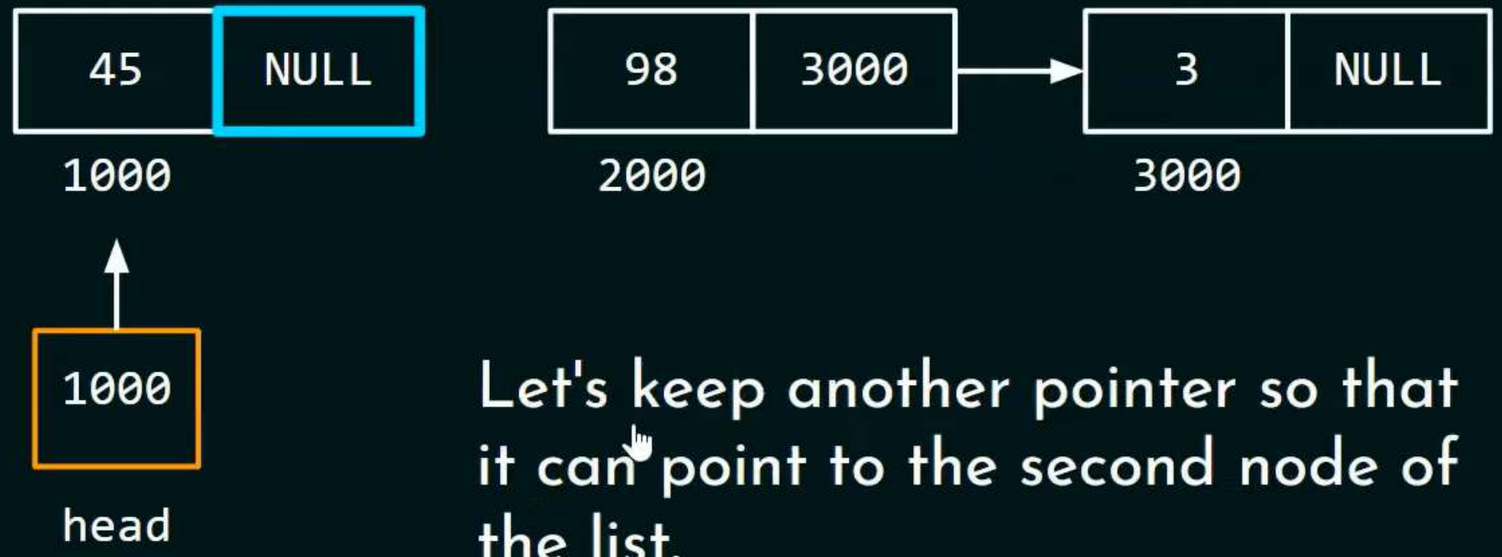

## Old List

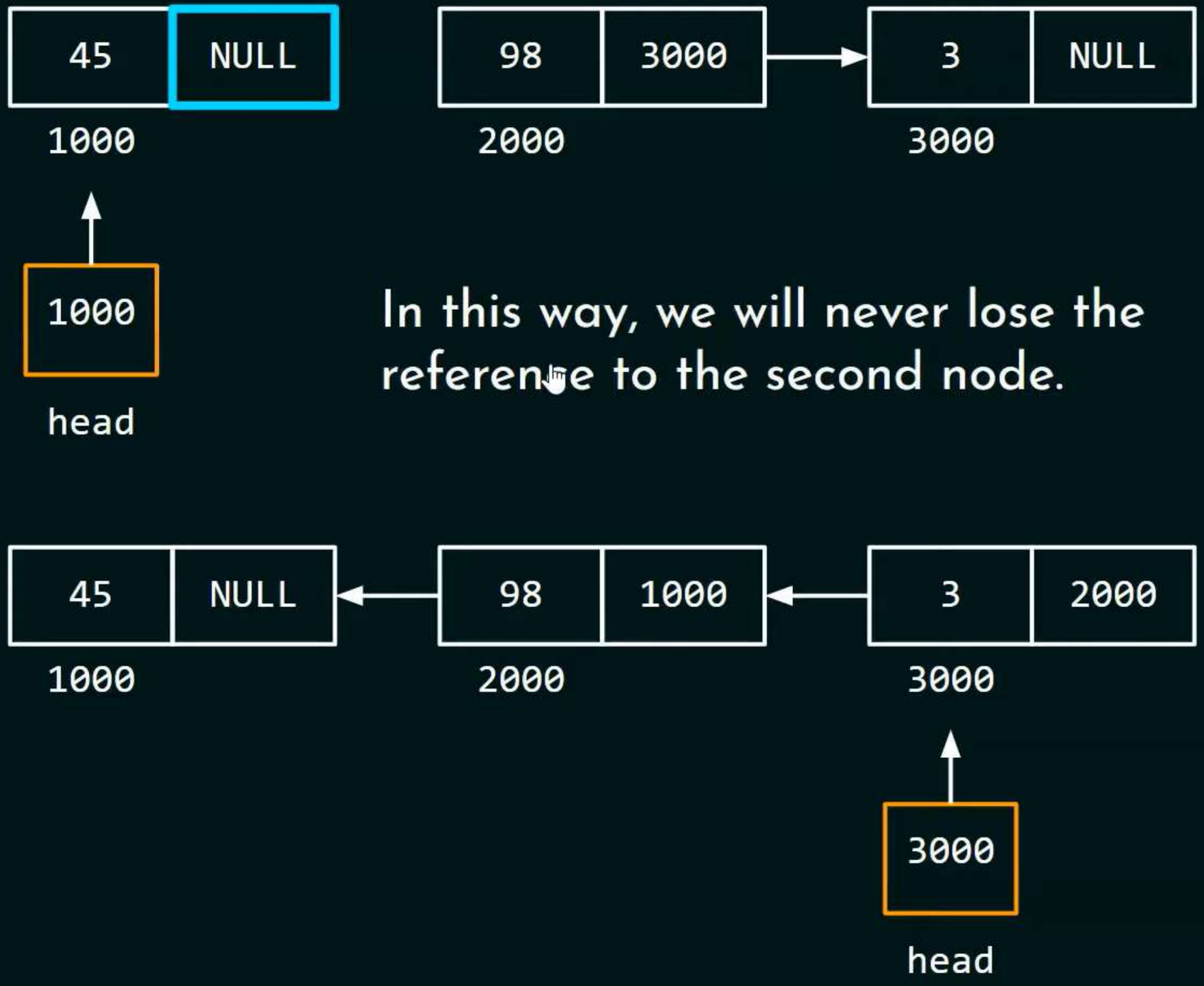

Let's try to update the link part of the first node.



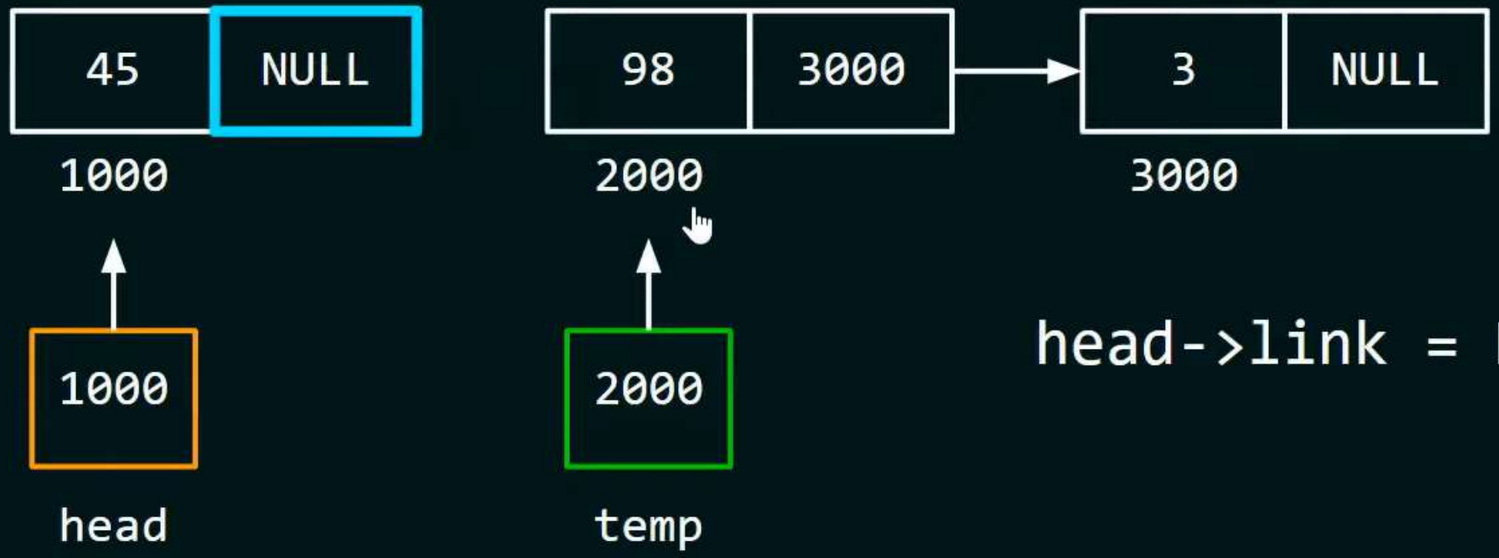

## New List





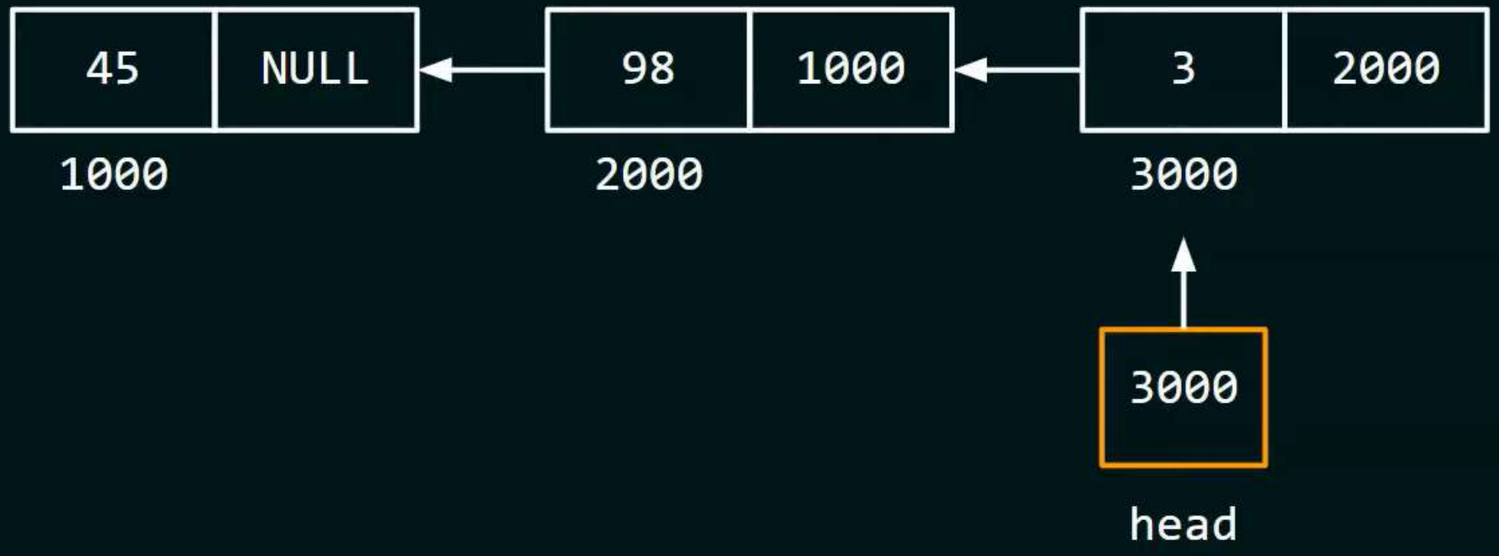


Old List

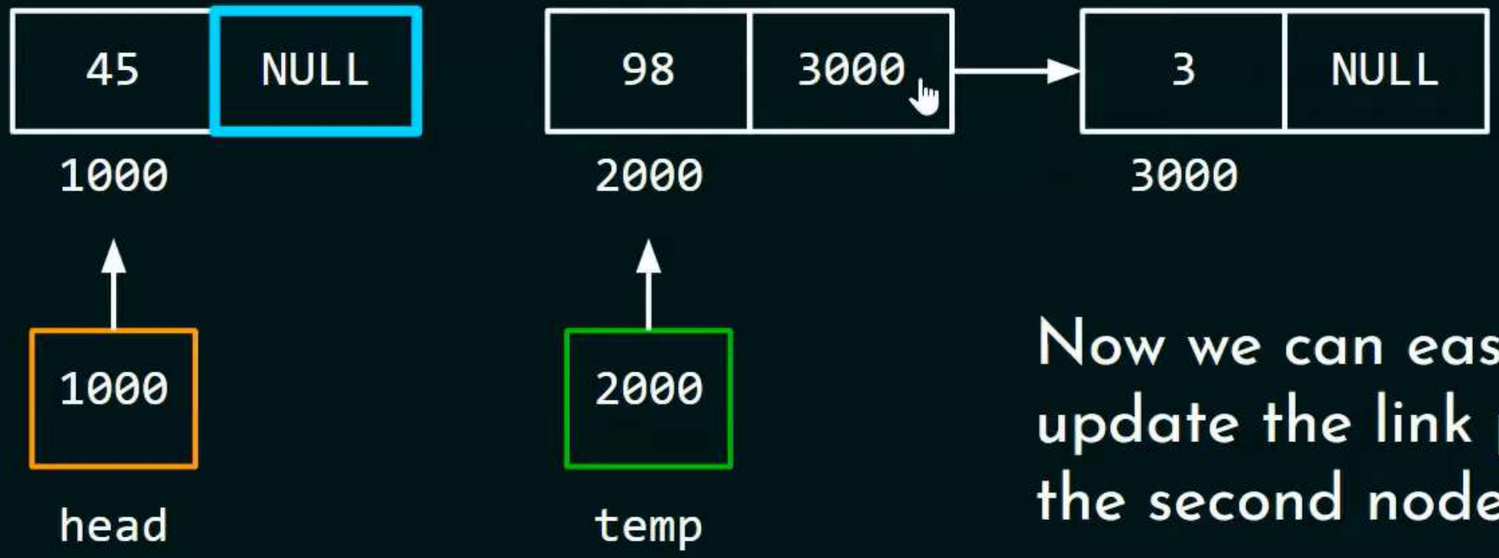



```
head->link = NULL;
```

New List

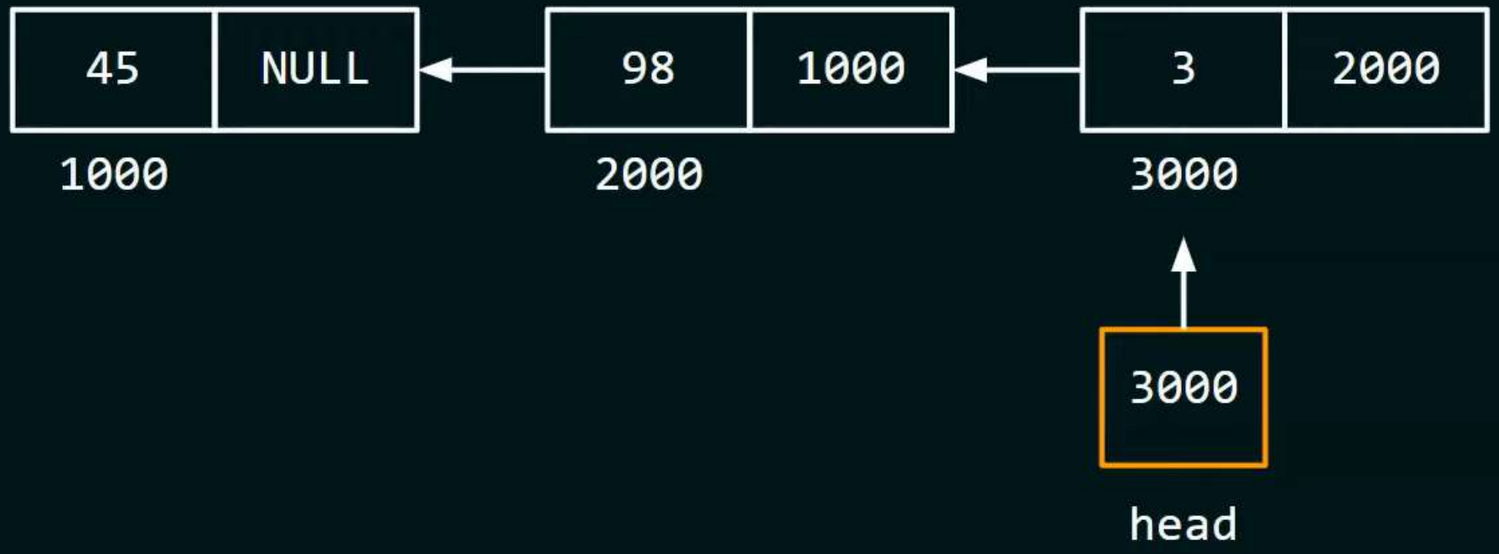


## Old List

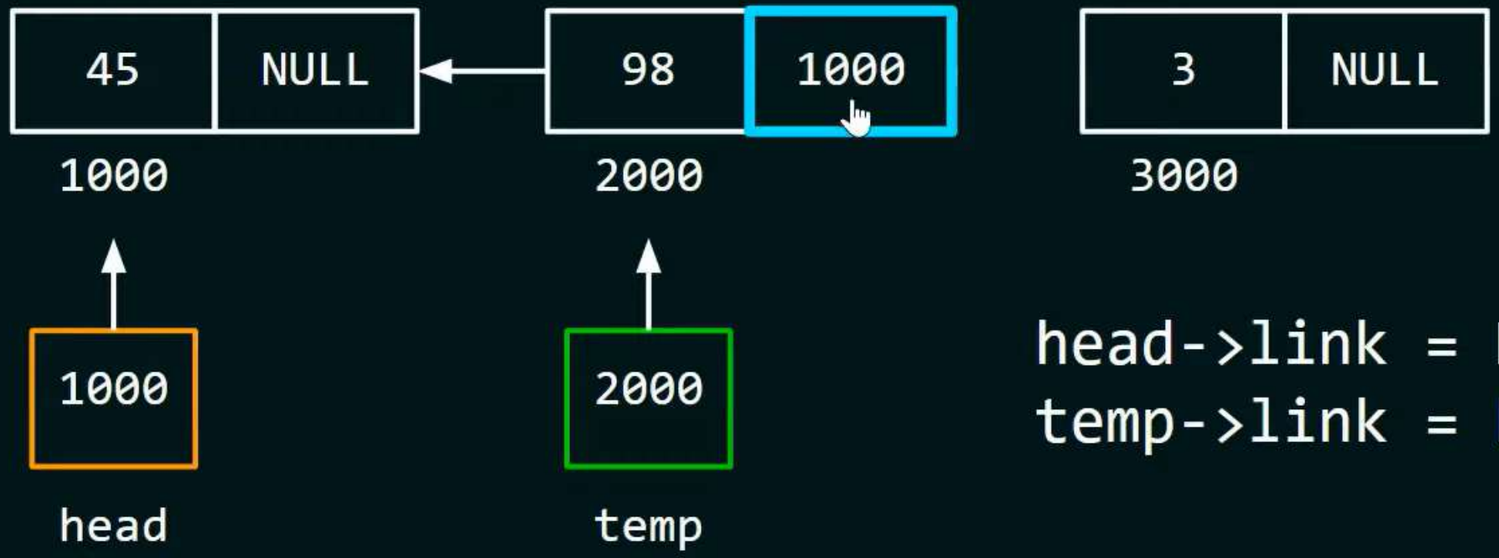



Now we can easily update the link part of the second node.

## New List

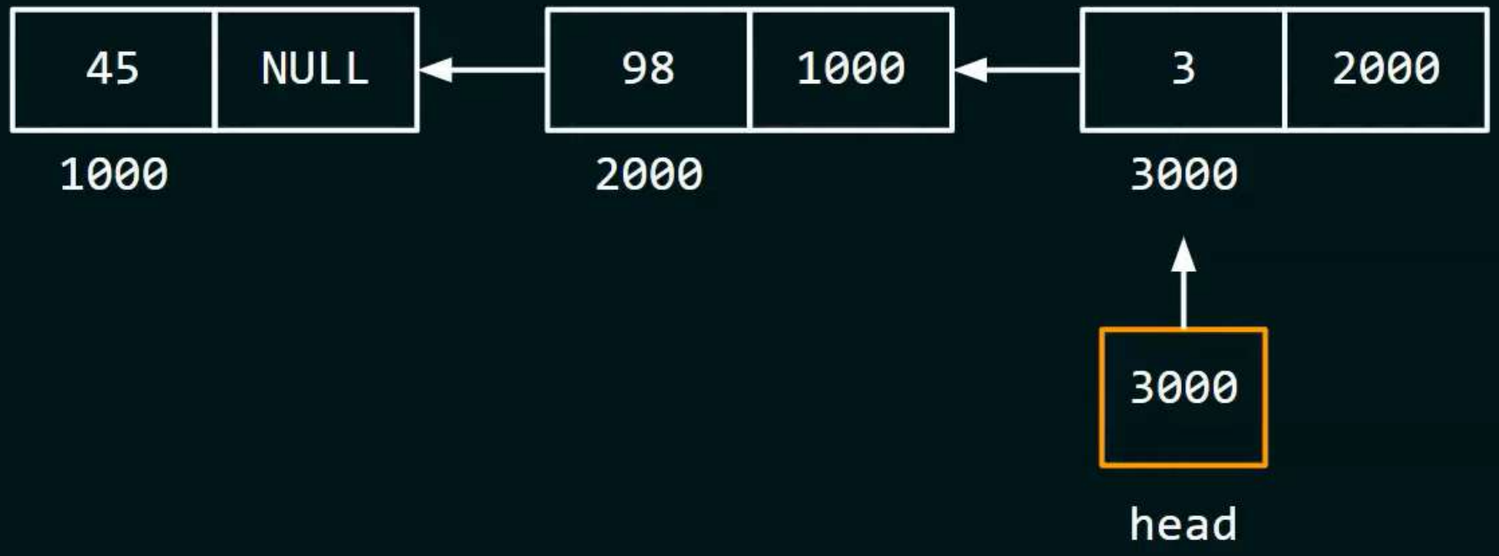


Old List

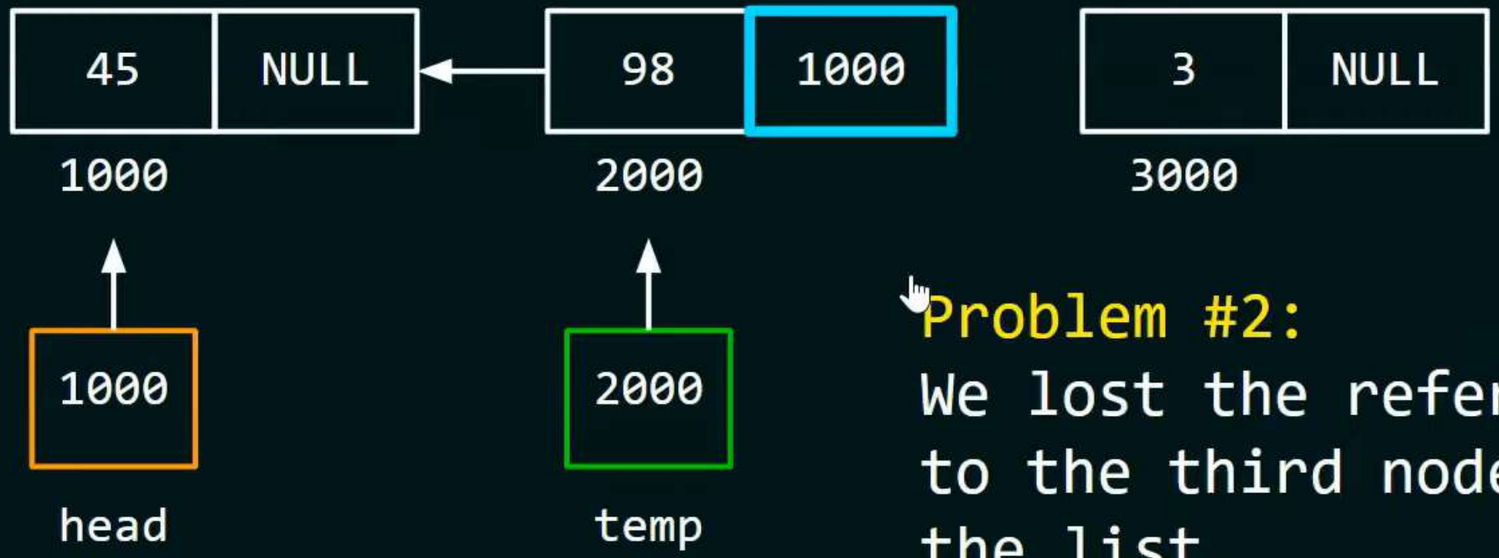



```
head->link = NULL;
temp->link = head;
```

New List

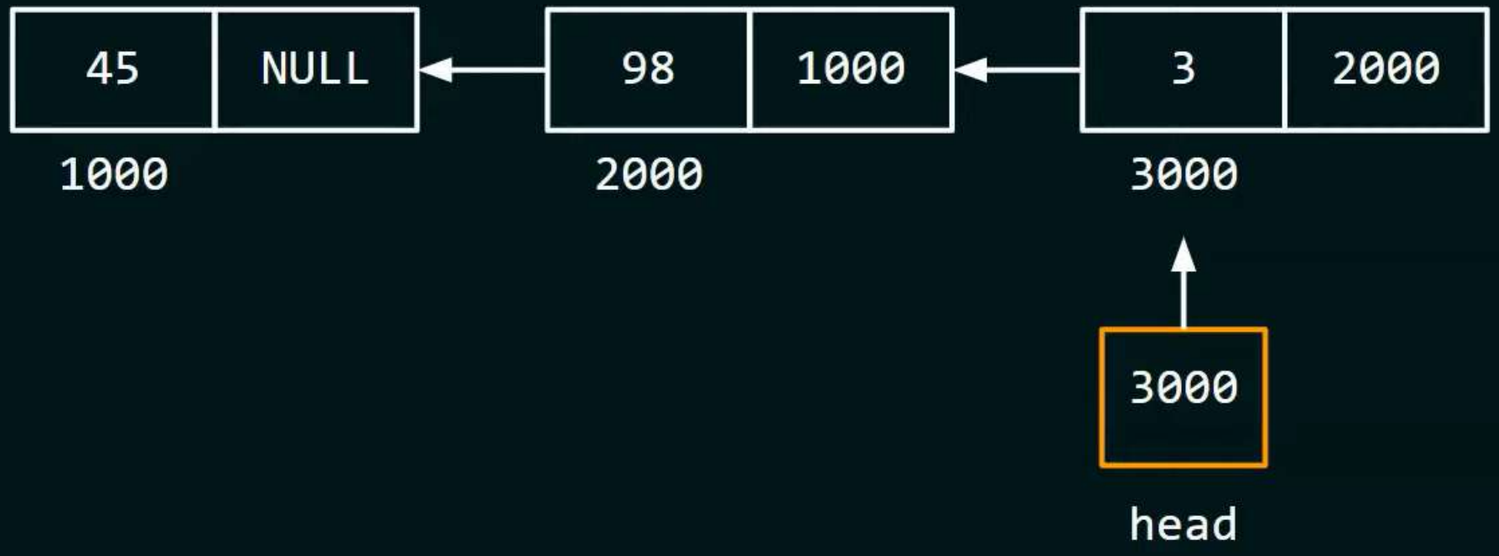


Old List

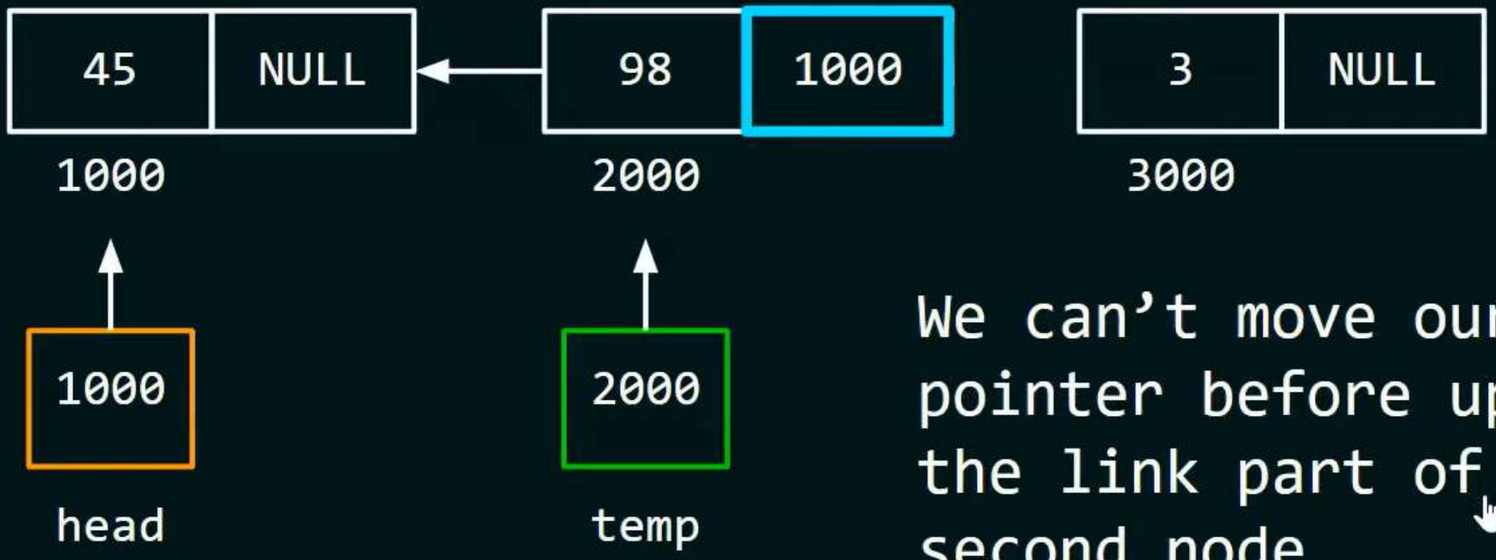



Problem #2:
We lost the reference to the third node of the list.

New List

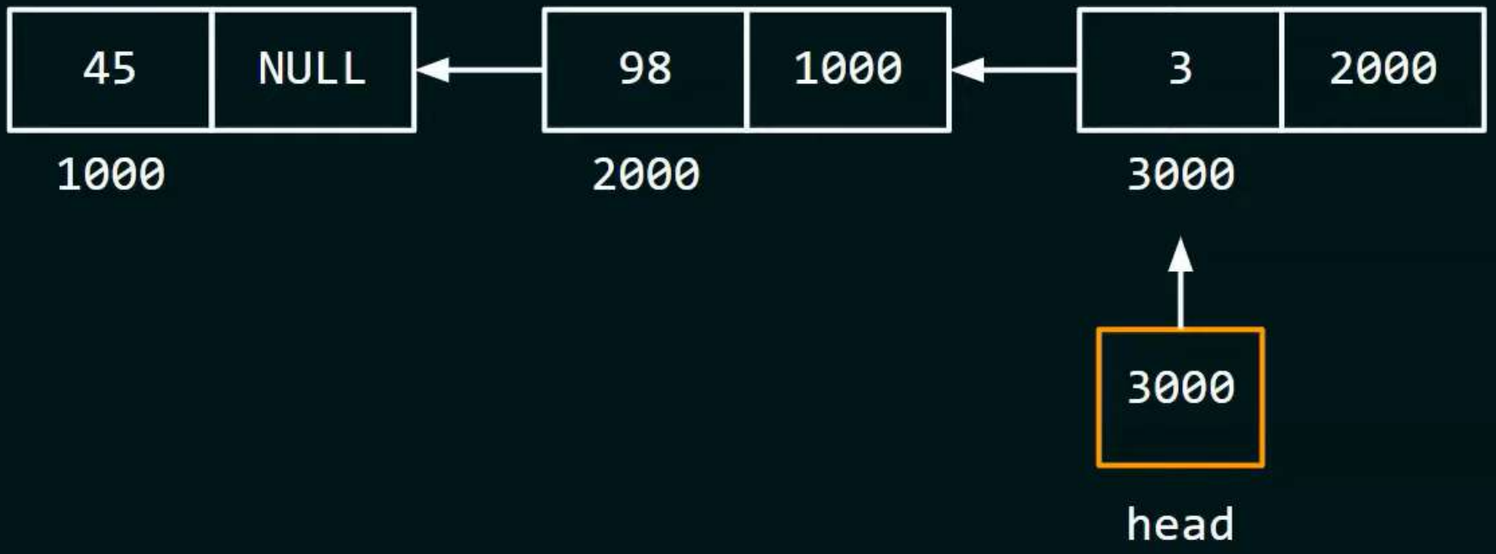


## Old List

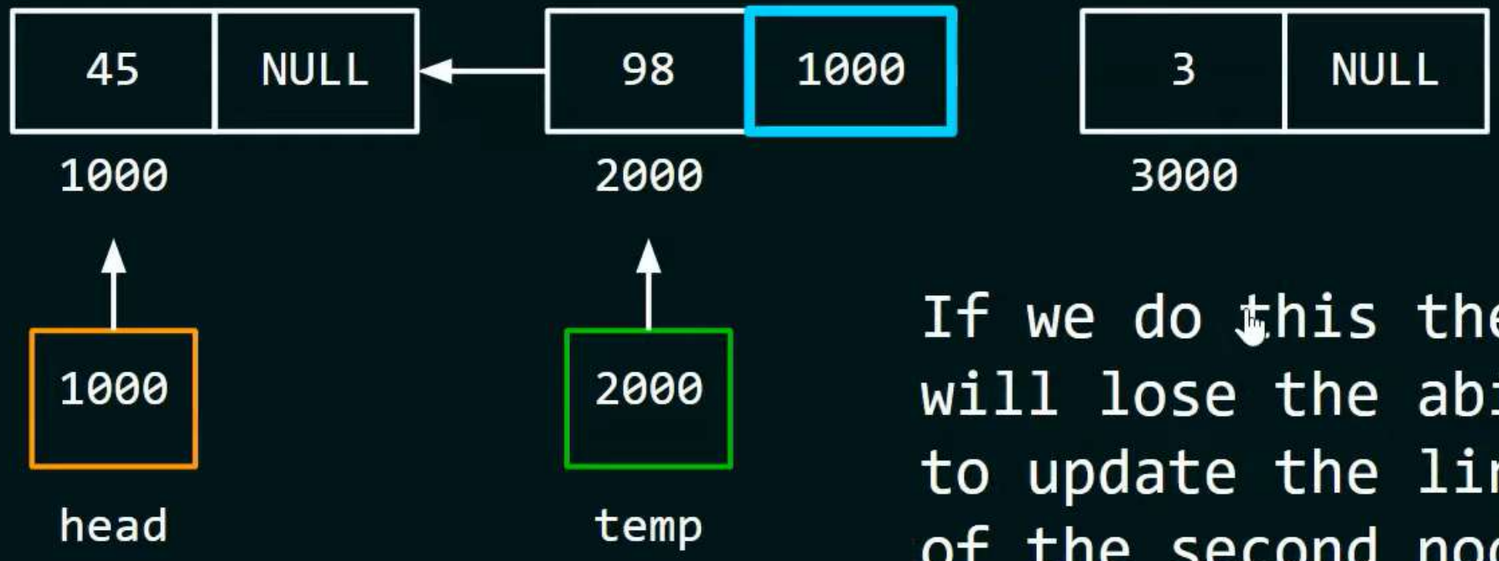



We can't move our head pointer before updating the link part of the second node.

## New List

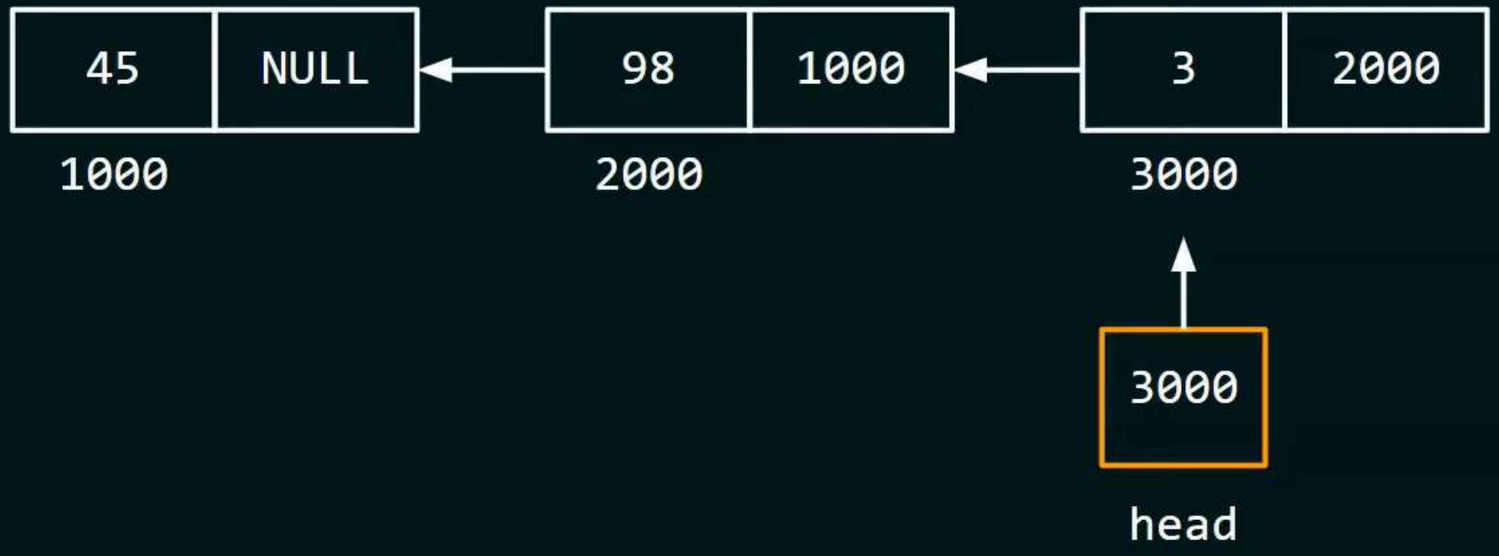


## Old List

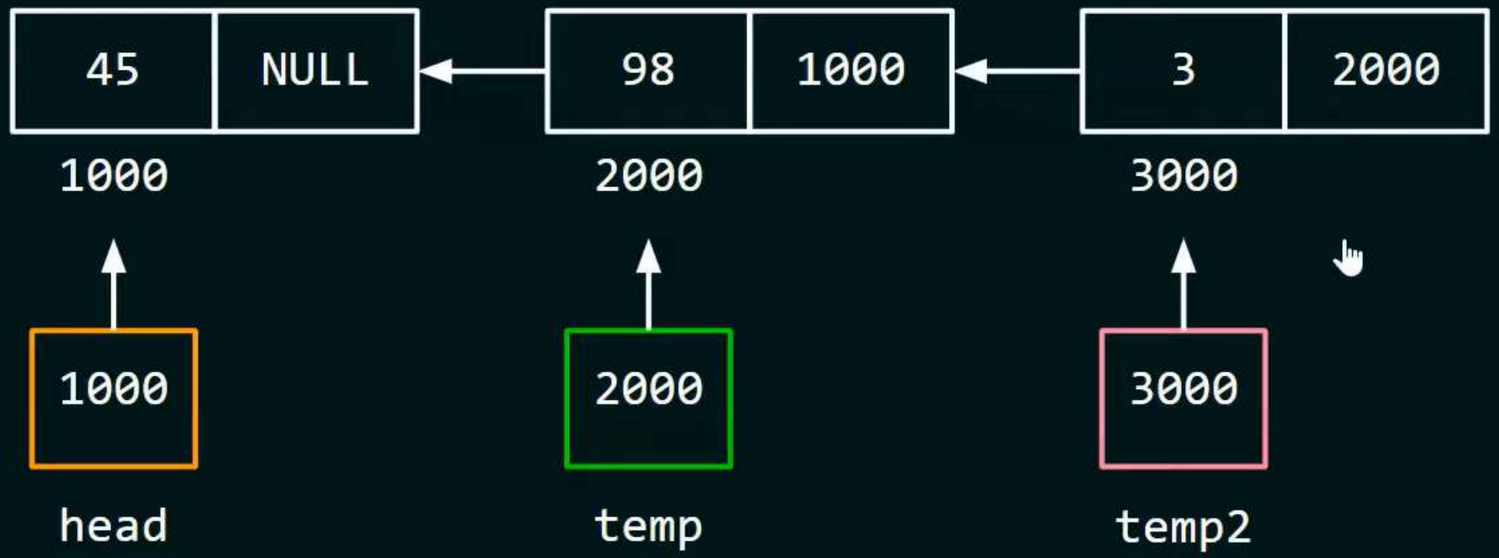



If we do this then we will lose the ability to update the link part of the second node.

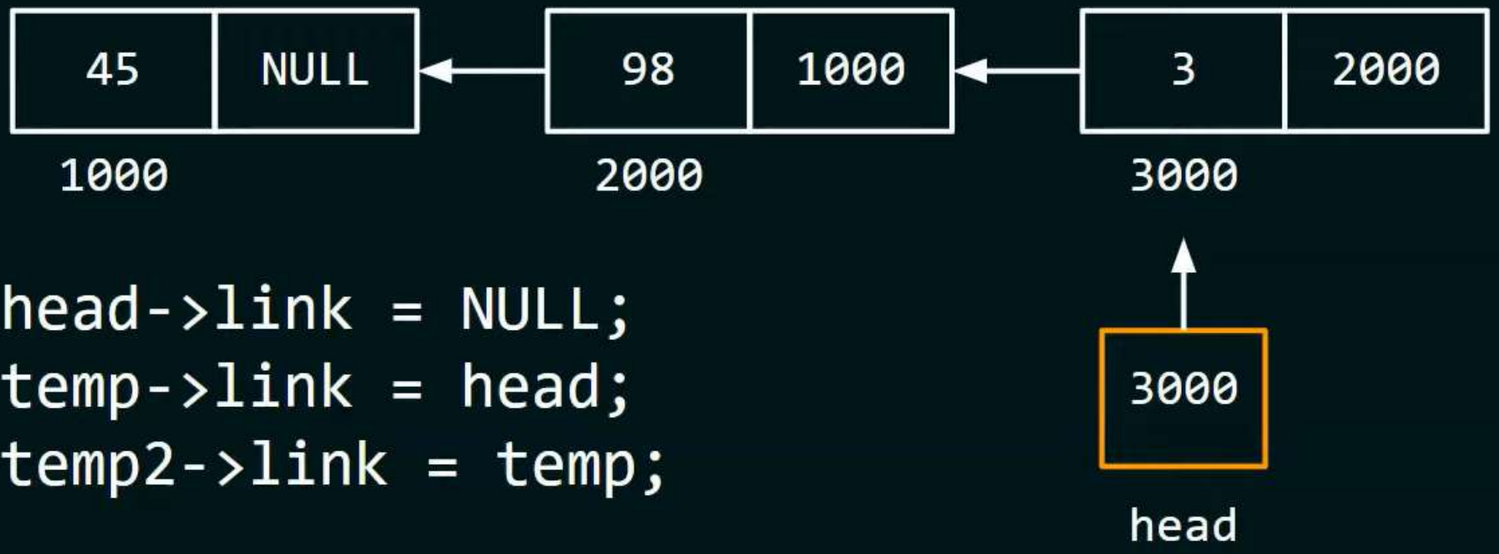
## New List



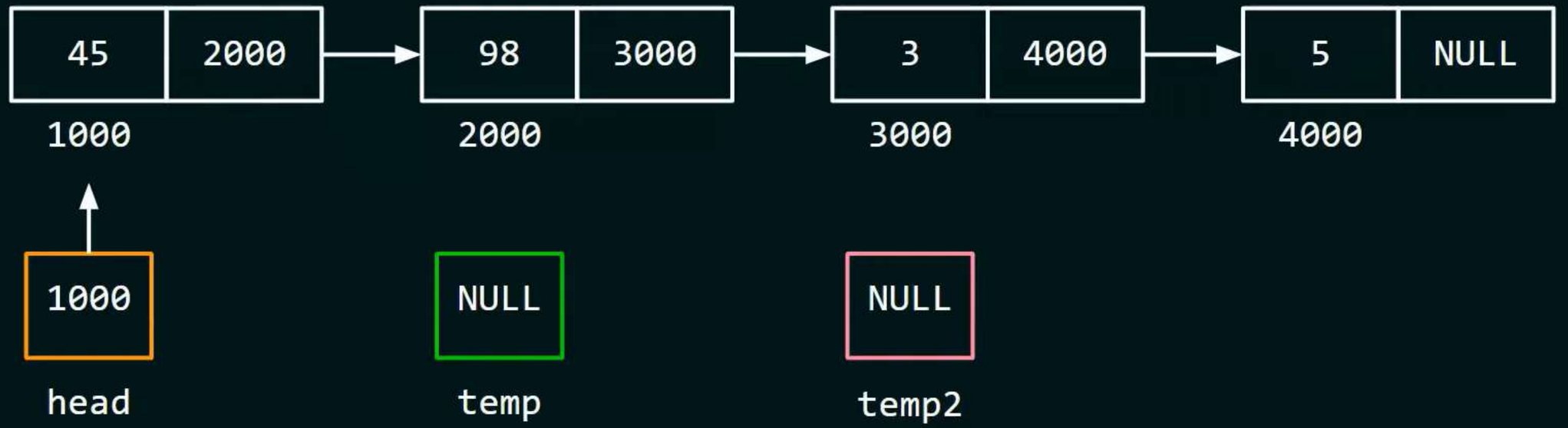
## Old List



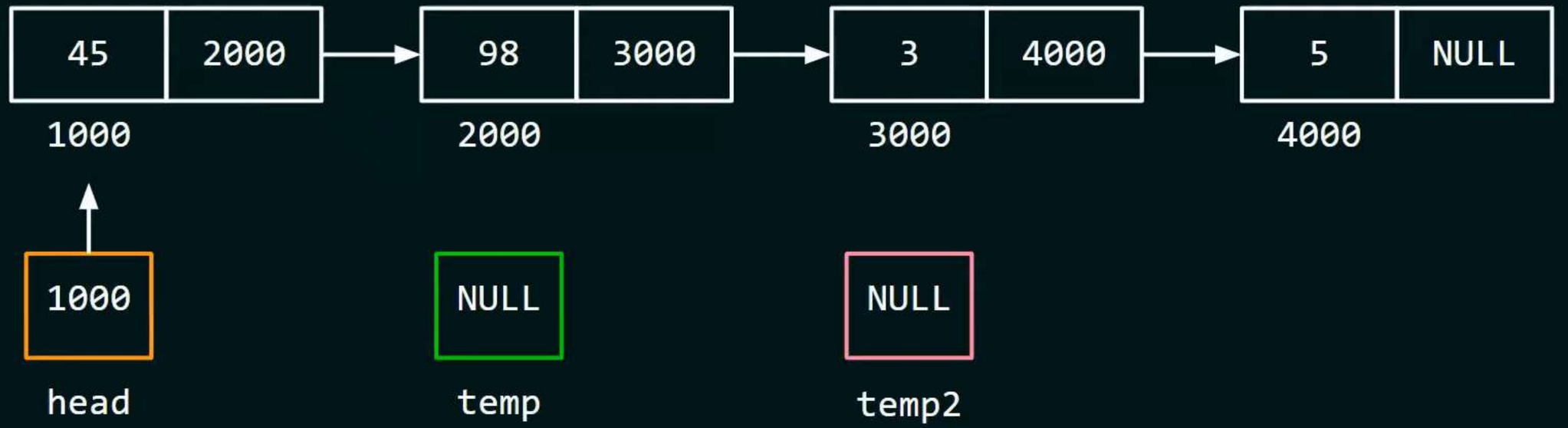

## New List



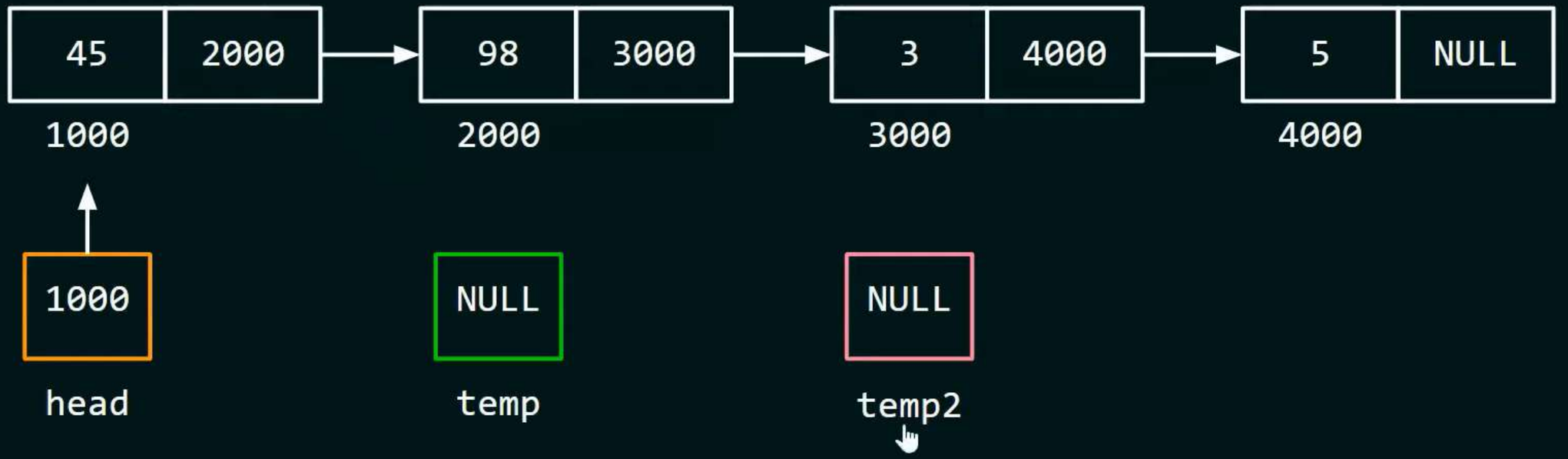

```
head->link = NULL;
temp->link = head;
temp2->link = temp;
```



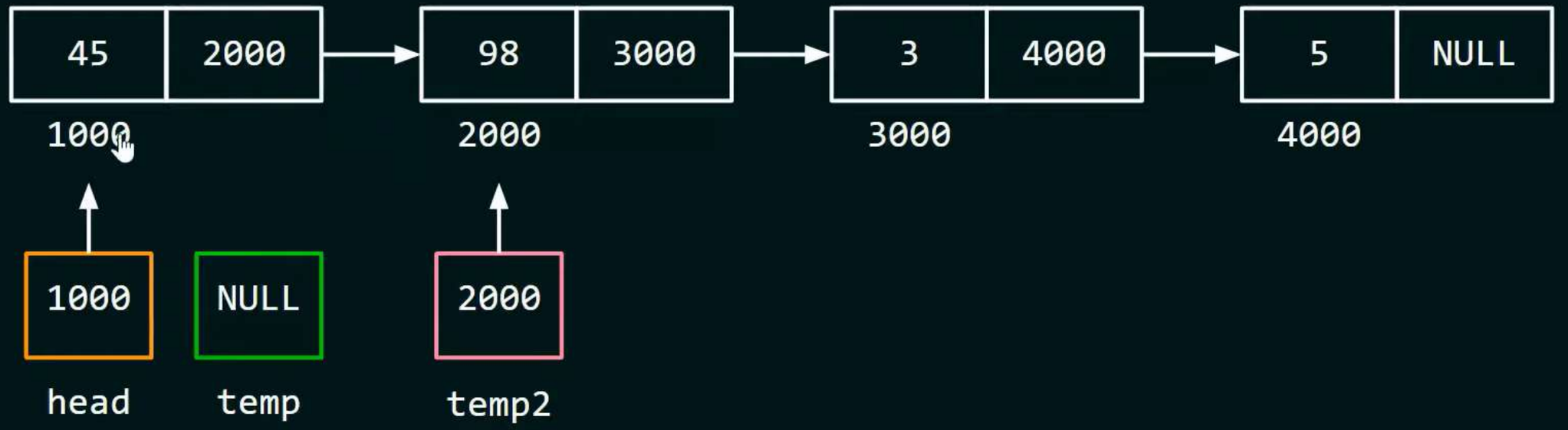

Let us suppose that initially
temp = NULL;
temp2 = NULL;



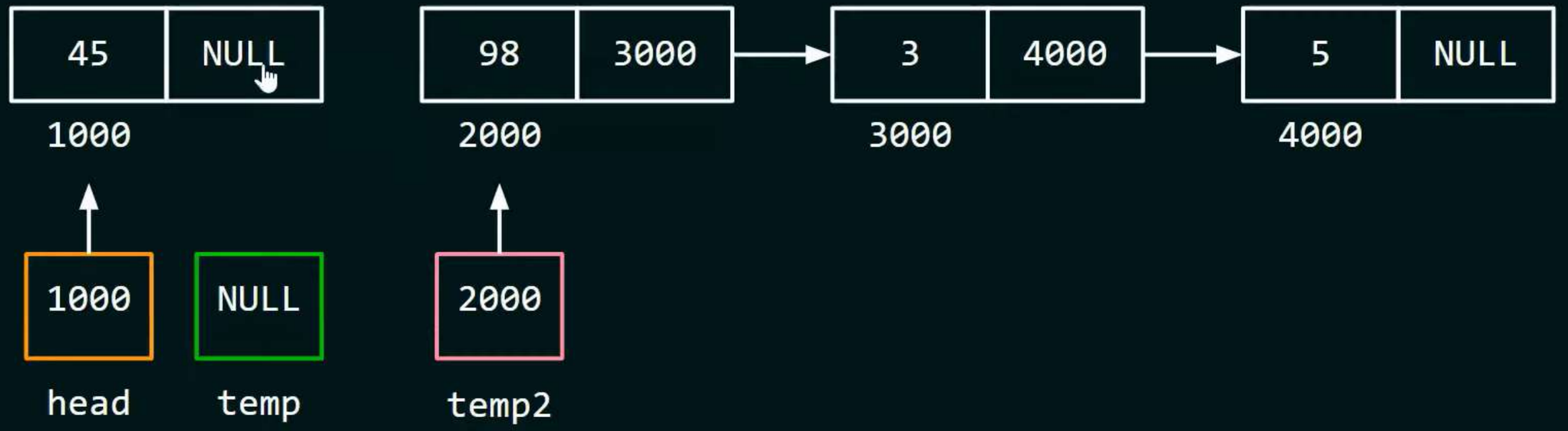

Our first target is to put NULL in the link part of the first node without losing the reference of the second node.



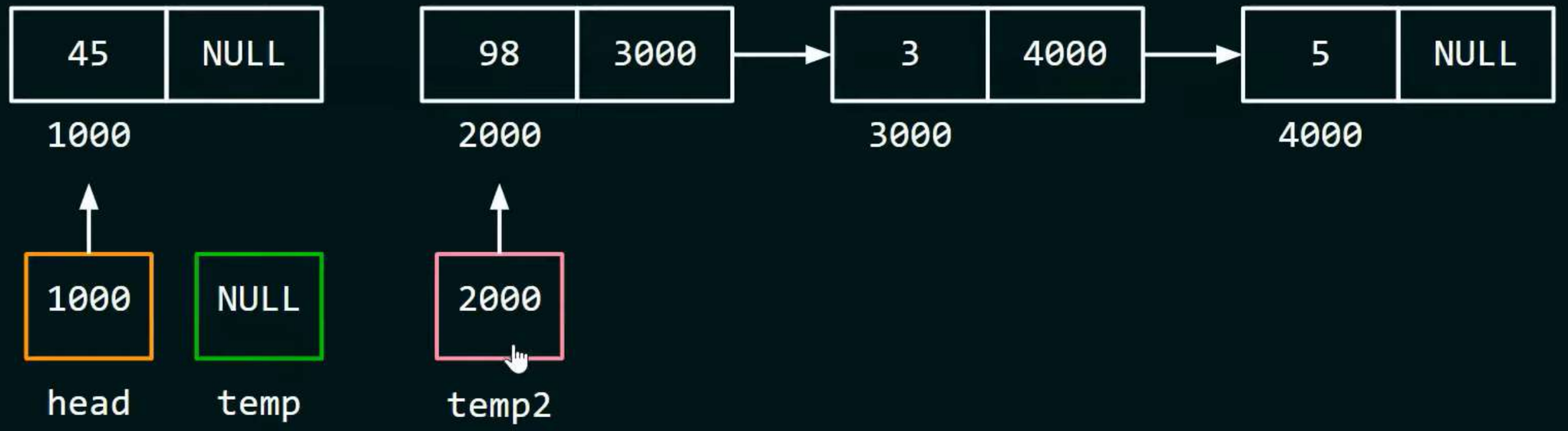

For this, we need to keep a pointer that will point to the second node of the list.
Let's keep temp2 for this purpose.



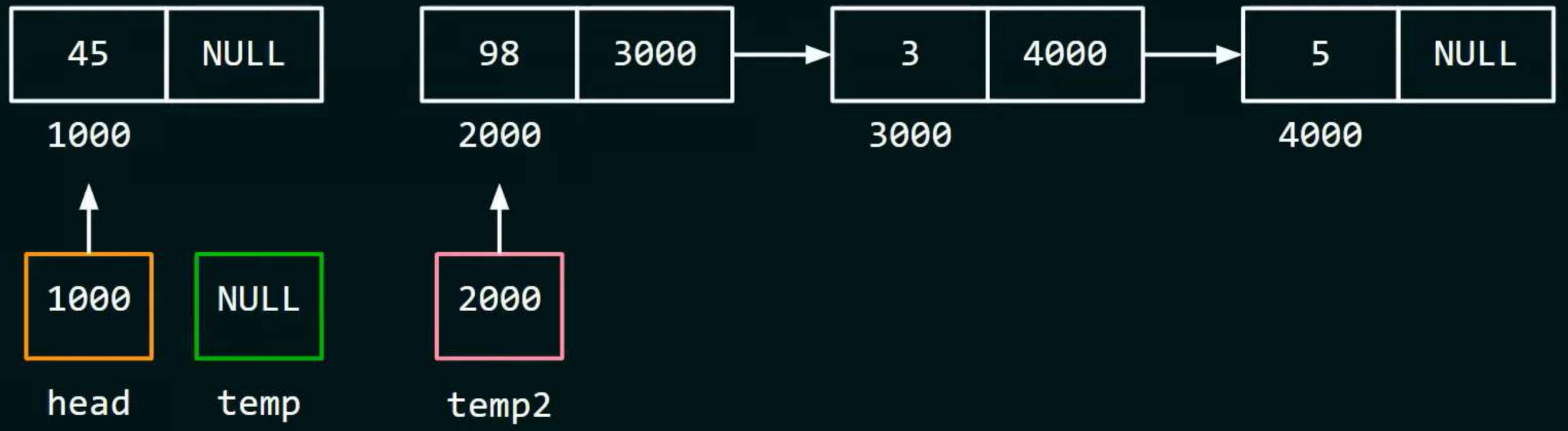

```
temp2 = head->link;
```



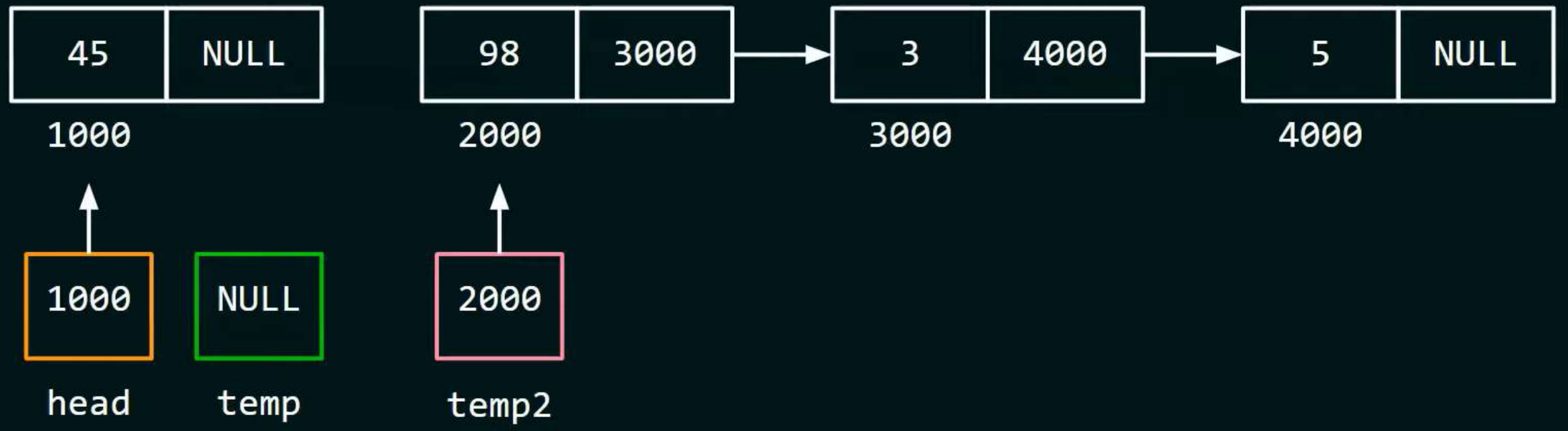

```
temp2 = head->link;
head->link = temp;
```



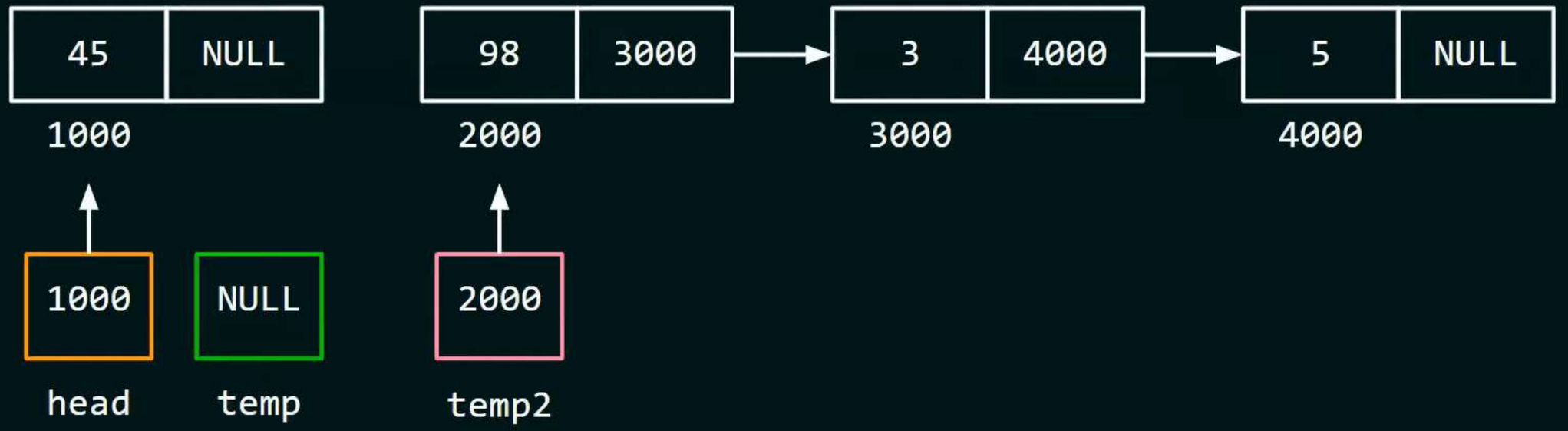

Now understand that we can't update the link part of the second node using temp2.



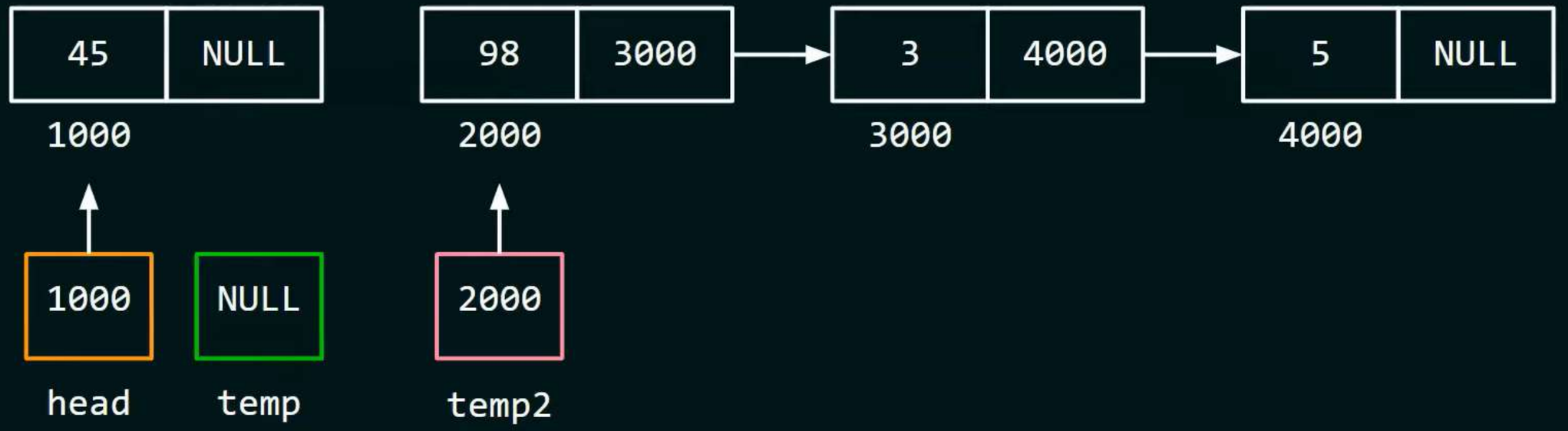

If we do this, we will lose the link to the third node.



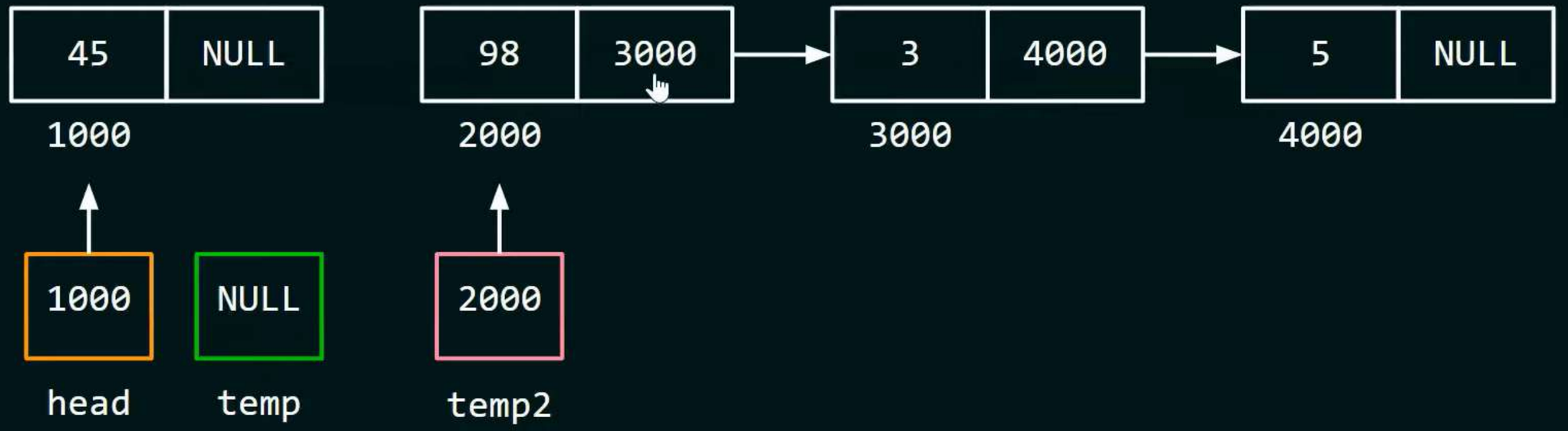

The idea is to update the link part of one node at a time which is pointed by the head pointer.



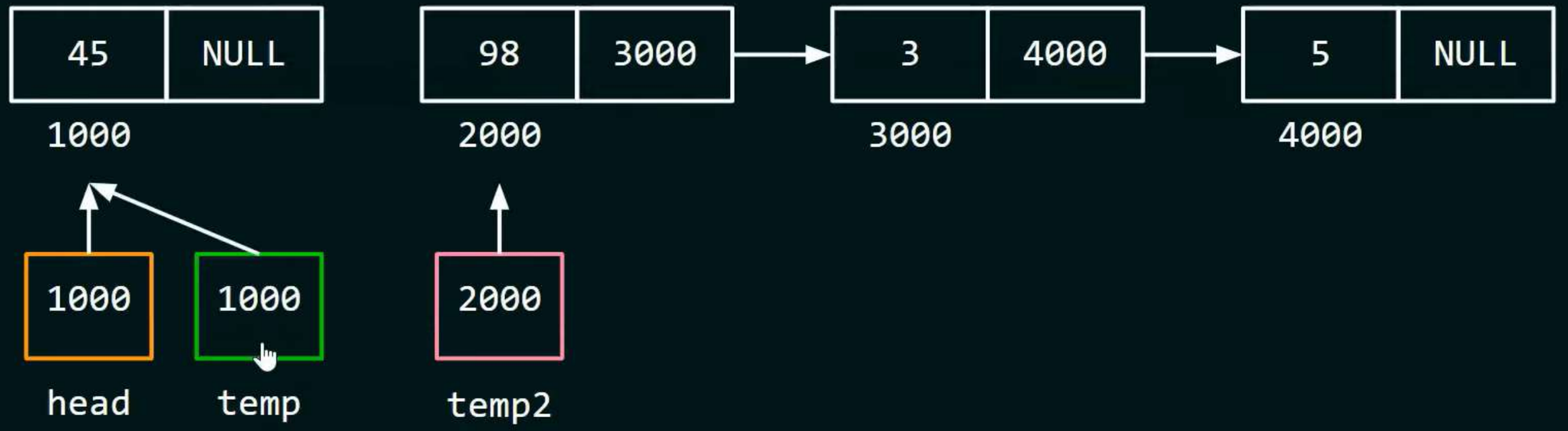

This means if we want to update the link part of the second node then we have to move our head pointer towards the second node of the list.



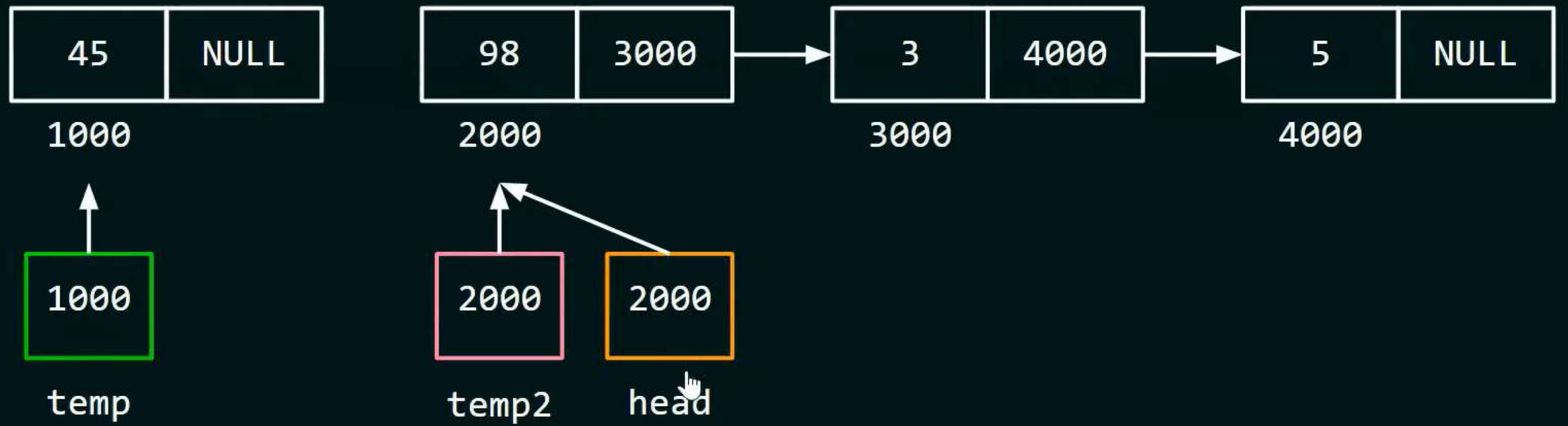

But before that, we have to keep a pointer that will point to the first node of the list.



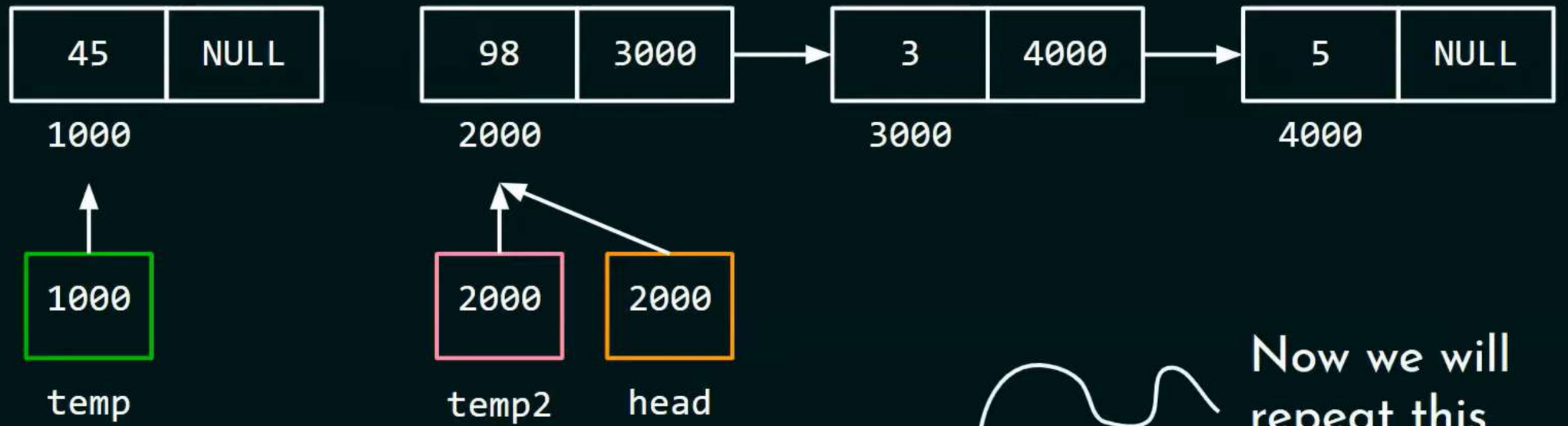

temp pointer will help us in updating the link part of the second node.




```
temp2 = head->link;
head->link = temp;
temp = head;
```

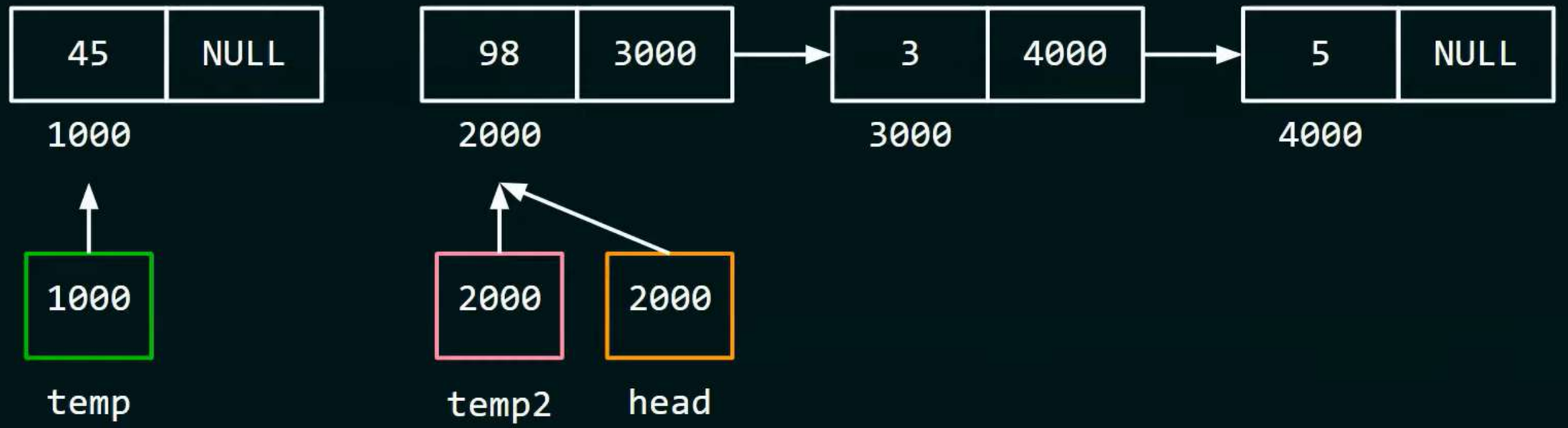



```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```

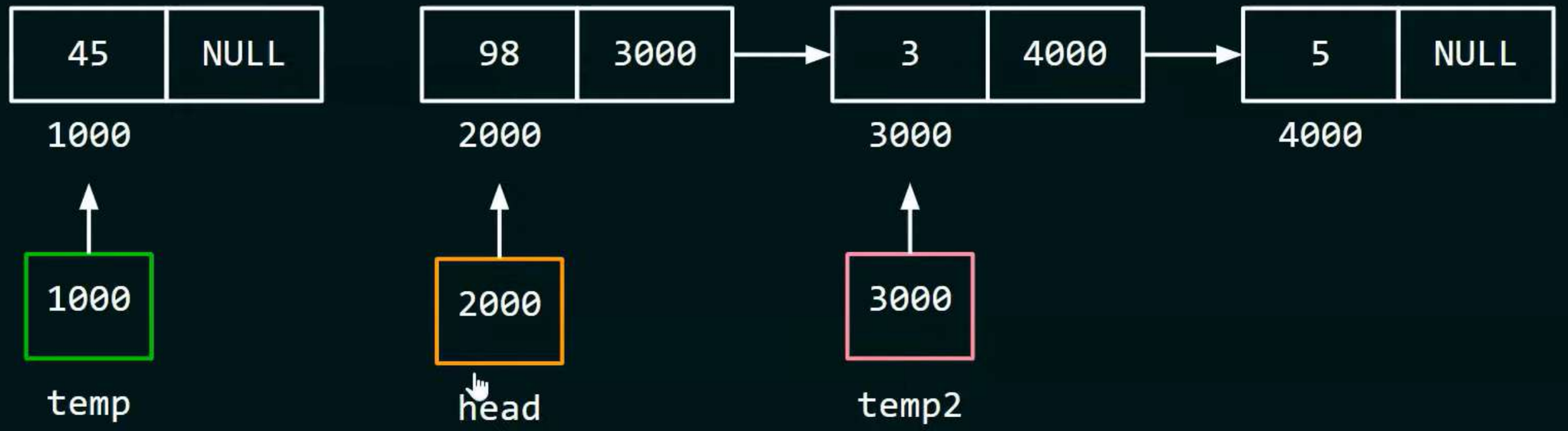



```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```

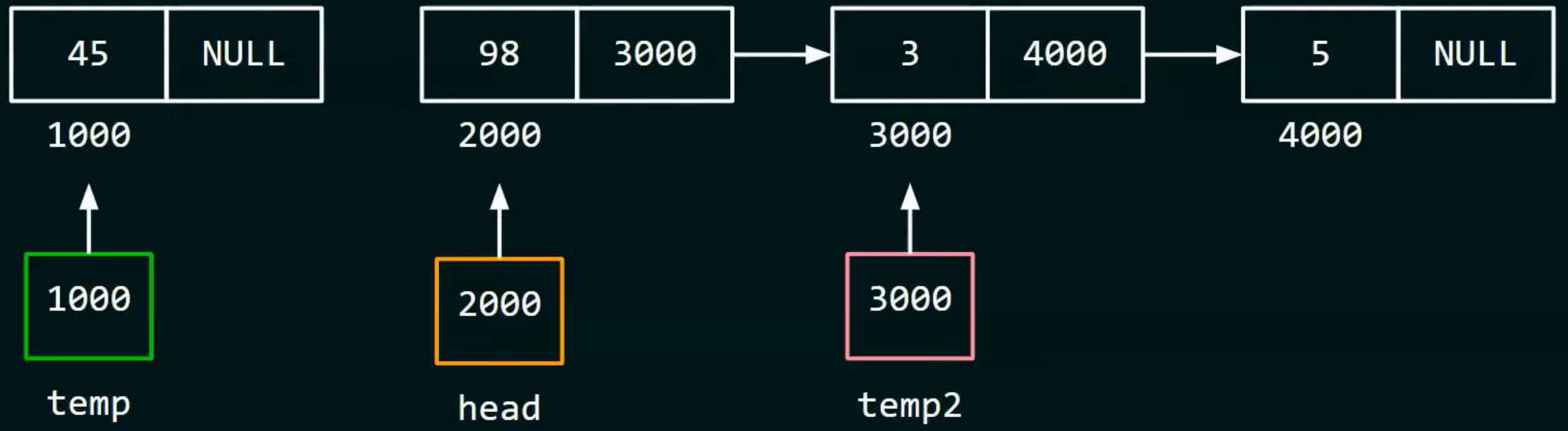
Now we will repeat this piece of code to update the linked part of the second node.



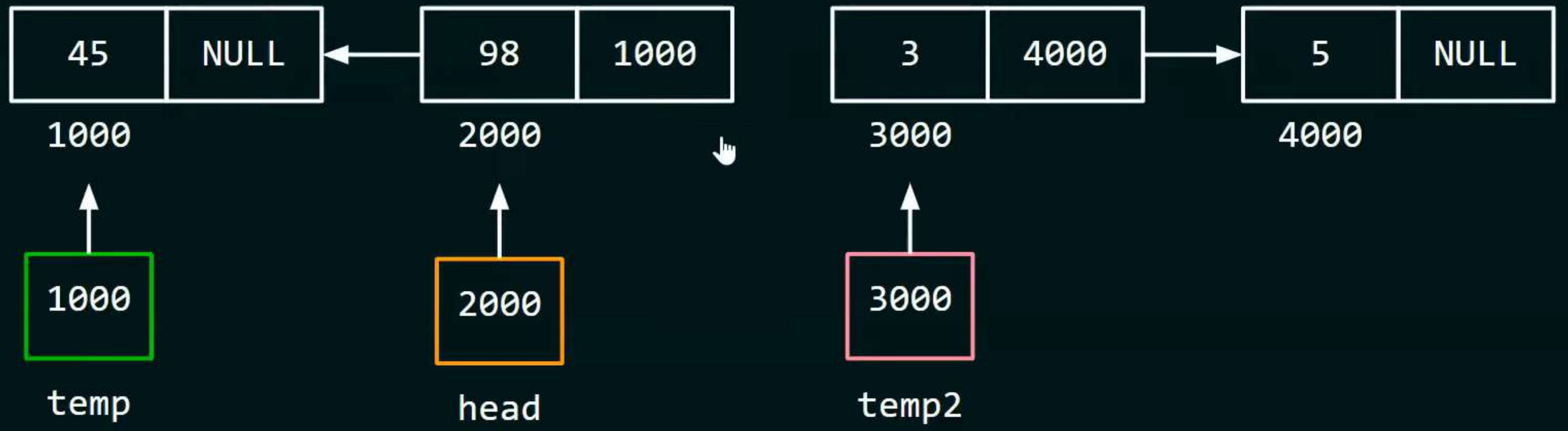

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



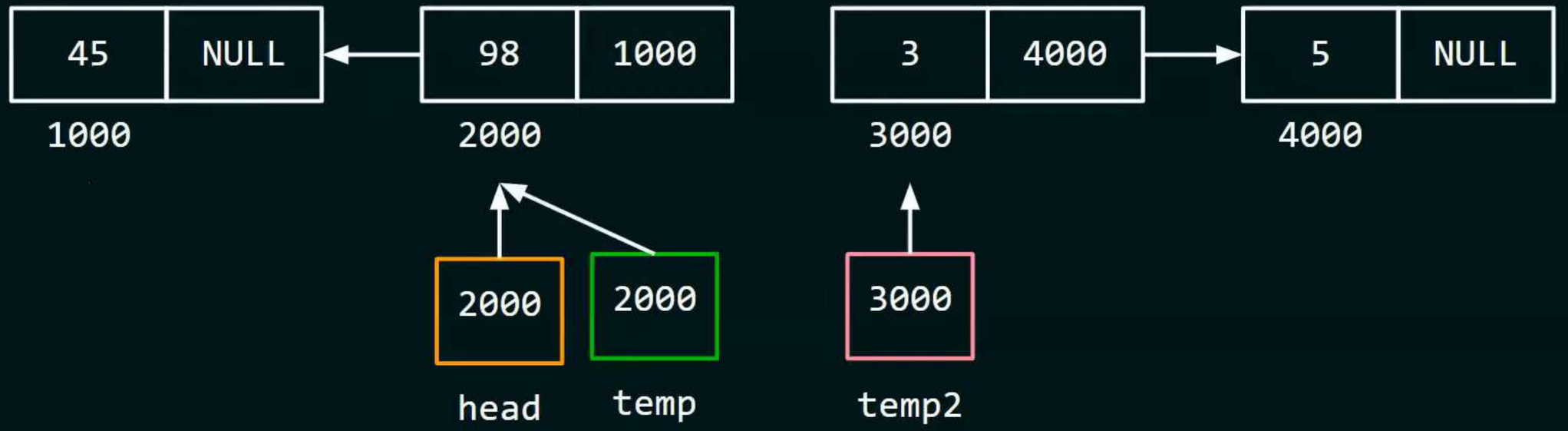

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



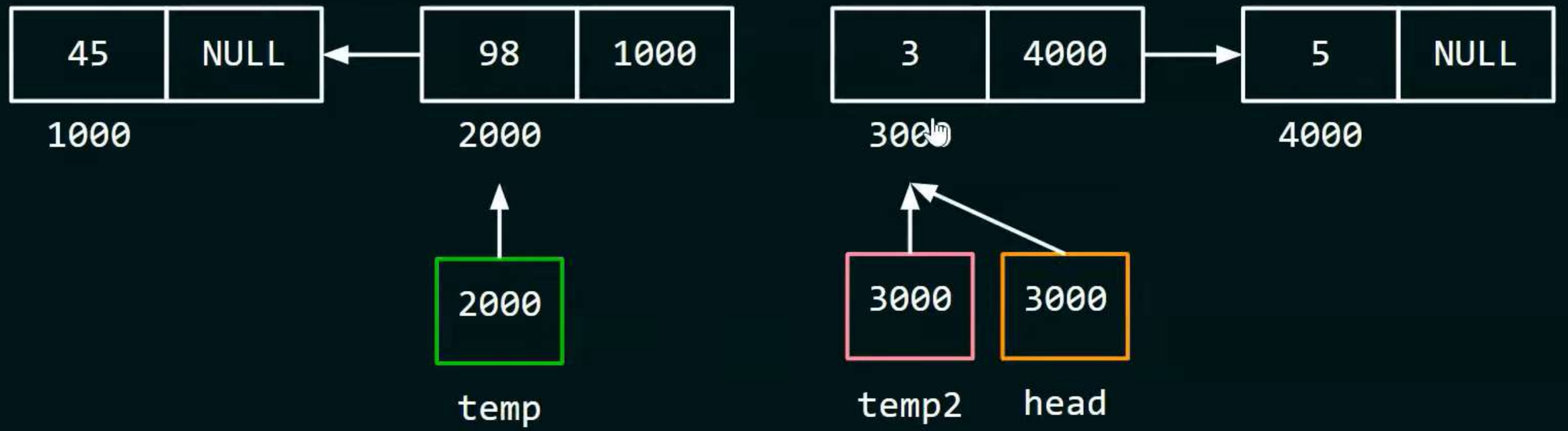

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



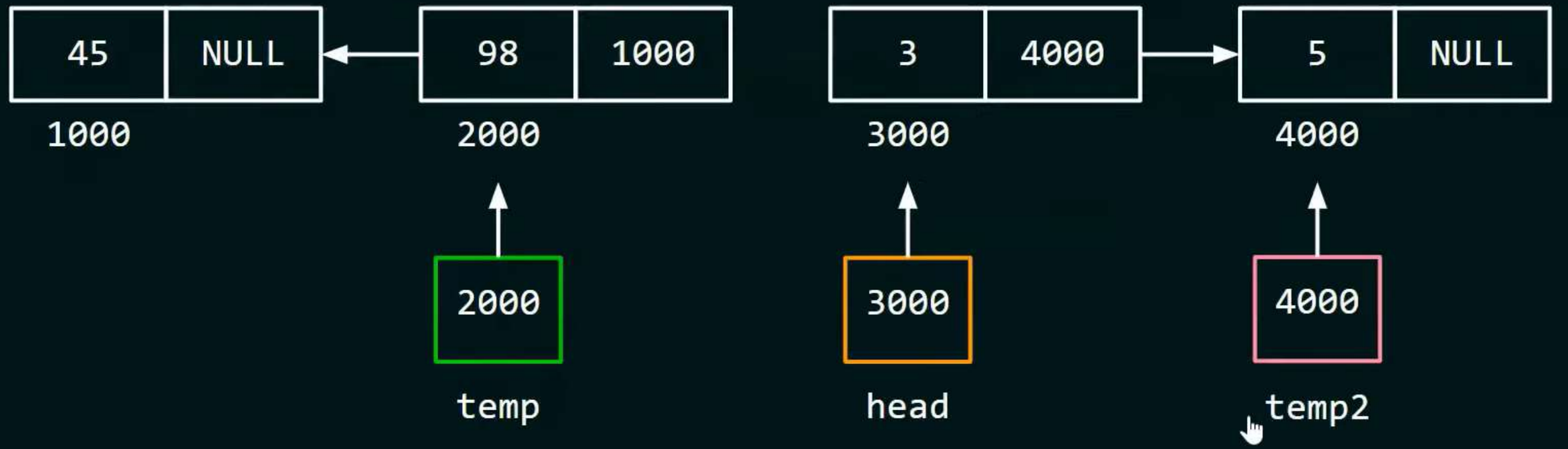

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



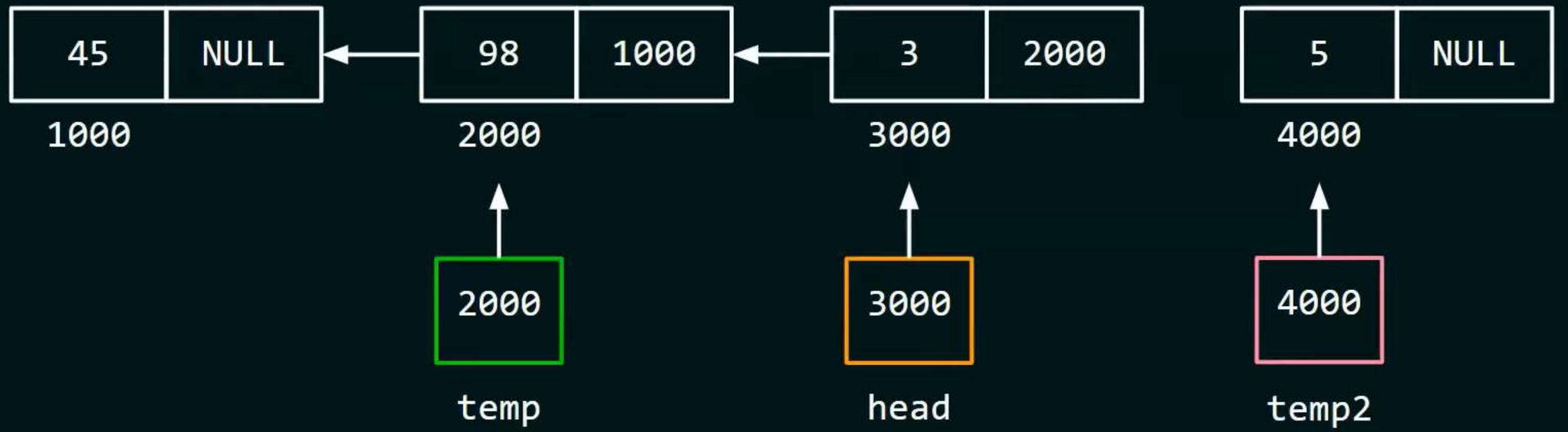

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



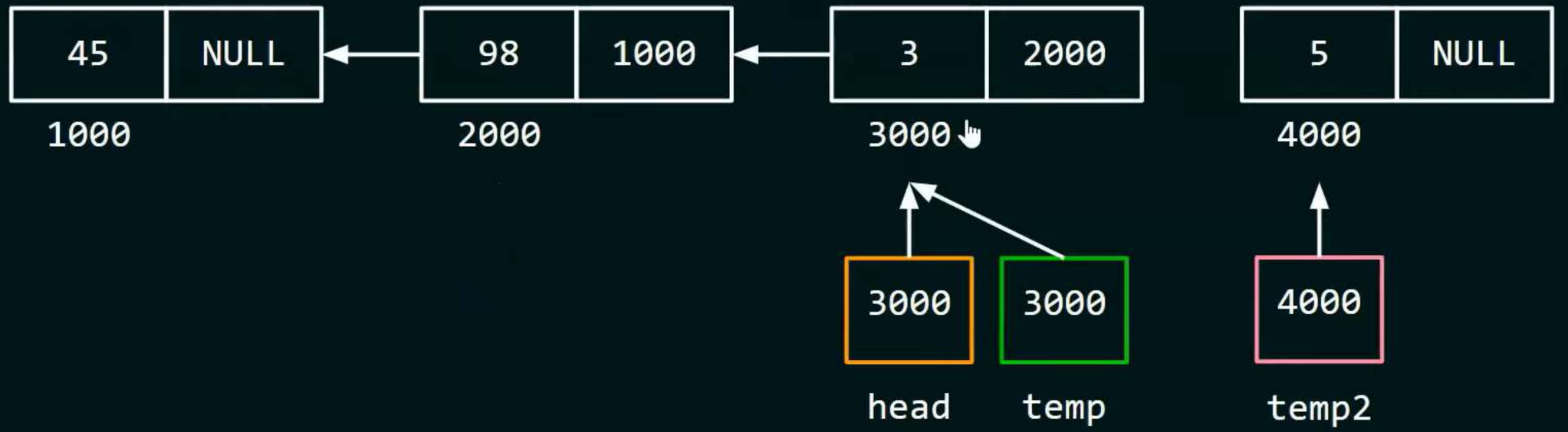

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



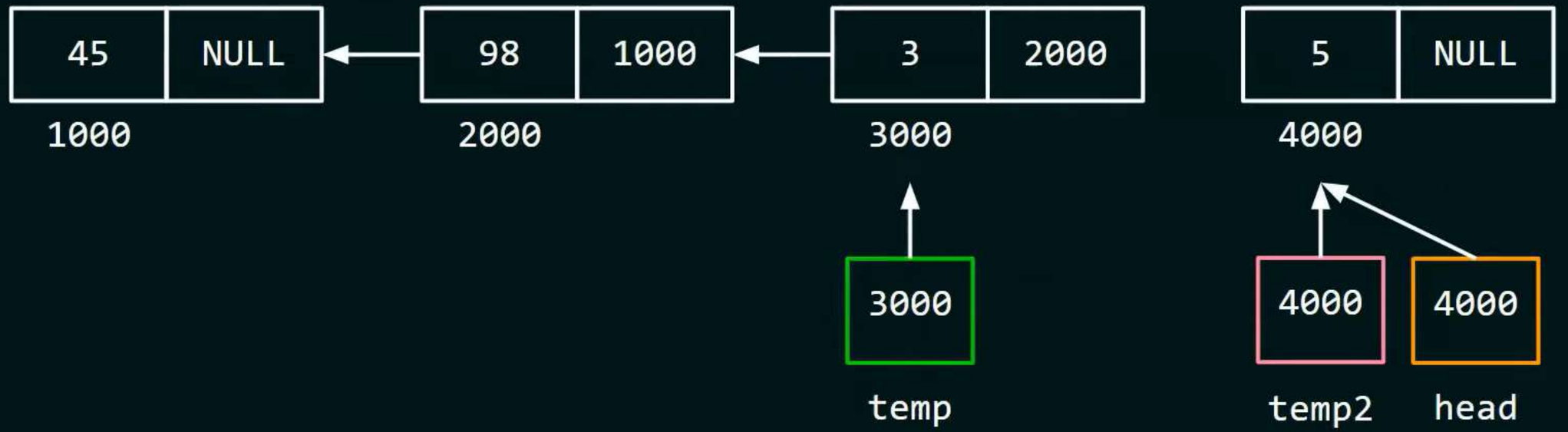

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



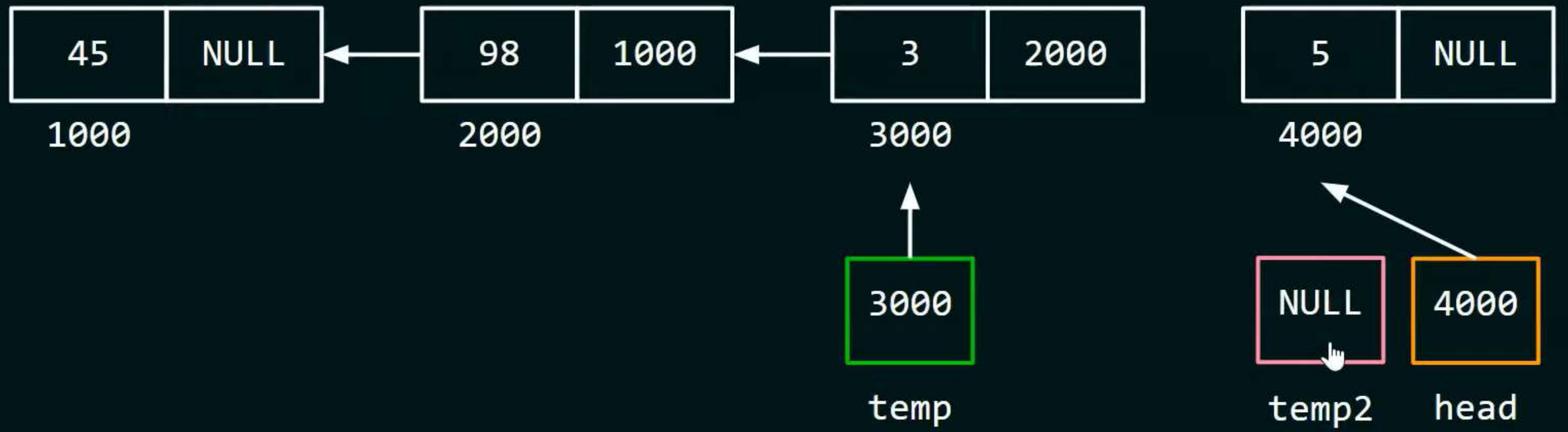

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



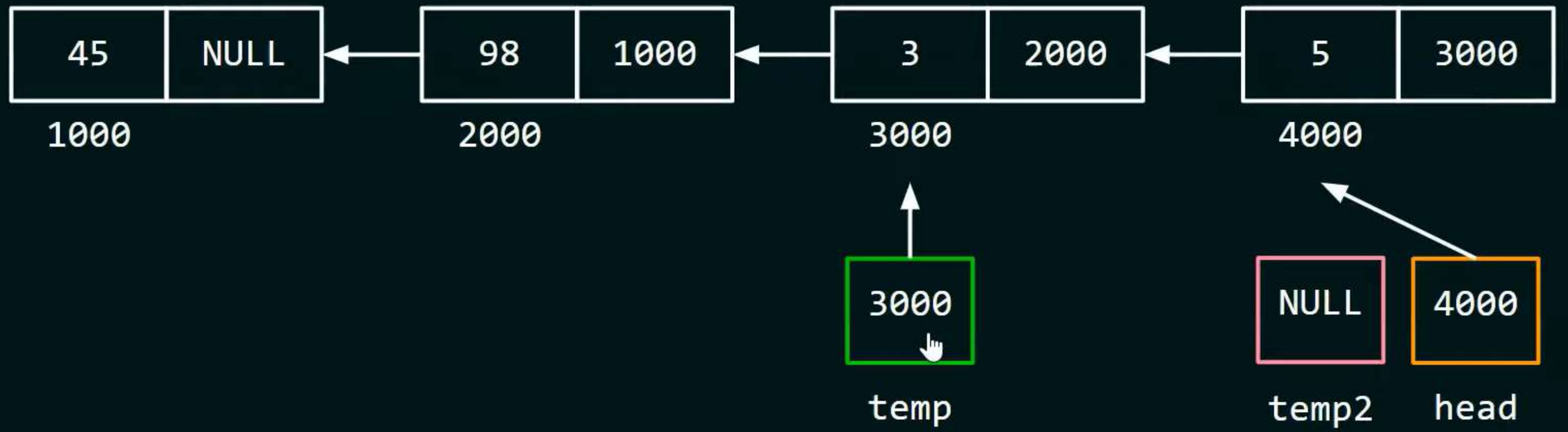

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



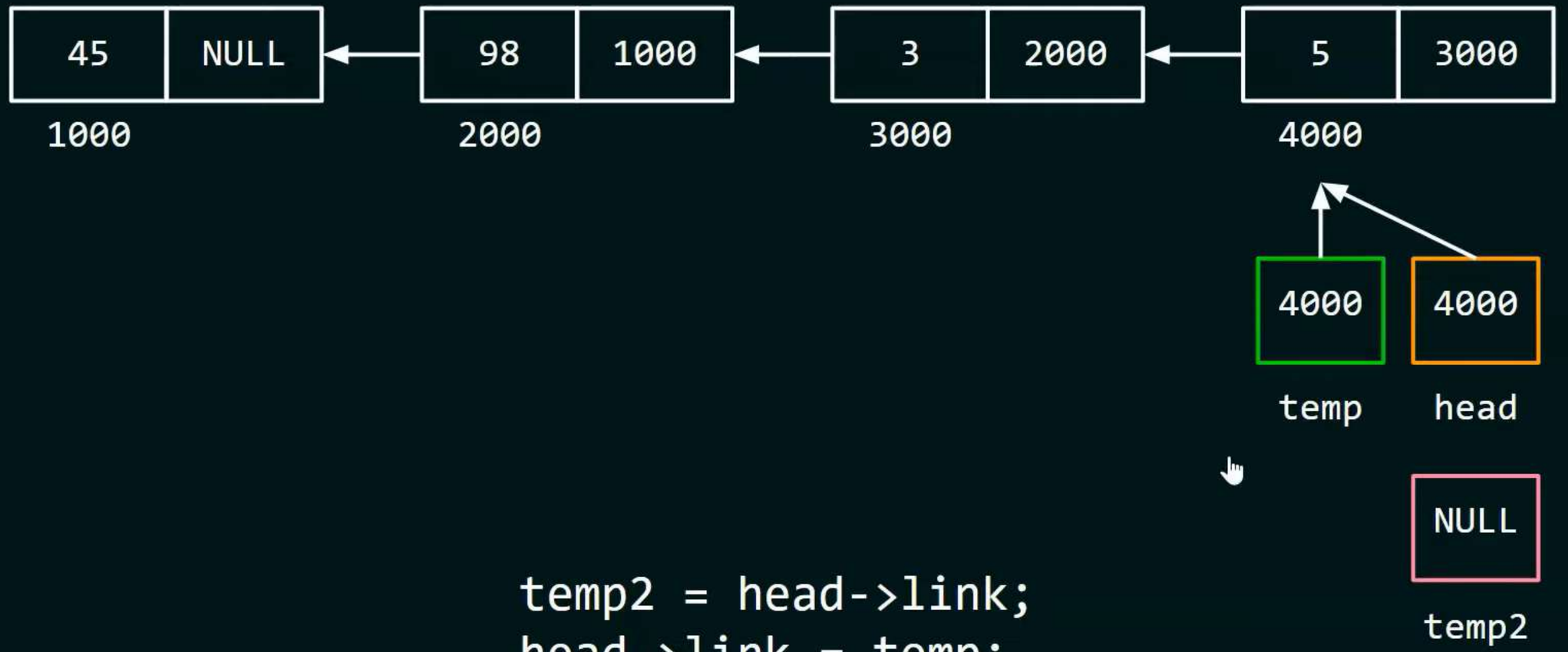

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



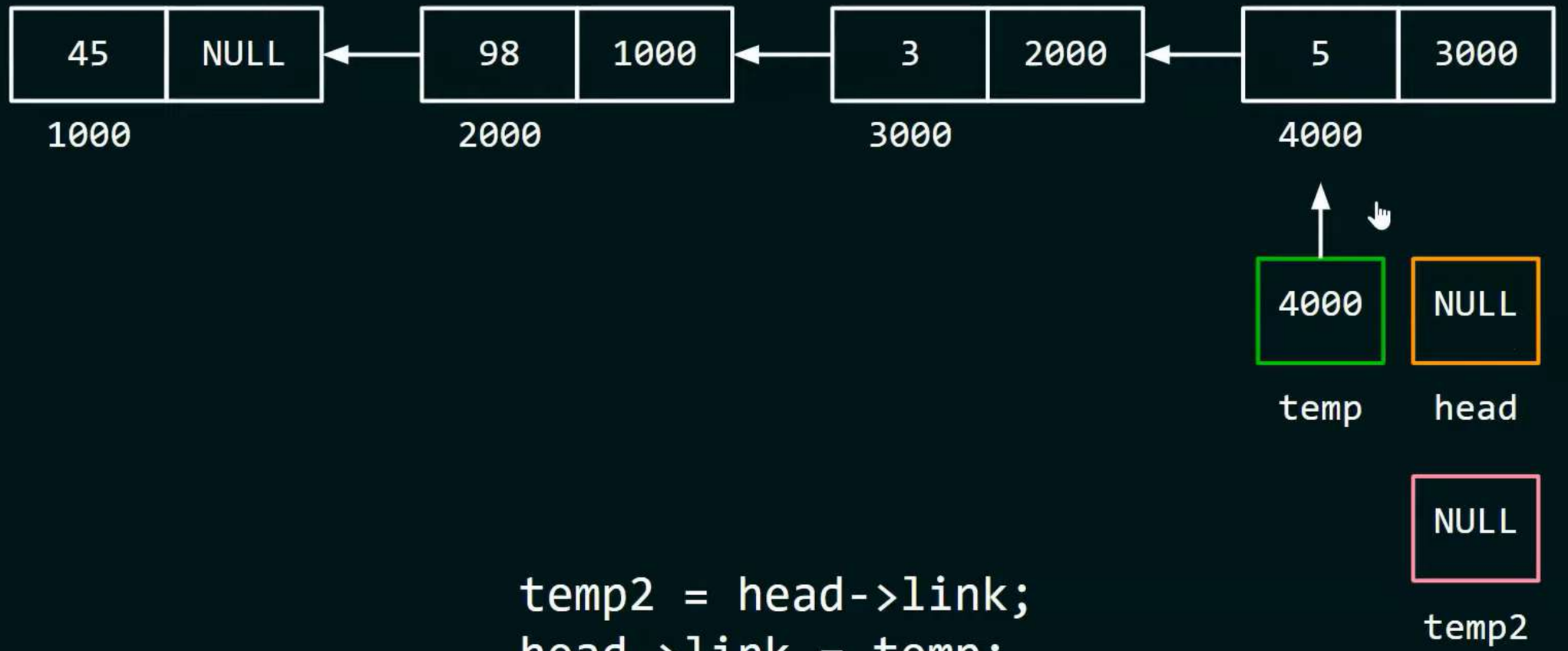

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```



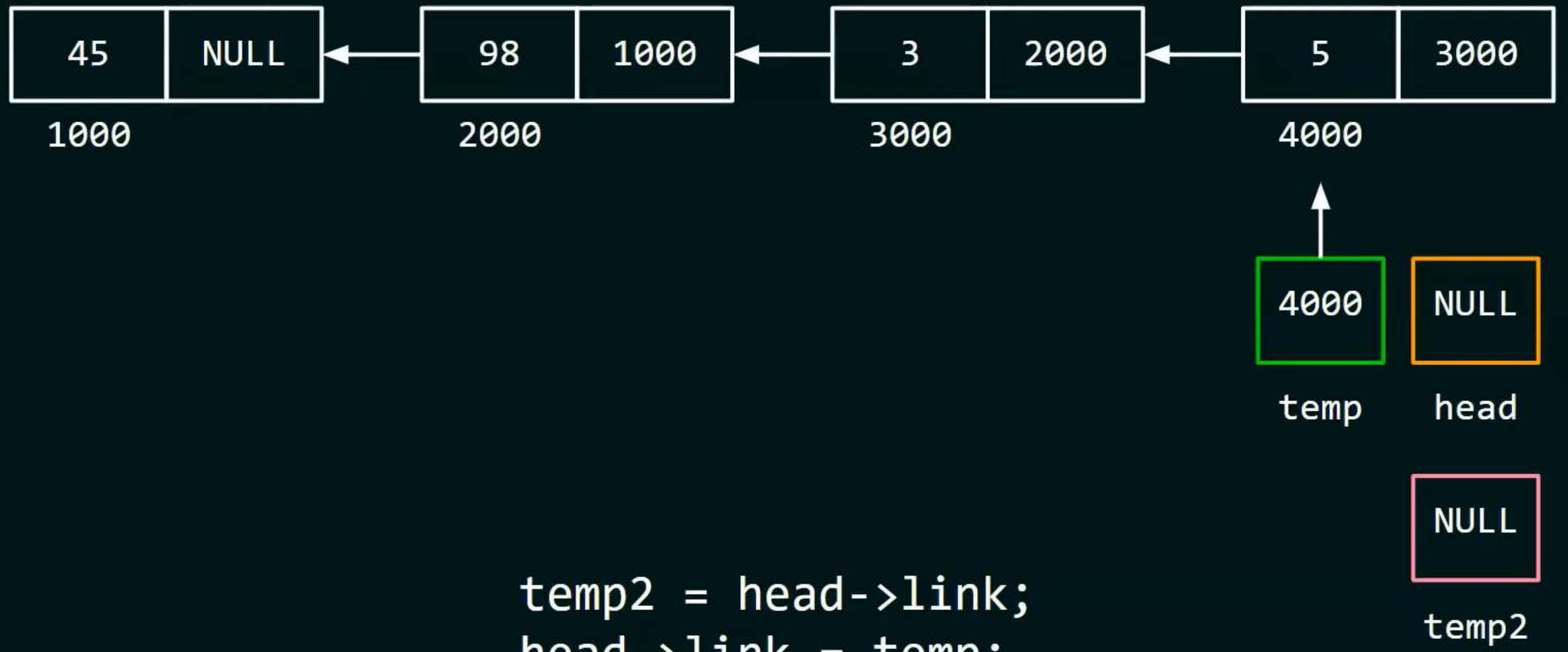

```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```




```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```

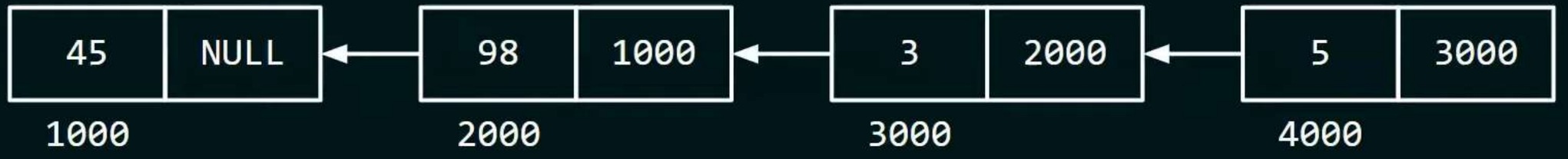



```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```

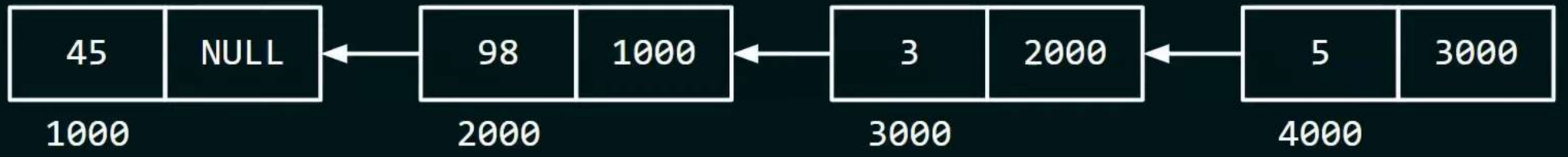



```
temp2 = head->link;
head->link = temp;
temp = head;
head = temp2;
```

We should stop the iteration when head becomes NULL.

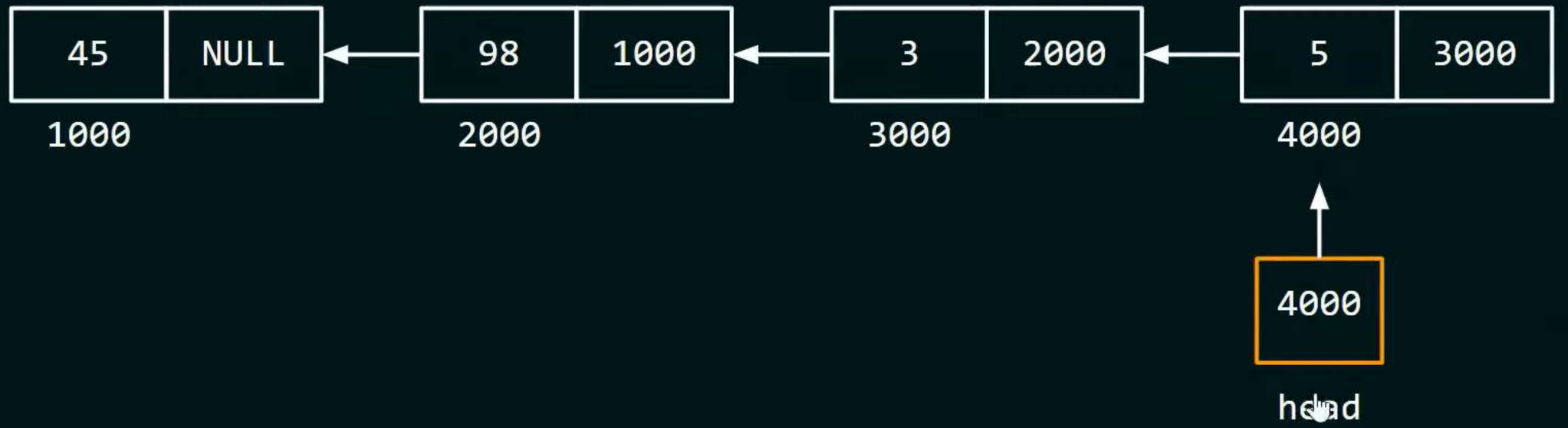



```
while(head != NULL)
{
    temp2 = head->link;
    head->link = temp;
    temp = head;
    head = temp2;
}
```



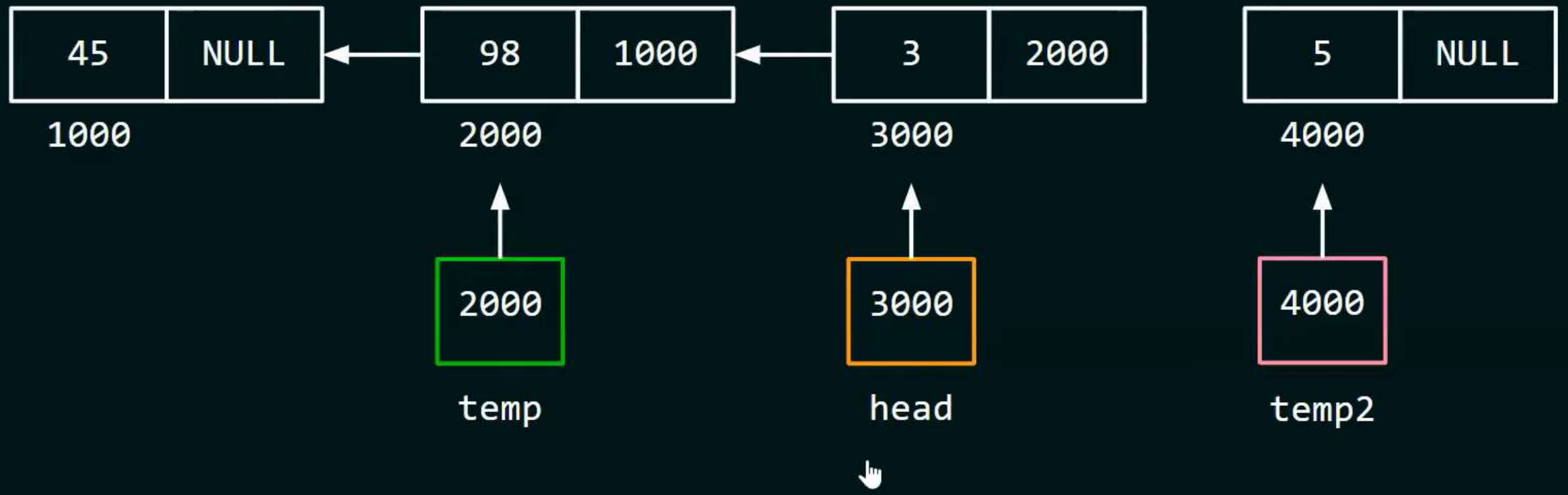


```
while(head != NULL)
{
    temp2 = head->link;
    head->link = temp;
    temp = head;
    head = temp2;
}

head = temp;
```

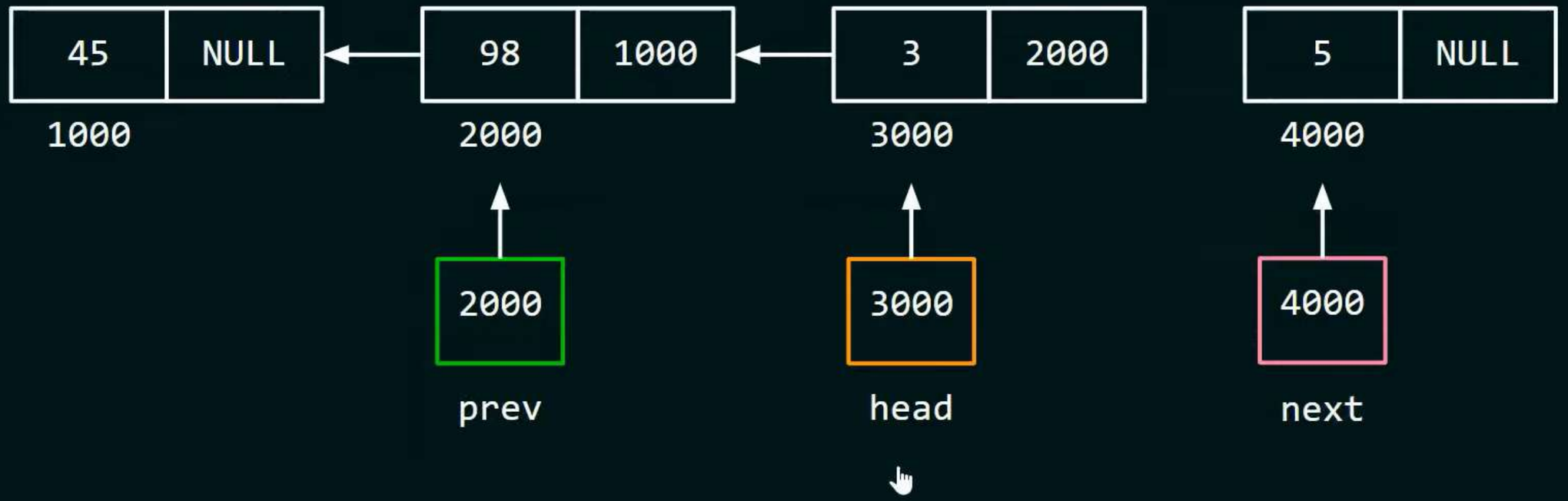



# Final Result

# Let's rename our pointers.



# Let's rename our pointers.



# Program

```c
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node* link;
};

int main() {
     head = reverse(head);
     ptr = head;
     while(ptr != NULL)
     {
          printf("%d ", ptr->data);
          ptr = ptr->link;
     }
     return 0;
}
```

```c
struct node* reverse(struct node *head)
{
     struct node *prev = NULL;
     struct node *next = NULL;
     while(head != NULL)
     {
          next = head->link;
          head->link = prev;
          prev = head;
          head = next;
     }
     head = prev;
     return head;
}
```

# Program

```
#include <stdio.h>
#include <stdlib.h>

struct node {
    int data;
    struct node* link;
};

int main() {
     head = reverse(head);
     ptr = head;
     while(ptr != NULL)
     {
          printf("%d ", ptr->data);
          ptr = ptr->link;
     }
     return 0;
}
```

```
struct node* reverse(struct node *head)
{
     struct node *prev = NULL;
     struct node *next = NULL;
     while(head != NULL)
     {
          next = head->link;
          head->link = prev;
          prev = head;
          head = next;
     }
     head = prev;
     return head;
}
```

Output: 3 98 45