DATA Structures

Dr. Asmaa Gad El-Kareem

Lecture #4

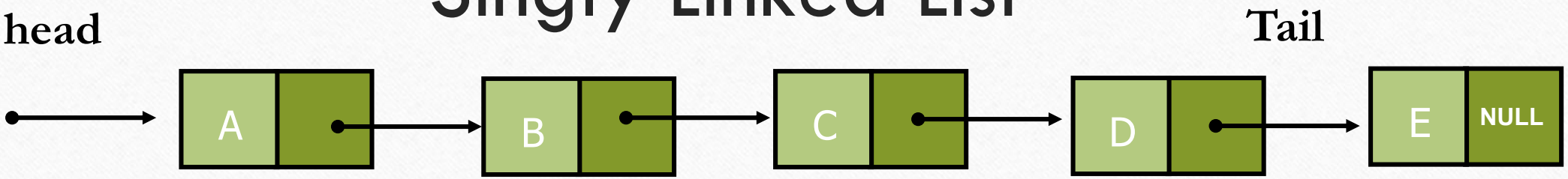
Data Structures

Lecture 4: Circular and Doubly Linked Lists

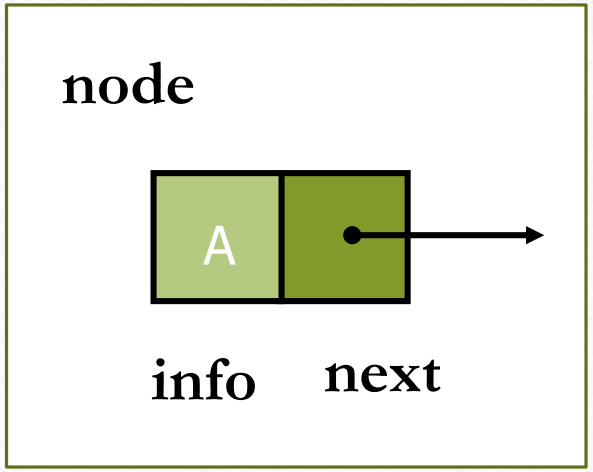
Overview

- **Variations of linked lists**
  - Circular linked lists
  - Doubly linked lists
- **Circular Linked lists**
- **Basic operations of Circular linked lists**
  - Insert, find, delete, print, etc.
- **Doubly Linked lists**
- **Basic operations of Doubly linked lists**
  - Insert, find, delete, print, etc.

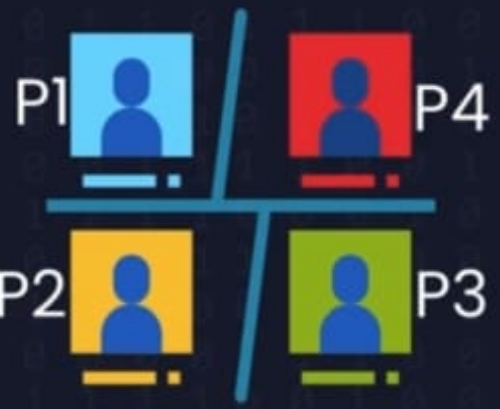
Singly Linked List

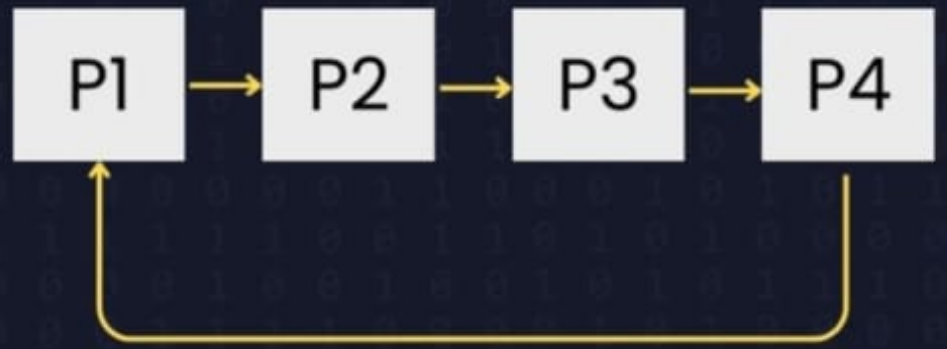


- A **linked list** is a series of connected nodes that form a linear sequence
- Each **node** contains two fields:
  - A piece of data being stored (any type) →info
  - Pointer to the next node in the list →next
- **Head**: pointer to the first node
- **Tail**: pointer to the last node
- The last node points to **NULL(address 0)**



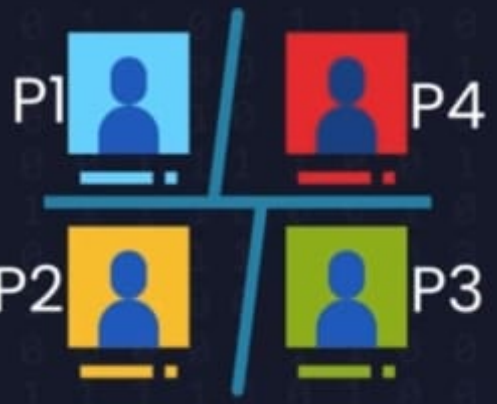
Data Structure

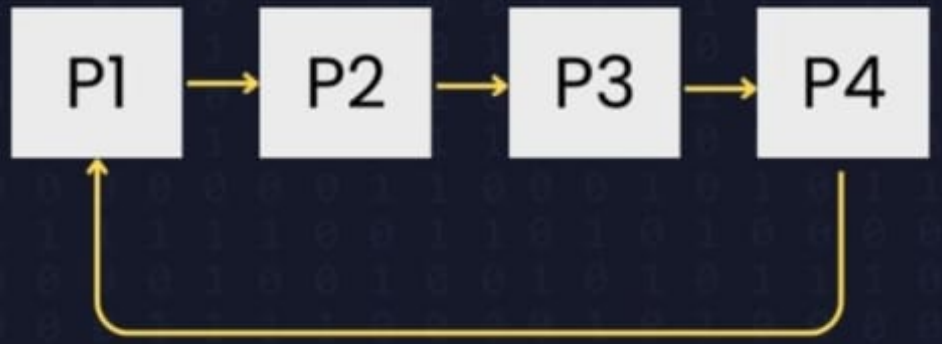


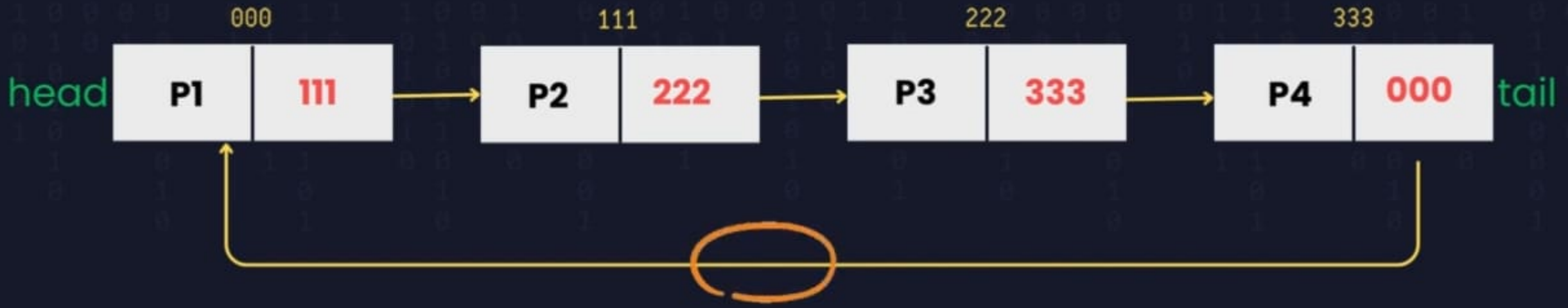


Data Structure

Circular Single LinkedList







Circular Linked Lists

- **There are many applications**

- which data can be more naturally viewed as having a cyclic order
- no fixed beginning or end

- **Many multiplayer games are turn-based**

- with player A taking a turn
- then player B
- then player C, and so on,
- but eventually back to player A again, and player B again, with the pattern repeating

Variations of Linked Lists

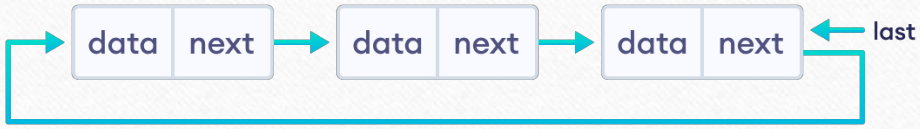
Circular Linked Lists

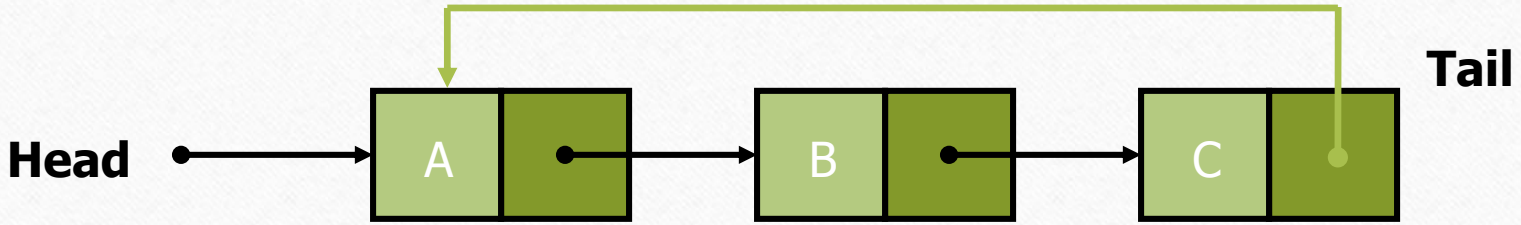
- A circular linked list is a type of linked list in which the first and the last nodes are also connected to each other to form a circle.

There are basically two types of circular linked list:

1. Circular Singly Linked List
2. Circular Doubly Linked List

Circular singly linked lists





- It is essentially a singly linked list in which the the next reference of the tail node set to refer back to the head of the list (rather than null).
- How do we know when we have finished traversing the list?
  - check if the pointer of the current node is equal to the head.

Circular doubly linked lists

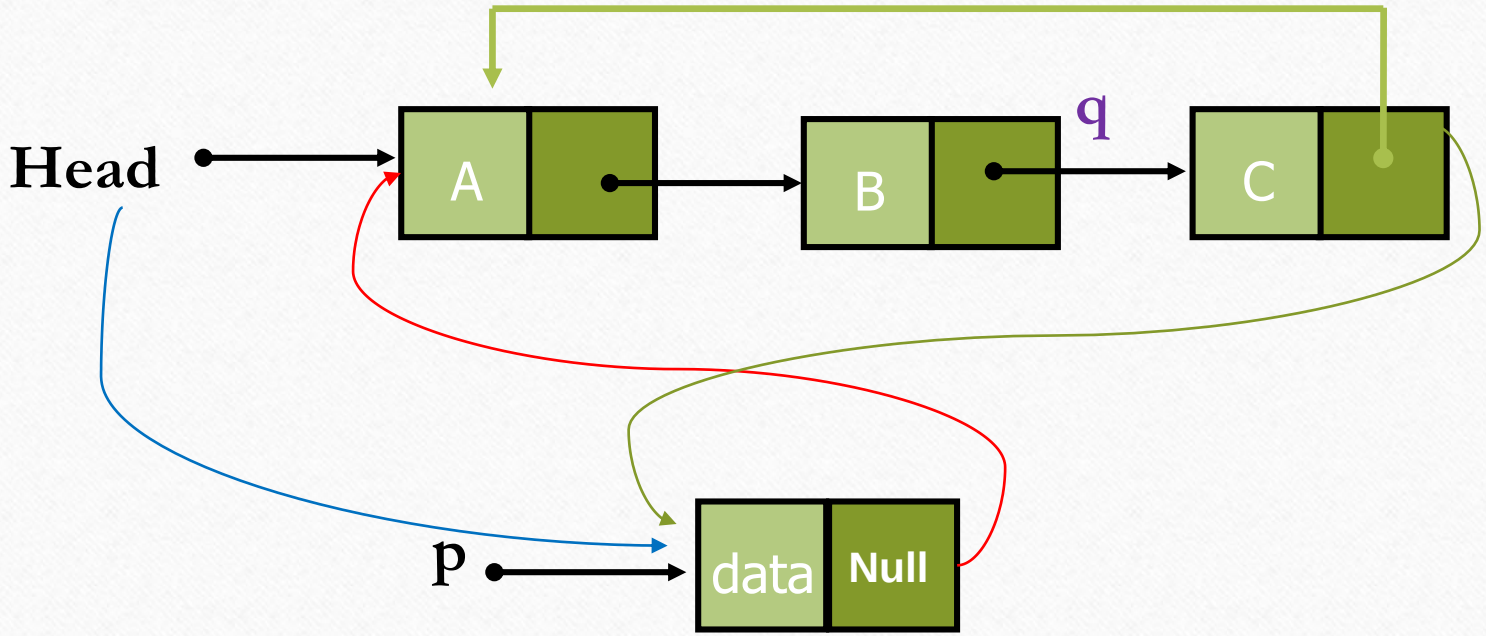
- Here, in addition to the last node storing the address of the first node, the first node will also store the address of the last node.
- **Note:** We will be using the circular singly linked list to represent the working of circular linked list.

AddToHead Function

- To insert a node at the front of the list (this list is used to store integer data):

1. An empty node is created.
2. The node's info member is initialized to a particular integer.
3. Because the node is being included at the front of the list, the next member becomes a pointer to the first node on the list; that is, the current value of head.
4. The new node precedes all the nodes on the list, but this fact has to be reflected in the value of head; otherwise, the new node is not accessible. Therefore, head is updated to become the pointer to the new node .
5. **The next member of the last node is updated to point to the first node of the list (the new node)**

AddToHead Function



AddToHead Function

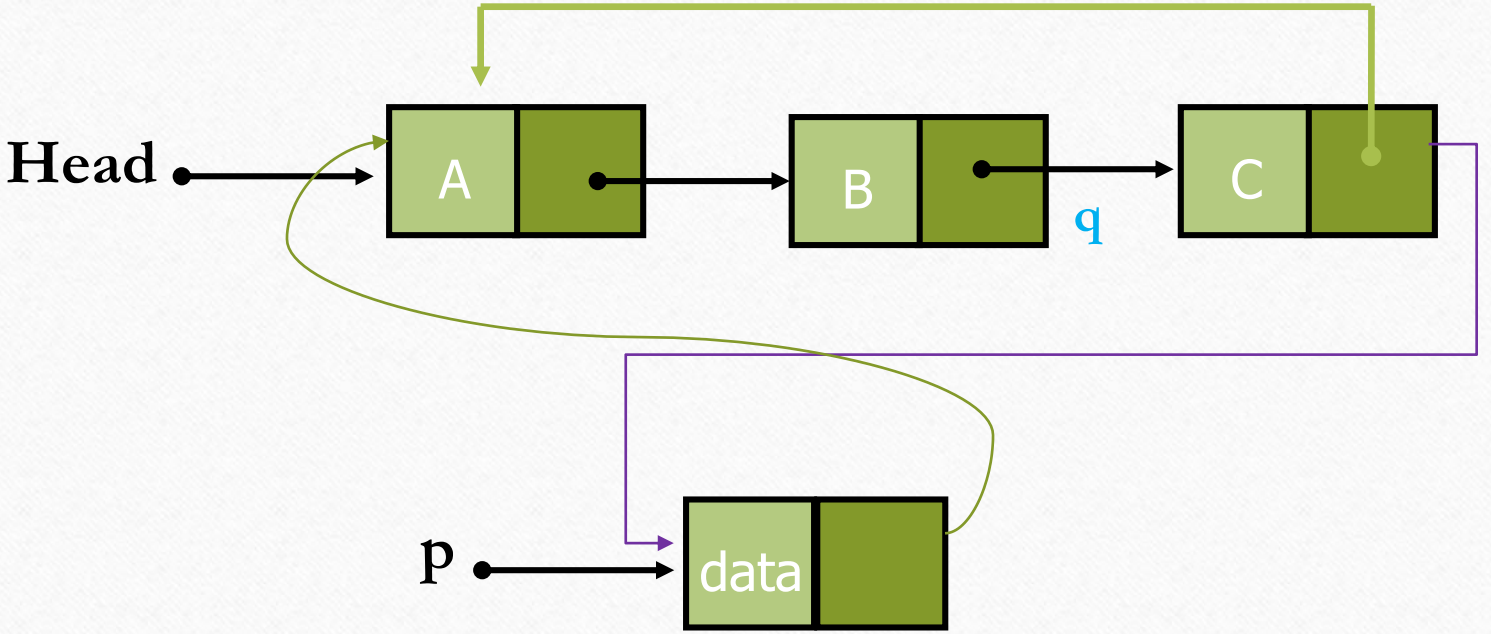
```
void circularList::addToHead(int data)
{
        node* p= new node;
        p->info=data;
        p->next=head;
        if(head==NULL)
        {
                head=p;
                head->next=head;
        }
        else
        {
                node*q=head;
                while(q->next!=head)
                        q=q->next;
                q->next=p;
                head=p;
        }
}
```

AddToTail Function

- To insert a node at the end of the list:

1. An empty node is created.
2. The node's info member is initialized to an integer.
3. Because the node is being included at the end of the list, the **next member is set to head.**
4. The node is now included in the list by making the next member of the last node of the list a pointer to the **newly created node.**

AddToTail Function



AddToTail Function

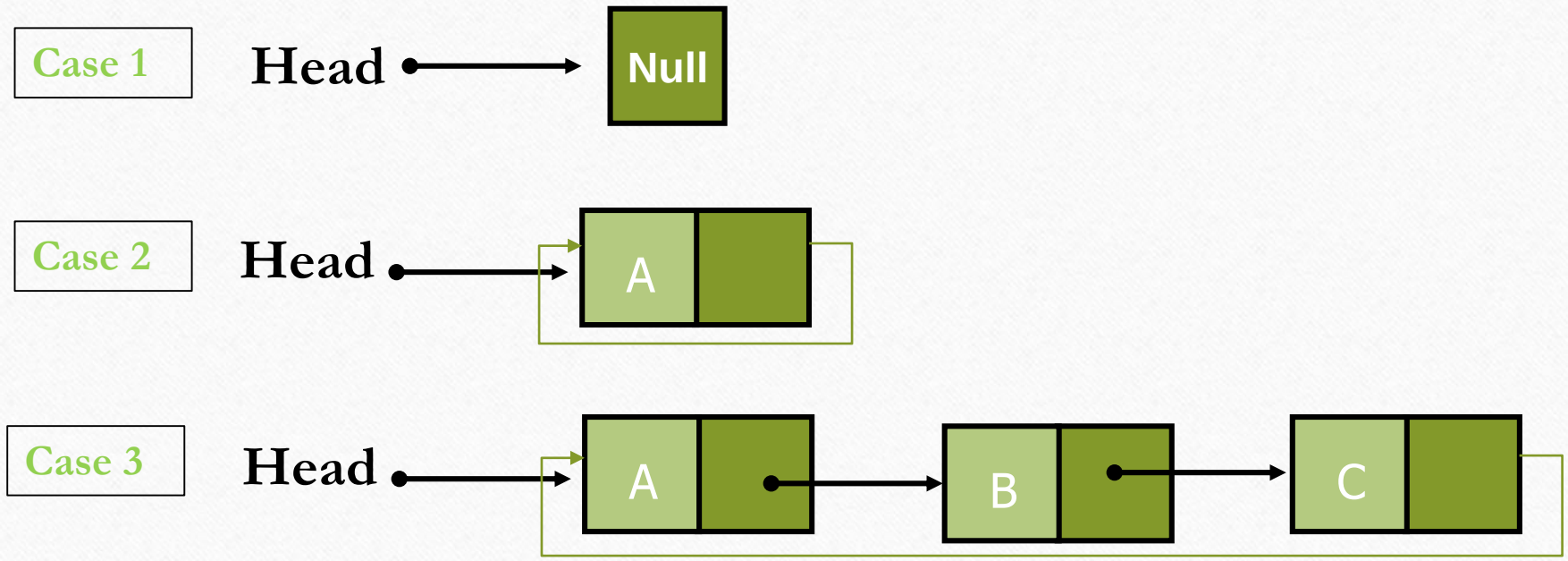
```
void circularList::addToTail(int data)
{
        node* p= new node;
        p->info=data;
        p->next=head;
        if(head==NULL)
        {
                head=p;
                head->next=head;
        }
        else
        {
                node*q=head;
                while(q->next!=head)
                        q=q->next;
                q->next=p;
        }
}
```

Deleting a node from a Circular List

When deleting a node from a list, we need to consider all the following cases:

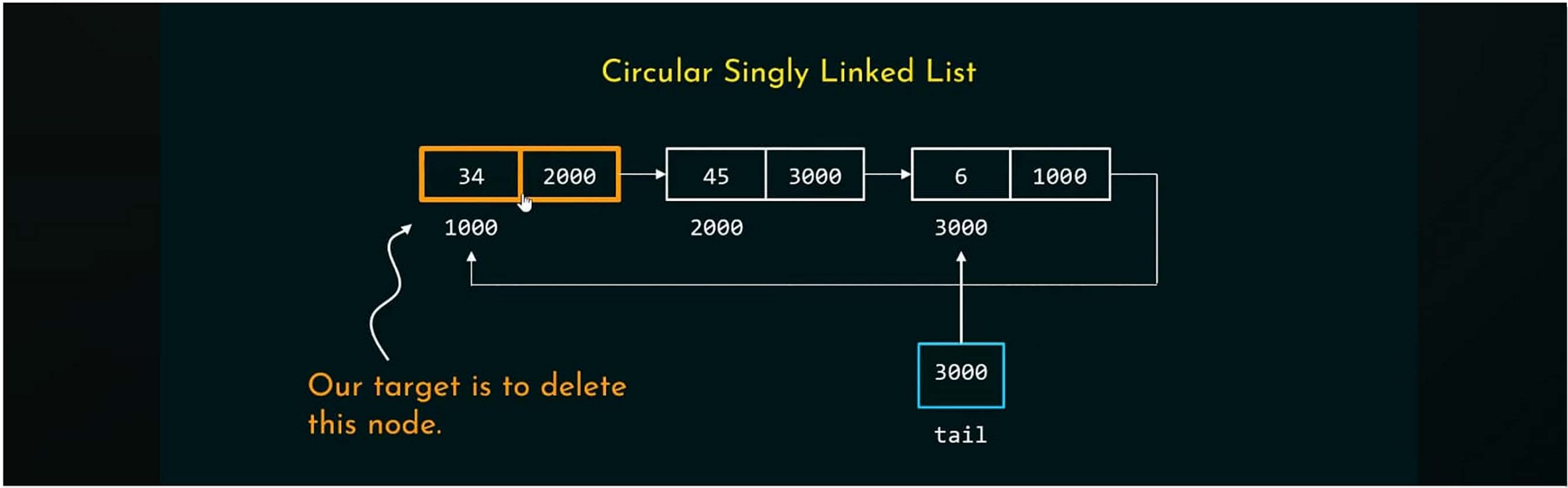
- Case 1: deleteing a node from an empty list → the function is immediately exited.
- Case 2: deleting the only node from a one-node linked list → the head is set to null.
- Case 3: deleting the first node of the list with at least two nodes → requires updating head.
- Case 4: deleting a node with a number that is not in the list → do nothing.

Deleting a node from a Circular List

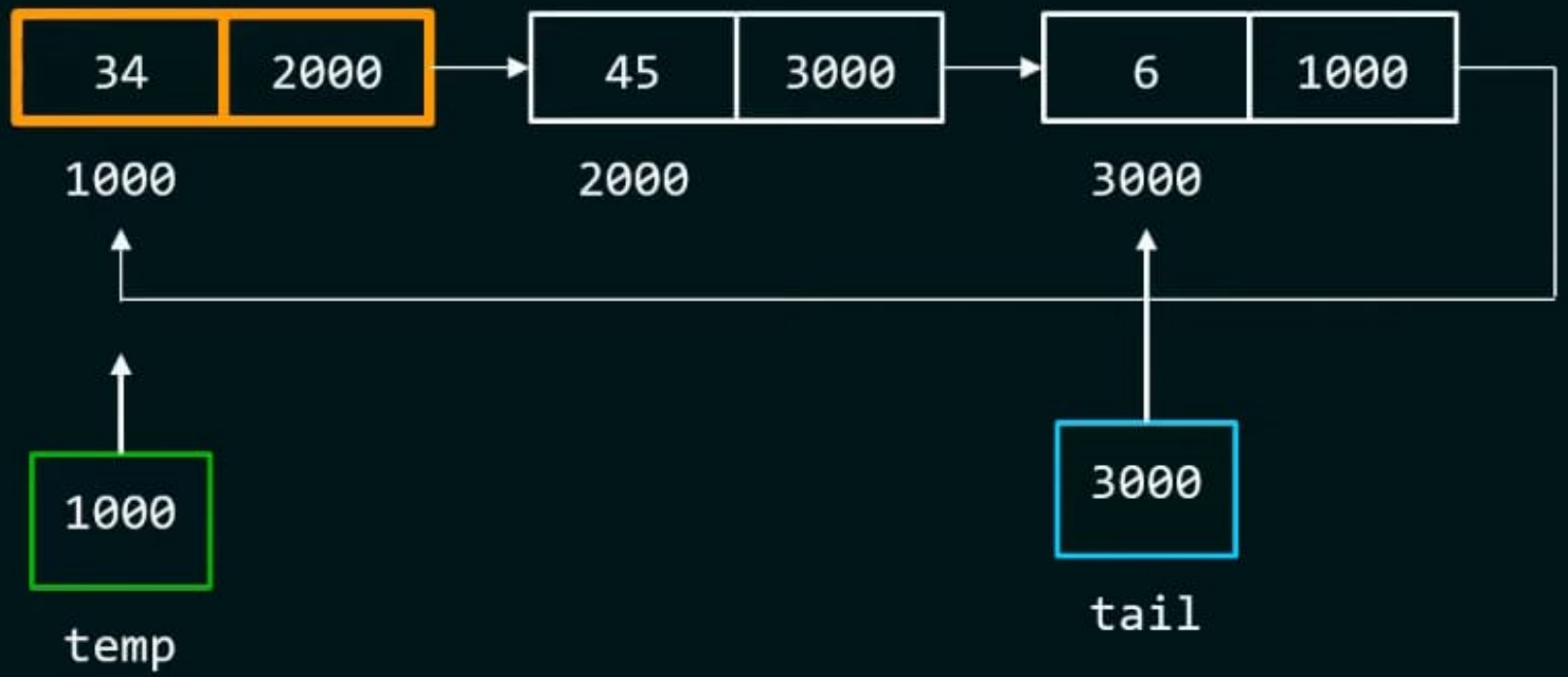


deleteFromHead Function

- In this operation, the information from the first node is temporarily stored in a local variable, the first node is deleted, and then head is reset so that what was the second node becomes the first node.
- The information of the deleted node is returned.
- The next member of the last node is reset to point to the first node of the list.


Circular Singly Linked List
34
2000
45
3000
6
1000
1000
2000
3000
Our target is to delete this node.
3000
tail

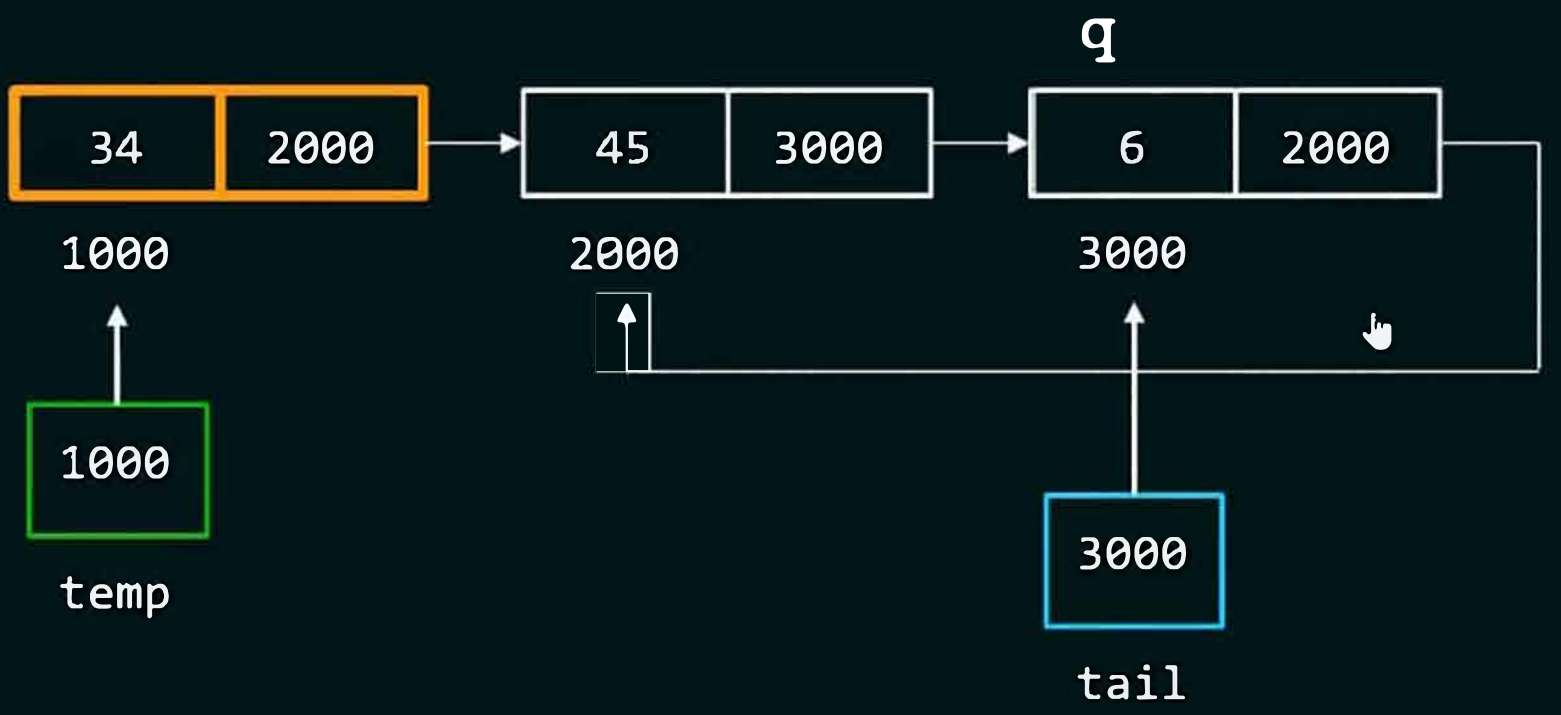
Circular Singly Linked List




```
struct node* temp;
temp = tail->next;
```

Now, we should update the next part of the last node so that it can point to the second node of the list.

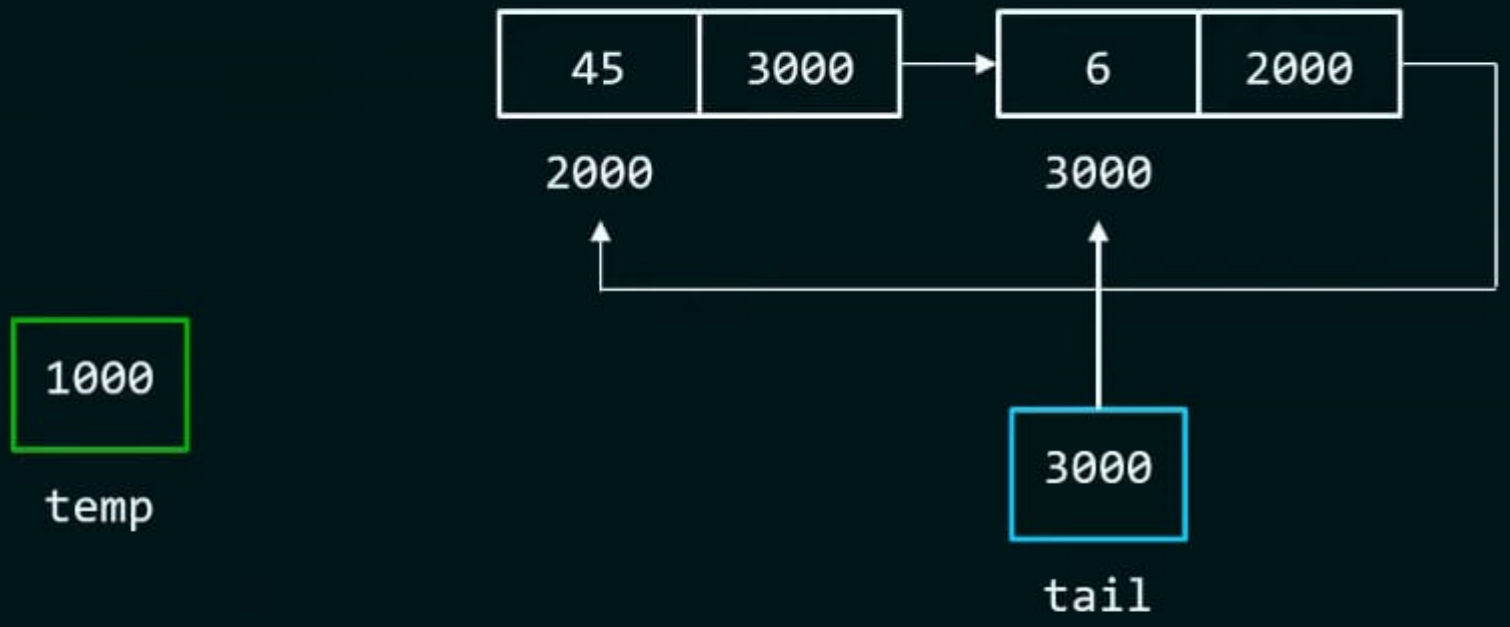
Circular Singly Linked List




```
struct node* temp;
temp = tail->next;
tail->next = temp->next;
```

Now, we should update the next part of the last node so that it can point to the second node of the list.

Circular Singly Linked List




```
struct node* temp;
temp = tail->next;
tail->next = temp->next;
free(temp);
temp = NULL;
```

deleteFromHead Function

```
int circularList::deleteFromHead(
)
{
        int x;
        if(head==NULL)
        {
                cout<<"list Empty";
                exit(0);
        }
```

Case 1

```
else if(head->next==head)
{
        x=head->info;
        delete head;
        return x;
}
```

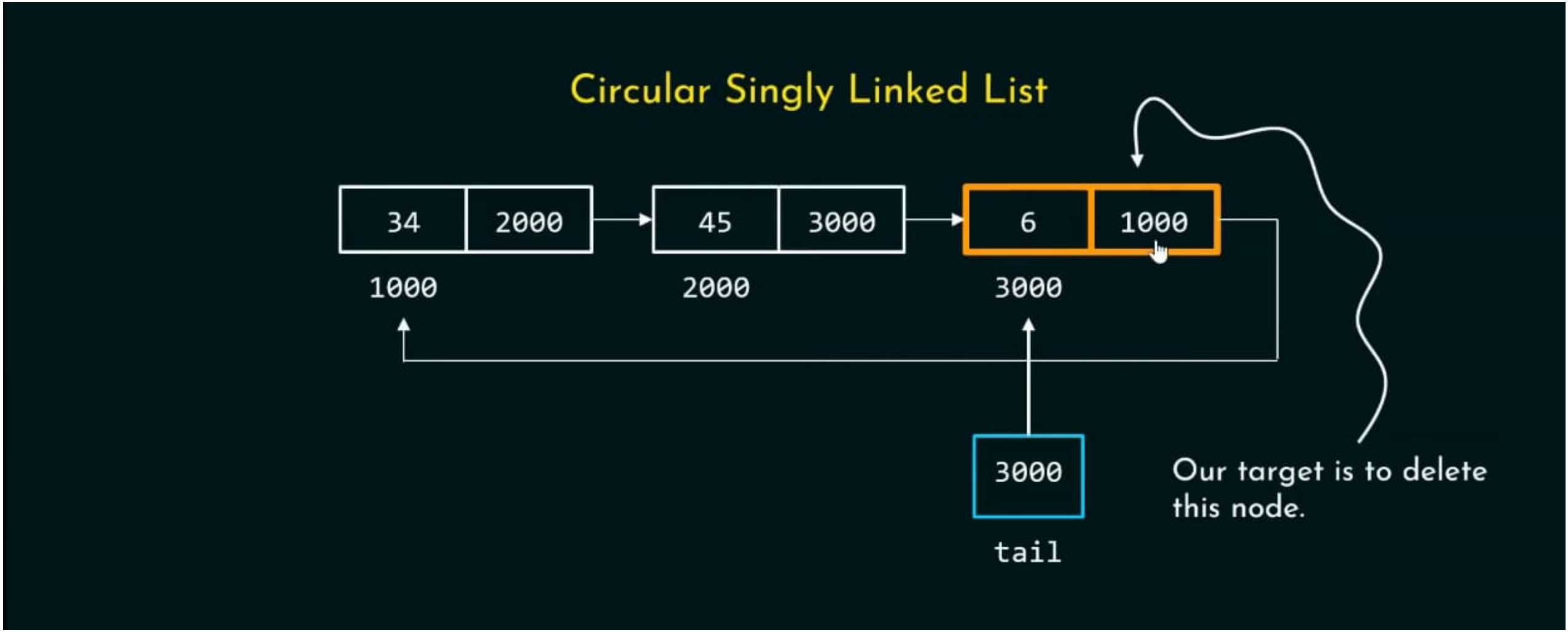
Case 2

```
else{
node* p=head,*q=head;
        while(q->next!=head)
                q=q->next;
        head=head->next;
        q->next=head;
        x=p->info;
        delete p; return x;
} }
```

Case 3

deleteFromTail Function

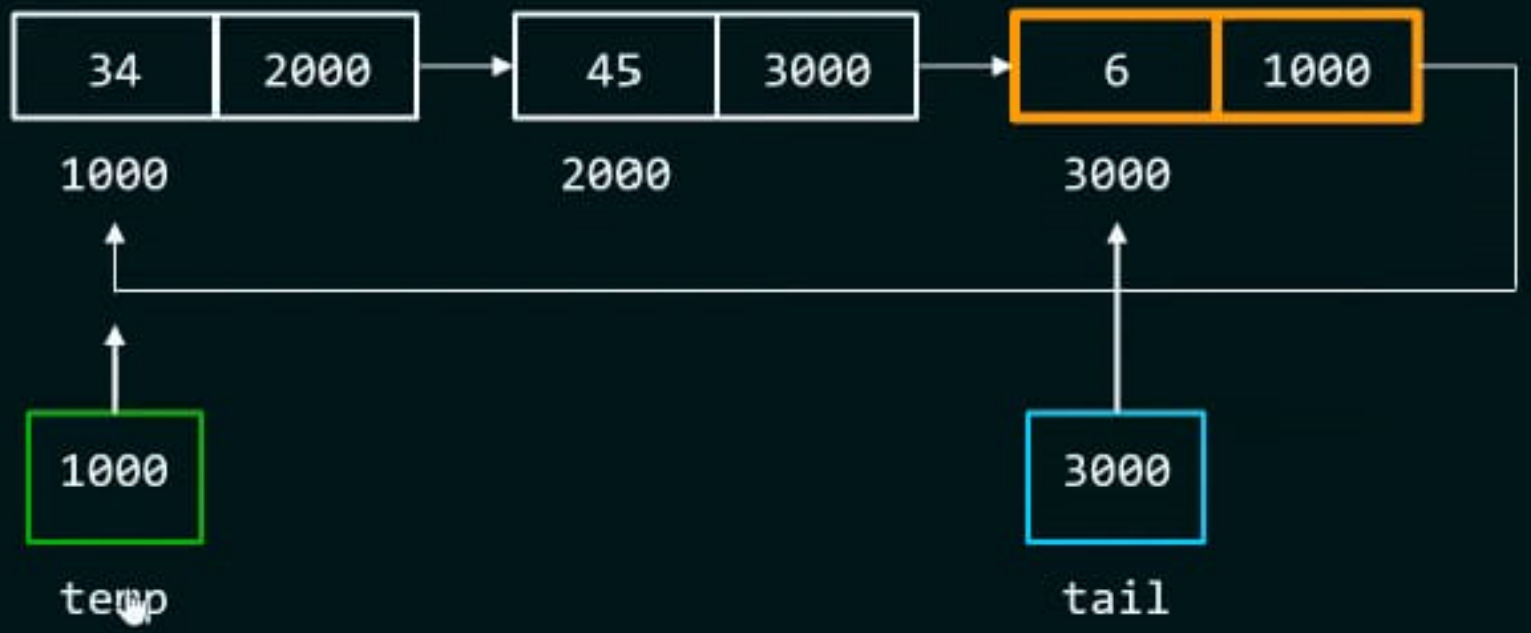
- In this operation, the information from the last node is temporarily stored in a local variable, the last node is deleted and then the next of the new last node is set to head.
- The information of the deleted node is returned.


Circular Singly Linked List
34
2000
45
3000
6
1000
1000
2000
3000
3000
tail
Our target is to delete this node.

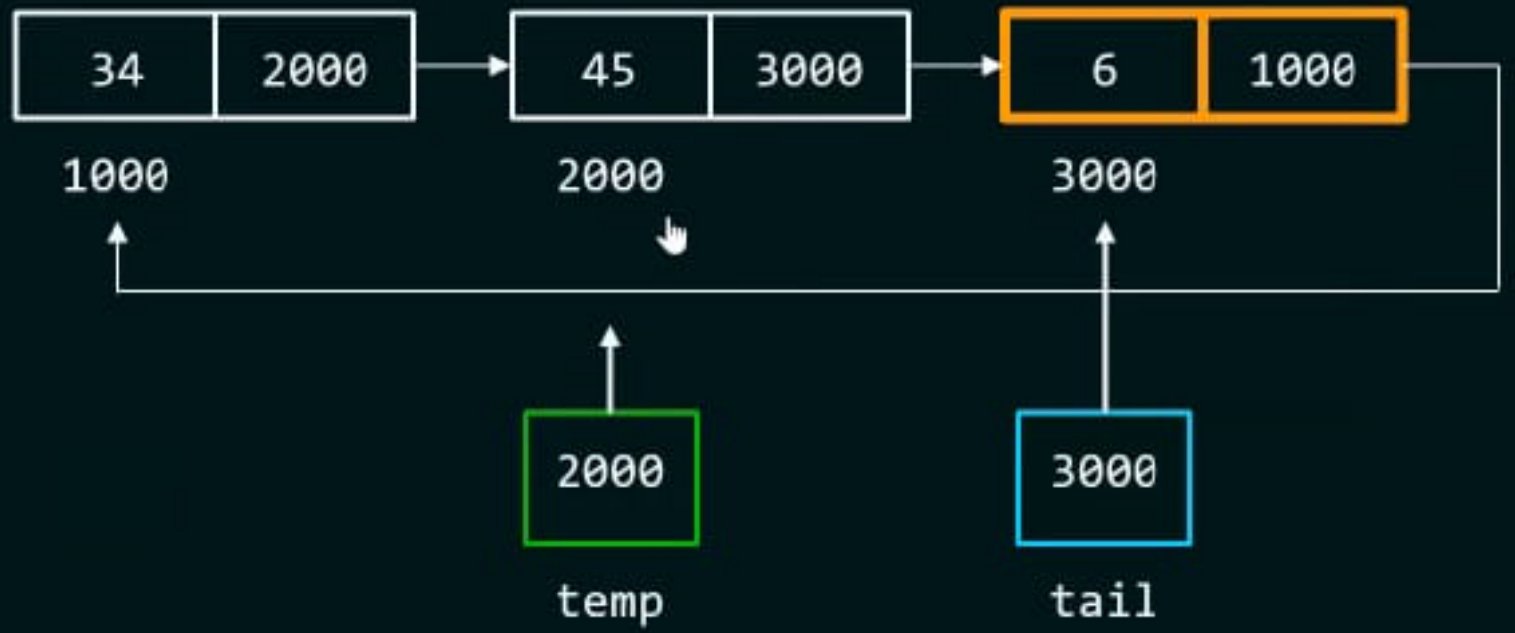
Circular Singly Linked List

temp

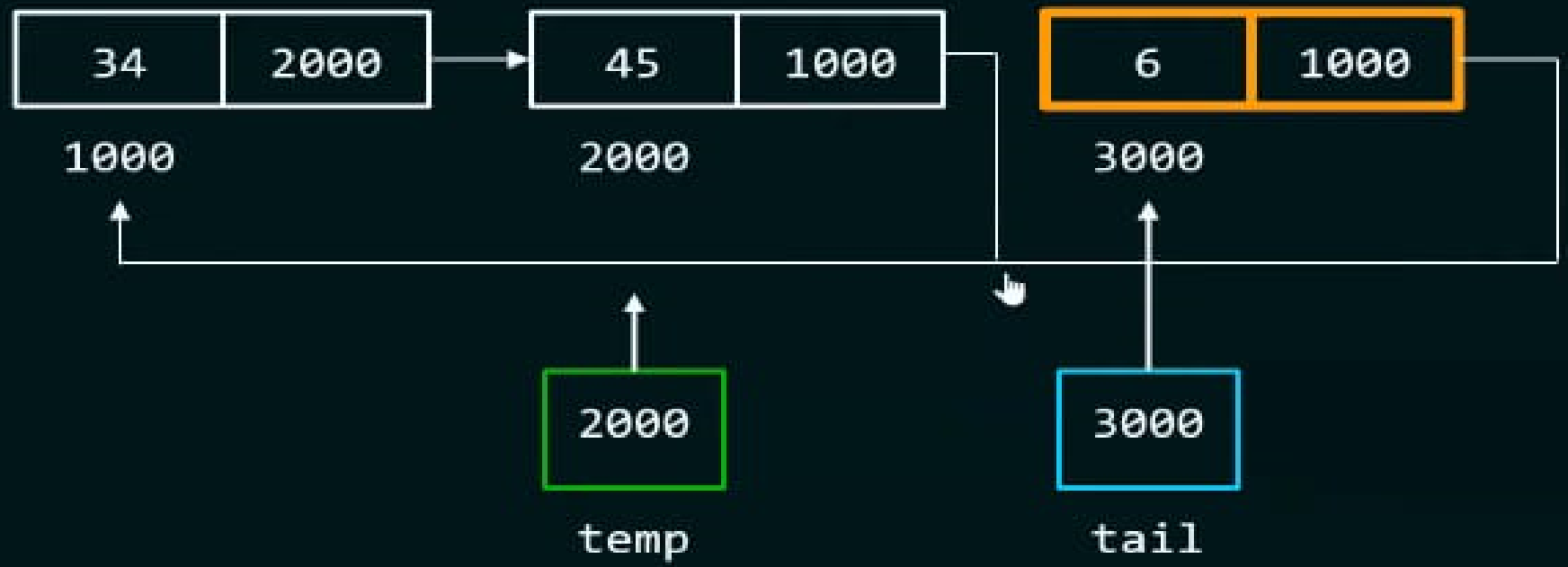
NULL



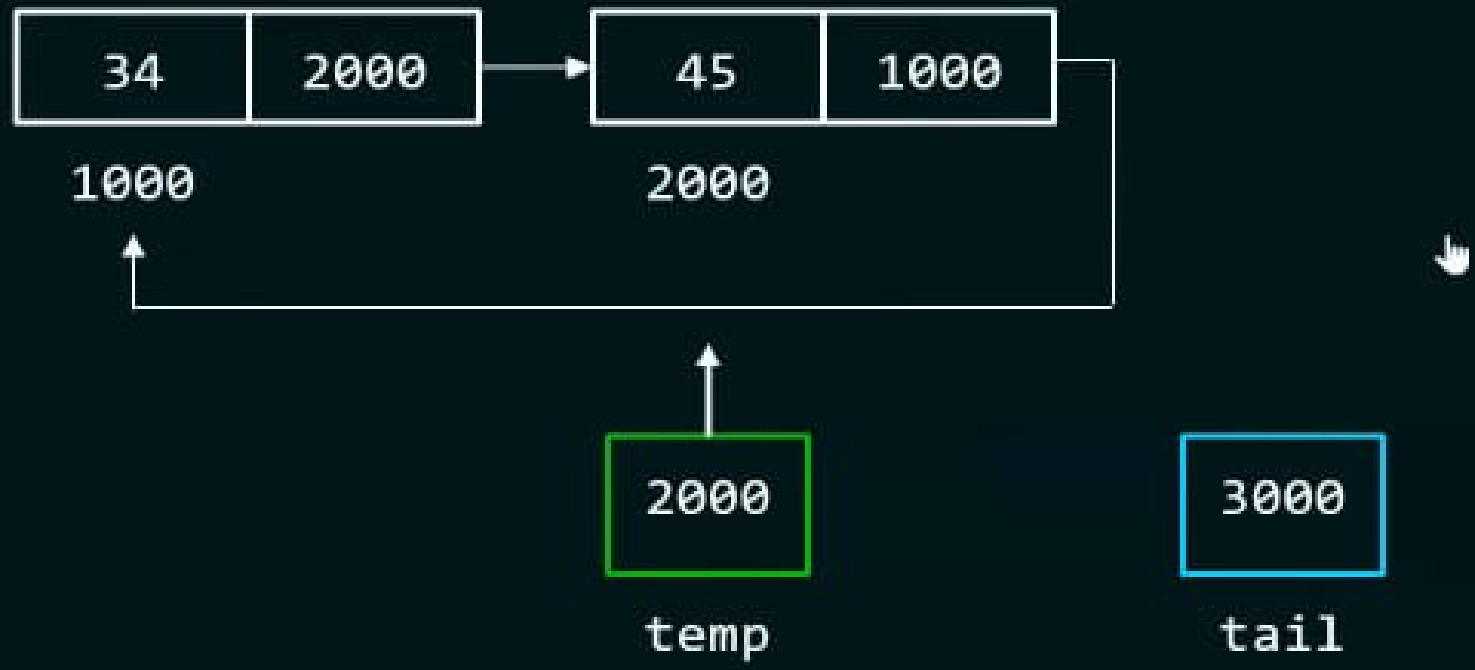
Circular Singly Linked List



Circular Singly Linked List



Circular Singly Linked List



deleteFromTail Function

```
int circularList::deleteFromTail()
{
        int x;
        if(head==NULL)
        {
                printf("list Empty");
                exit(0);
        }
       else if(head->next==head)
        {
                x=head->info;
                delete head;
                return x;
        }
```

Case 1

Case 2

```
else
{
        node* p=head,*q;
        while(p->next!=head)
        {
                q=p;
                p=p->next;
        }
        x=p->info;
        delete p;
        q->next=head;
        return x;
}
}
```

Case 3

Data Structure

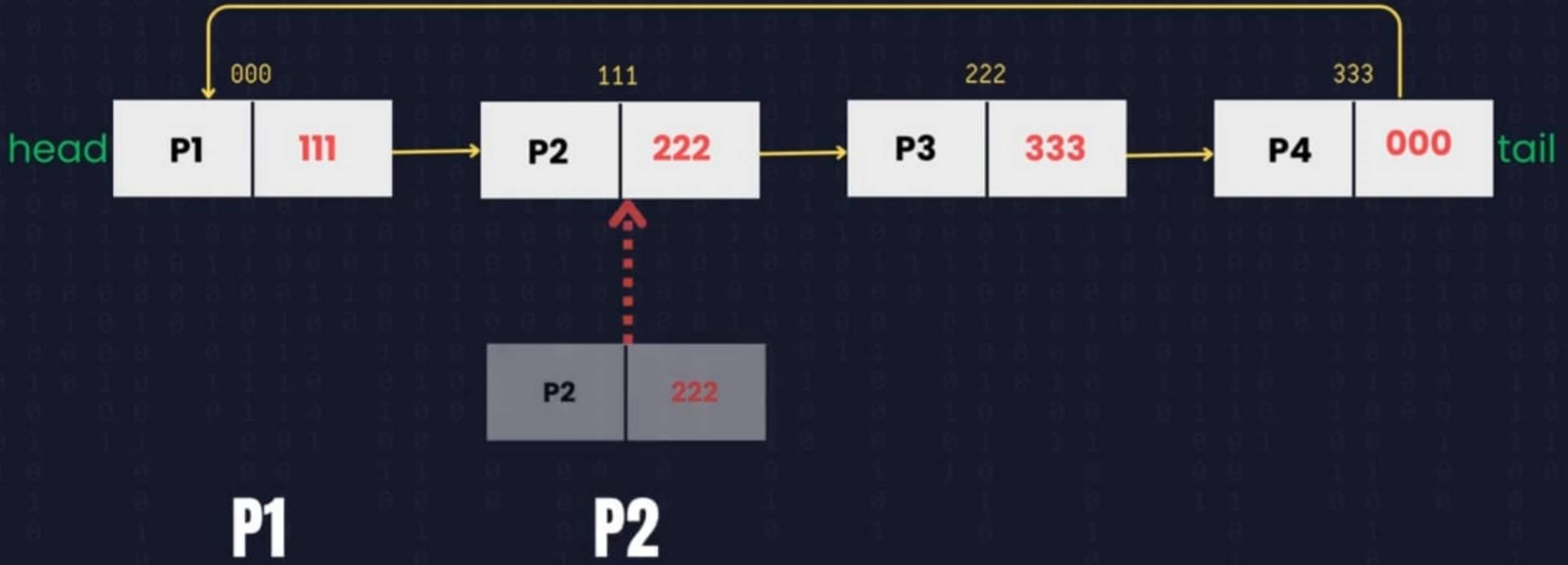
Circular Single LinkedList

printList

Data Structure

Circular Single LinkedList

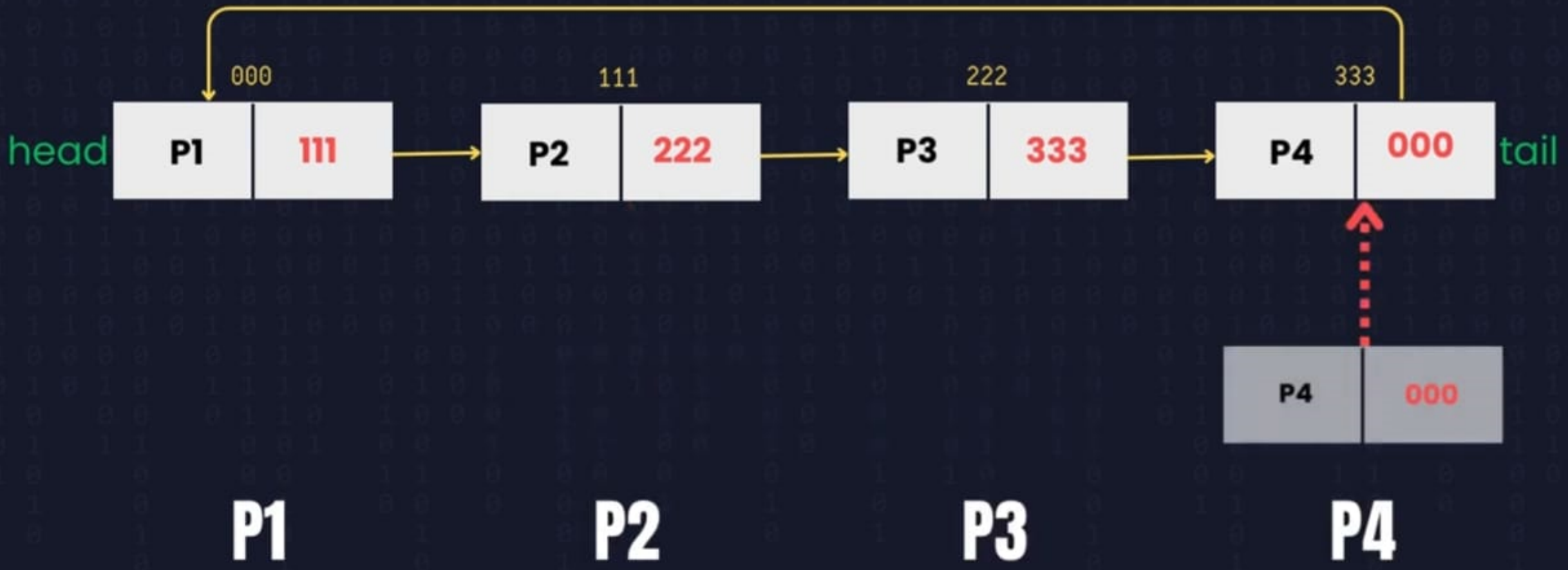
printList



Data Structure

Circular Single LinkedList

printList



printList Function

- This function is used to print the information of all nodes in the list.
- If the list is empty, it prints List Empty.

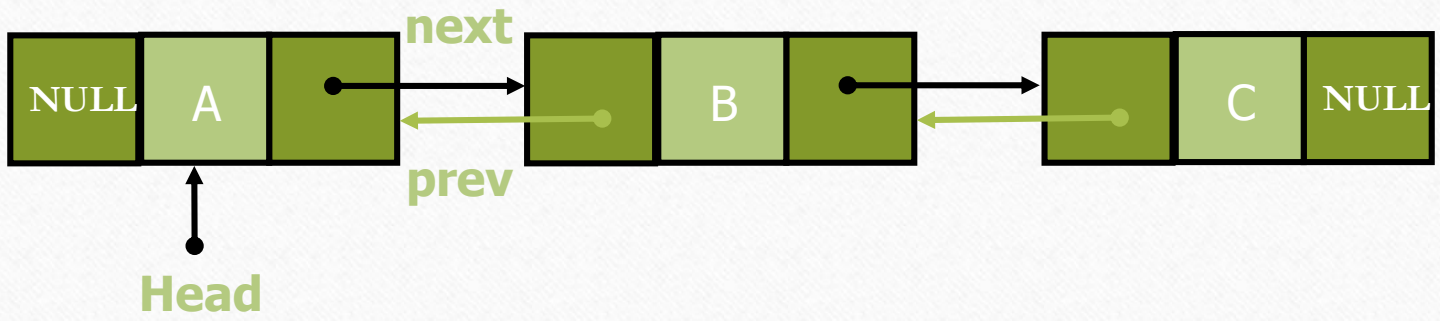
```
void circularList::printList()
{
      if(head==NULL)
            cout<<"List Empty";
      else
      {
            Node *p=head;
            while(p->next!=head){
                  printf("%d\t",p->info);
                  p=p->next;}
               printf("%d\t",p->info);
               printf("\n");
      }}
```

countList Function

- This function is used to print the number of nodes in the list.

```
int circularList::countList()
{
      int c=1;
      node* p=head;
      if(head==0)  return 0;
      else {
            while(p->next!=head)
            {
                  c++;
                  p=p->next;
            }
             return c;}
}
```

Doubly linked lists



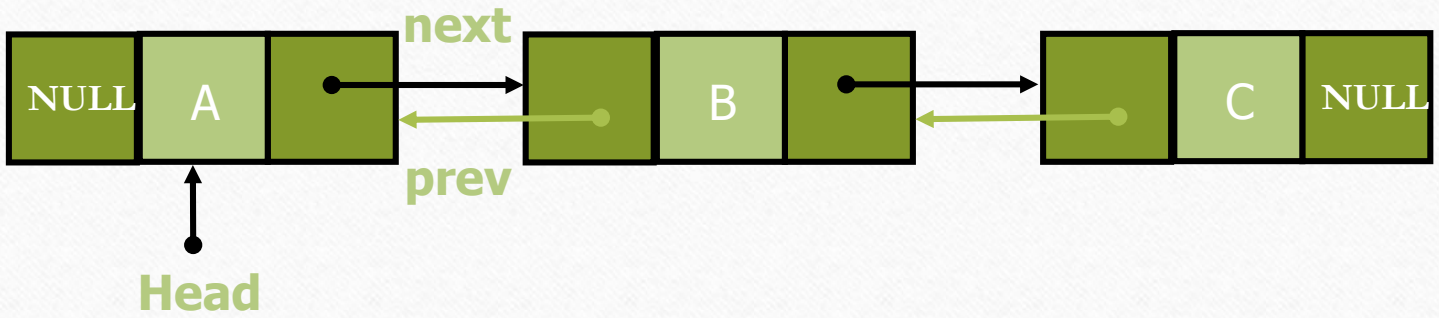
Singly linked lists

- we cannot efficiently delete an arbitrary node from an interior position of the list if only given a reference to that node.

Doubly linked lists

- Each node keeps an explicit **reference to the node before** it and **a reference to the node after it**
- we continue to use the term "**next**" for the reference to the node that follows another, and we introduce the term "**prev**" for the reference to the node that precedes it

Doubly linked lists



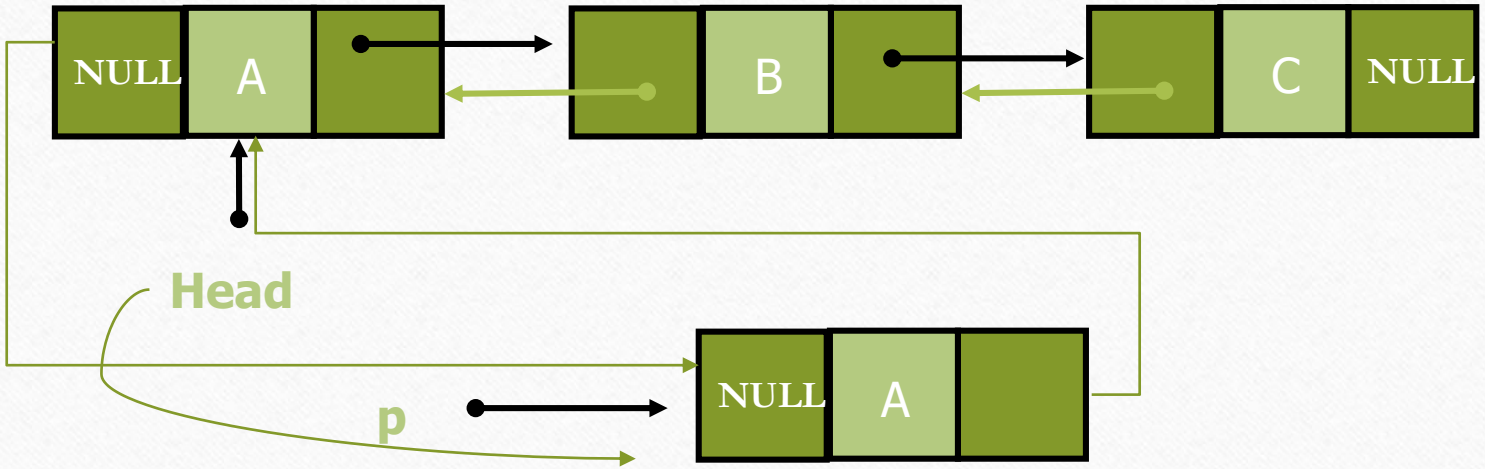
- Each node points to not only the successor but also the predecessor.
- There are two NULL pointers: one at the first node and the other at last node in the list
- **Advantage:**
  - given a node, it is easy to visit its predecessor. Convenient to traverse lists backwards

AddToHead Function

- To insert a node at the front of the list (this list is used to store integer data):

1. An empty node is created.
2. The node's info member is initialized to a particular integer.
3. The prev member of the new node is set to NULL.
4. Because the node is being included at the front of the list, the next member becomes a pointer to the first node on the list; that is, the current value of head.
5. The prev member of the node pointed to by head becomes a pointer to the new node.
6. The new node precedes all the nodes on the list, but this fact has to be reflected in the value of head; otherwise, the new node is not accessible. Therefore, head is updated to become the pointer to the new node .

AddToHead Function



AddToHead Function

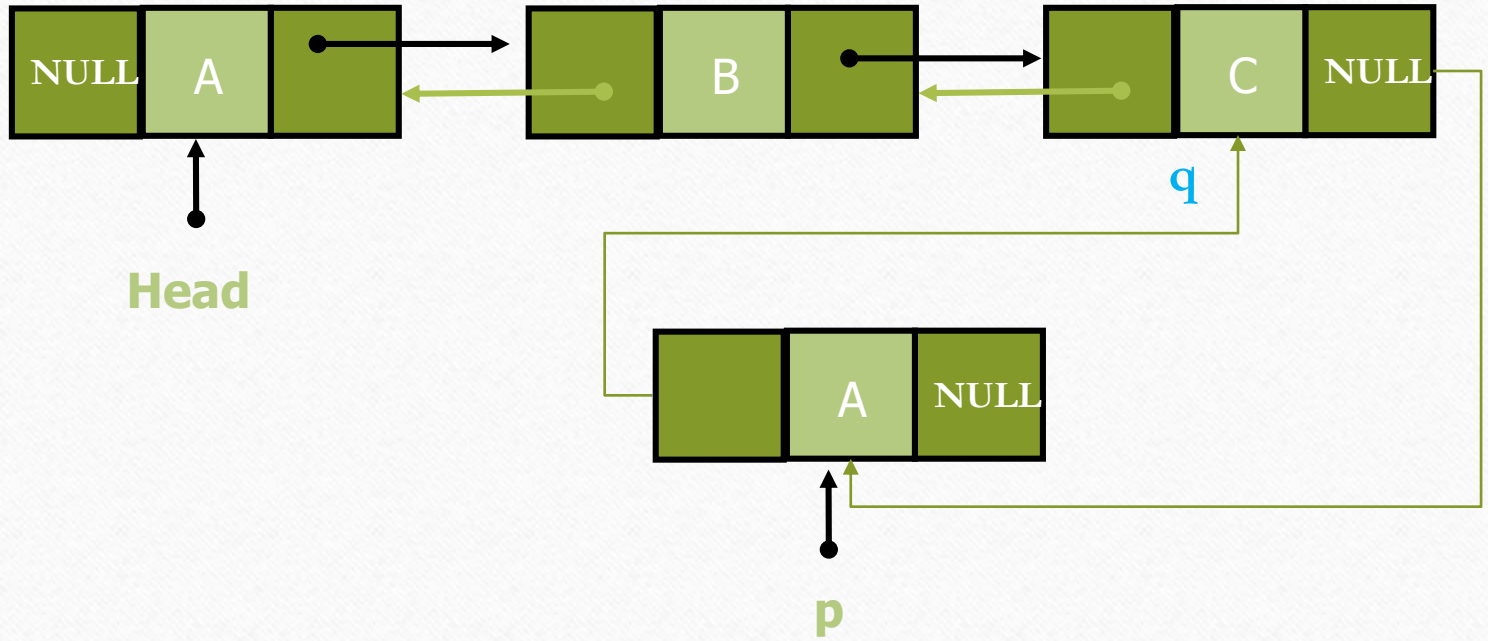
```
void doublyList::addToHead(int data)
{     node* p= new node;
      p->info=data;
      p->prev=NULL;
      if(head!=NULL)
            head->prev=p;
      p->next=head;
      head=p;
}
```

AddToTail Function

- To insert a node at the end of the list:

1. An empty node is created.
2. The node's info member is initialized to an integer.
3. Because the node is being included at the end of the list, the next member is set to **NULL.**
4. The node is now included in the list by making the prev member of the new node a pointer to the last node of the list and the next member of the last node of the list a pointer to the newly created node.

AddToTail Function



AddToTail Function

```
void doublyList::addToTail(int data) {
        node* p= new node;
        p->info=data;
        p->next=NULL;
        if(head!=NULL) {
                node* q=head;
                while(q->next!=NULL)
                        q=q->next;
                q->next=p;
                p->prev=q;   }
```

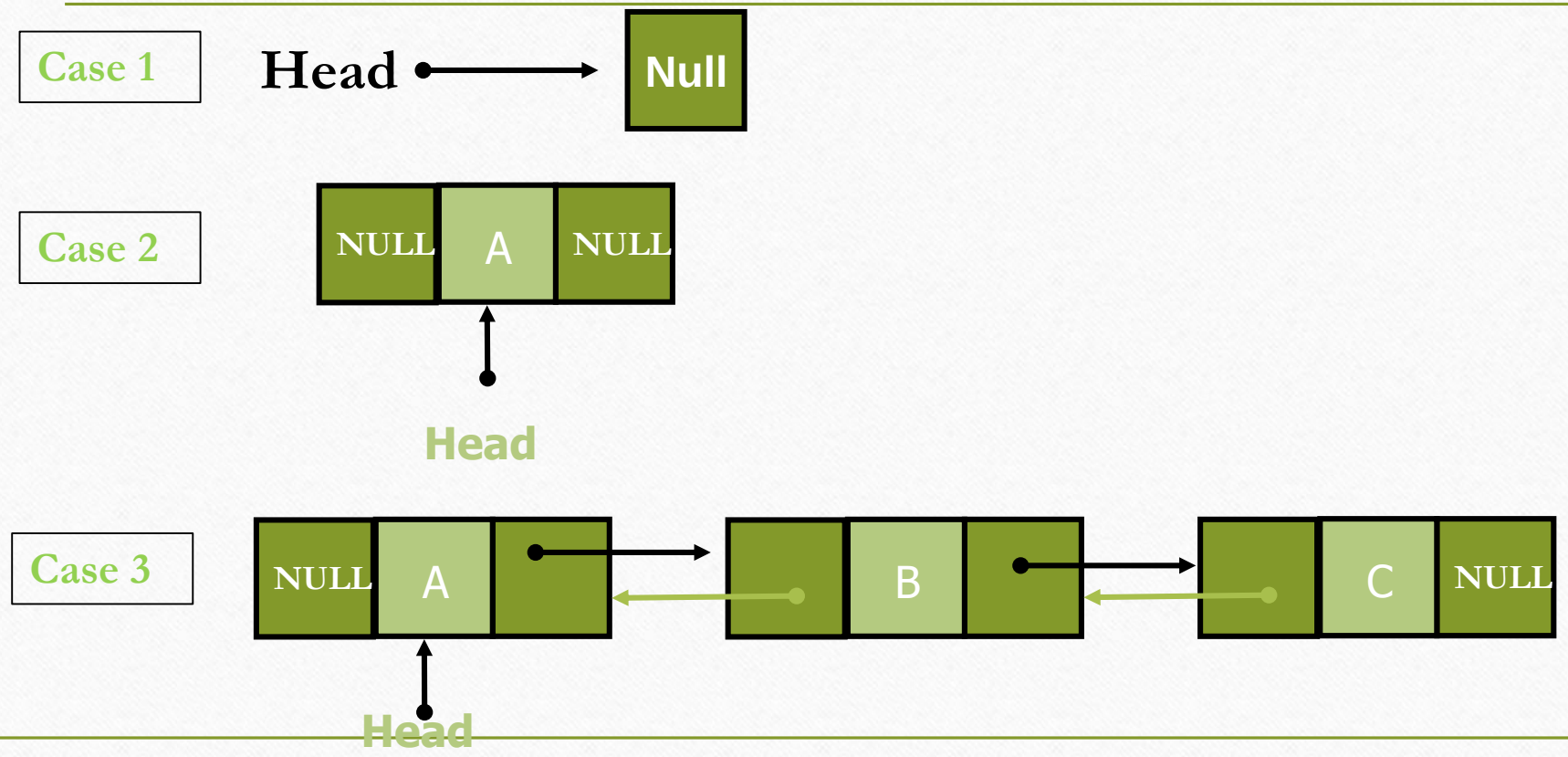
```
else {
                p->prev=NULL;
                head=p;
        }
}
```

Deleting a node from a Doubly List

When deleting a node from a list, we need to consider all the following cases:

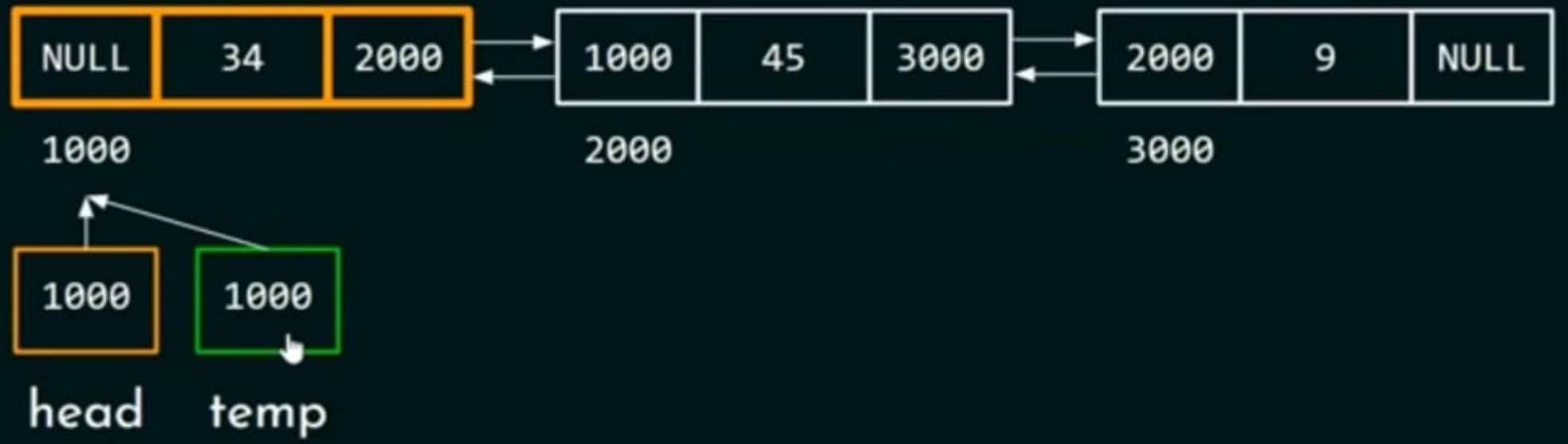
- Case 1: deleteing a node from an empty list → the function is immediately exited.
- Case 2: deleting the only node from a one-node linked list → the head is set to null.
- Case 3: deleting the first node of the list with at least two nodes → requires updating head.
- Case 4: deleting a node with a number that is not in the list → do nothing.

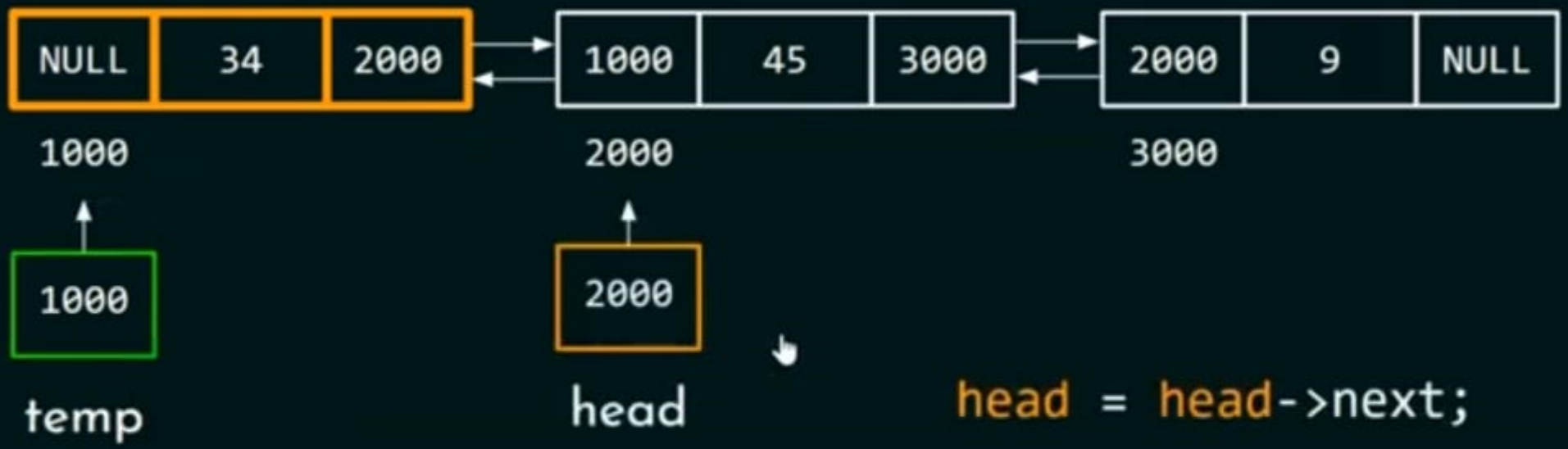
Deleting a node from a Circular List

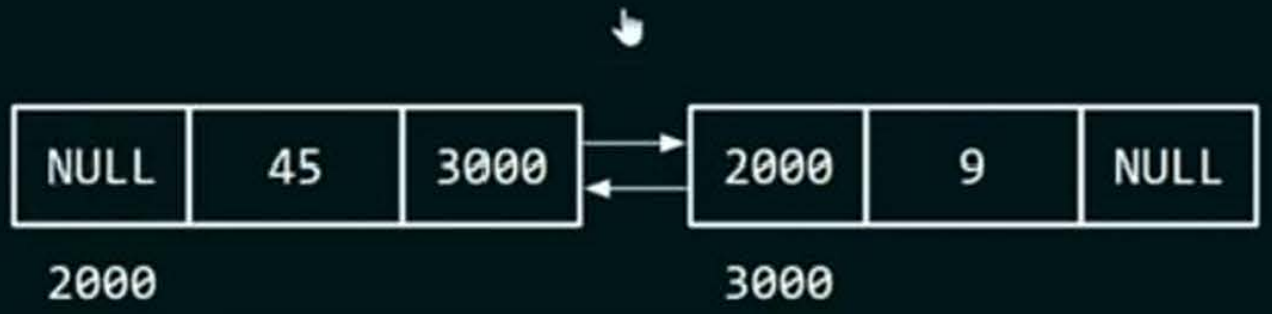


deleteFromHead Function

- In this operation, the information from the first node is temporarily stored in a local variable, the first node is deleted, and then head is reset so that what was the second node becomes the first node.
- The prev member of the first node pointed to by head is reset to be NULL;
- The information of the deleted node is returned.







temp

head

```
head = head->next;
free(temp);
temp = NULL;
head->prev = NULL;
```

deleteFromHead Function

```
int doublyList::deleteFromHead(
)
{
        int x;
        if(head==NULL)
        {
                cout<<"list Empty";
                exit(0);
        }
```

Case 1

```
else if(head->next==NULL)
{
        x=head->info;
        delete head;
        head=NULL; // list
        become empty//
}       return x;
```

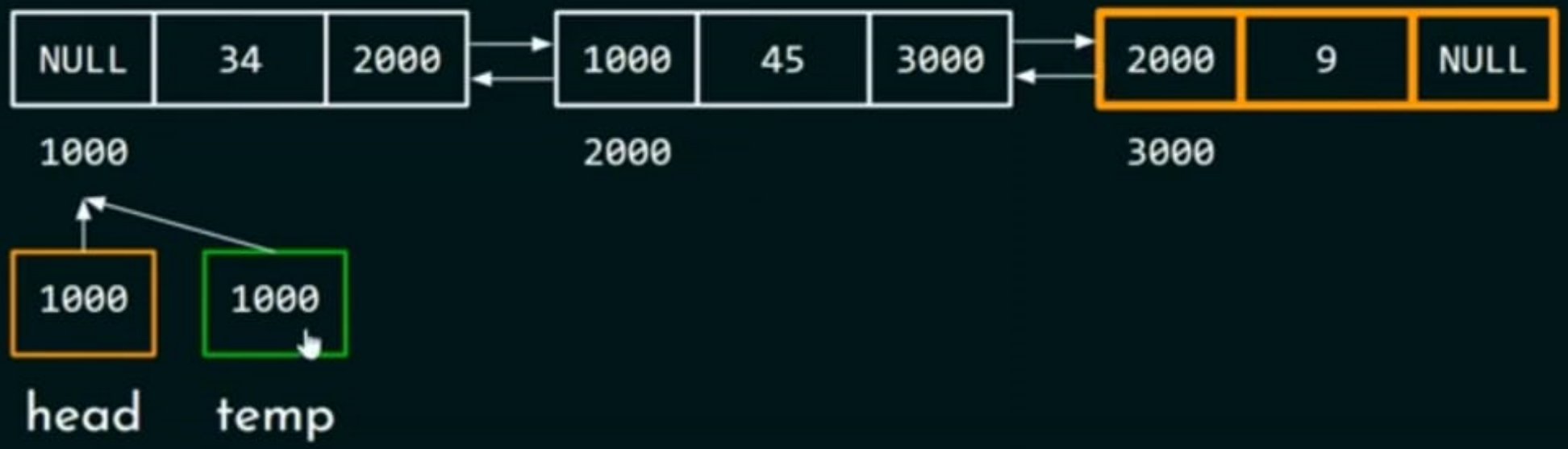
Case 2

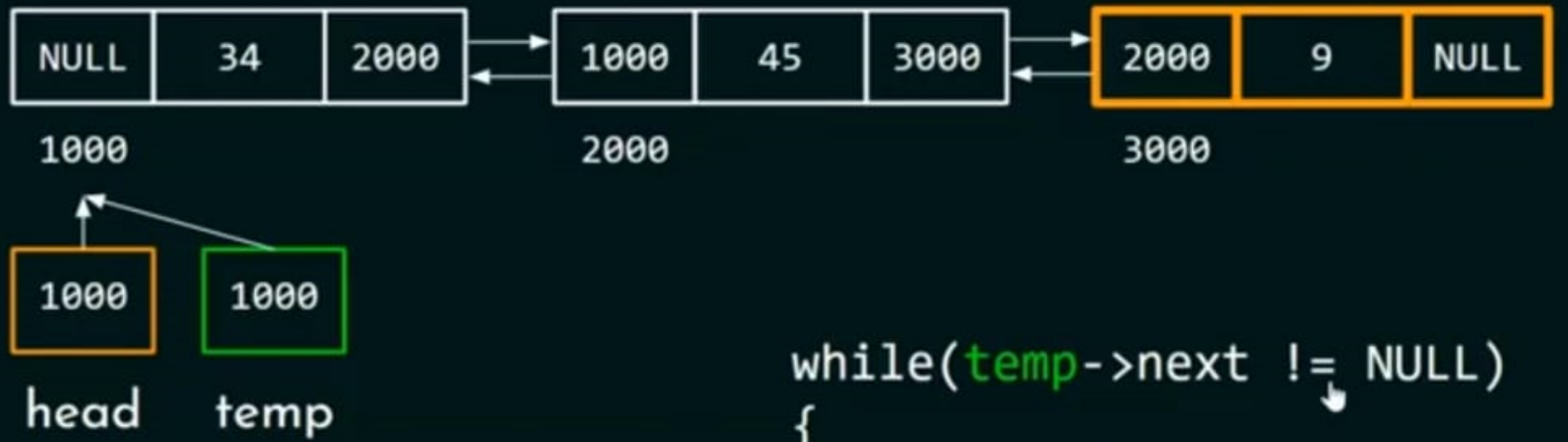
```
else{
node* p=head;
p->next->prev=NULL;
head=head->next;
x=p->info;
delete p;
return x;} }
```

Case 3

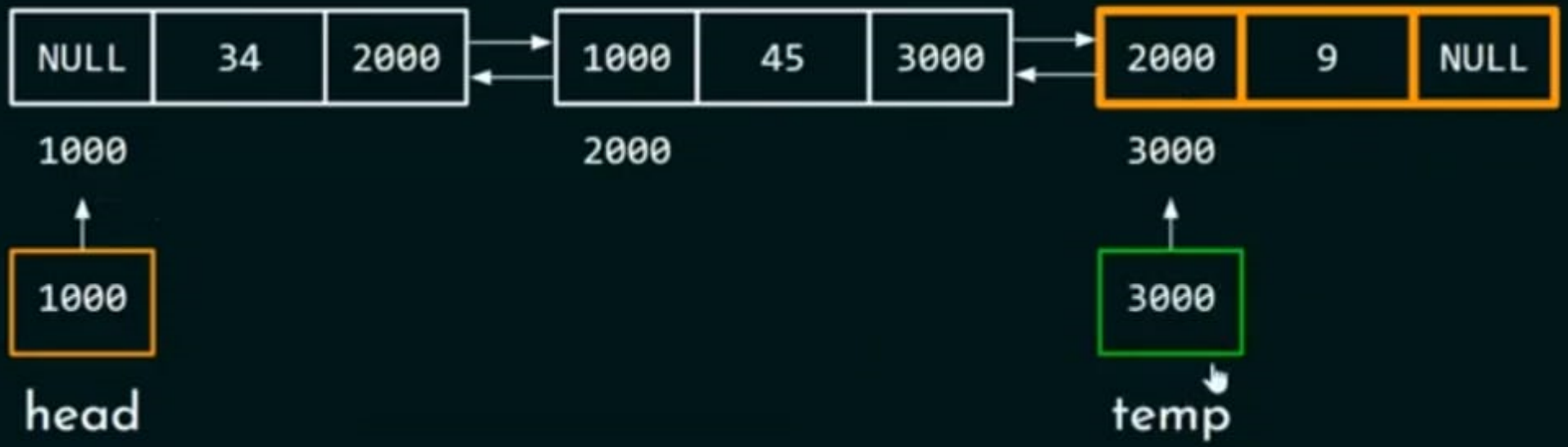
deleteFromTail Function

- In this operation, the information from the last node is temporarily stored in a local variable, the last node is deleted and then the next of the new last node is set to NULL.
- The information of the deleted node is returned.

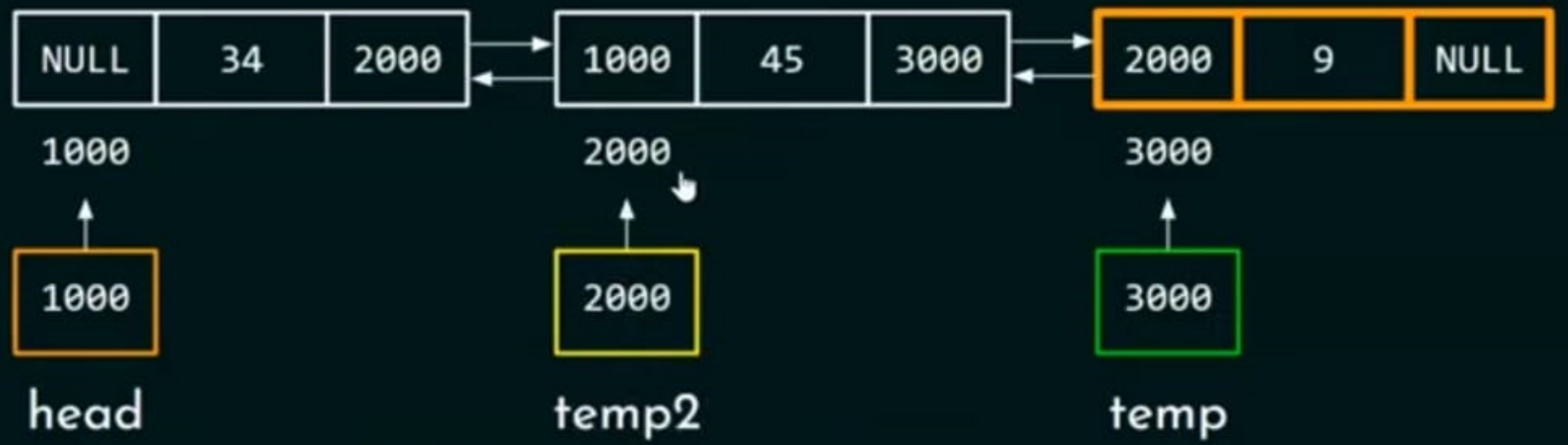





```
while(temp->next != NULL)
{
    temp = temp->next;
}
```




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




```
while(temp->next != NULL)
{
    temp = temp->next;
}
temp2 = temp->prev;
```

deleteFromTail Function

```
int doublyList::deleteFromTail()
{
        int x;
        if(head==NULL)
        {
                printf("list Empty");
                exit(0);
        }
       else if(head->next==NULL)
        {
                x=head->info;
                delete head;  head=NULL;
                return x;
        }
```

Case 1

Case 2

```
else
{
        node* p=head;
                while(p->next!=NULL)
                        p=p->next;
                x=p->info;
                p->prev->next=NULL;
                delete p;
                return x;
}
}
```

Case 3

printList Function

- This function is used to print the information of all nodes in the list.
- If the list is empty, it prints List Empty.

```
void doublyList::printList()
{
      if(head==NULL)
            cout<<"List Empty";
      else
      {
            Node *p=head;
            while(p->next!=Null){
                  printf("%d\t",p->info);
                  p=p->next;}
               printf("%d\t",p->info);
               printf("\n");
      }}
```

countList Function

- This function is used to print the number of nodes in the list.

```
int doublyList::countList()
{
      int c=1;
      node* p=head;
      if(head==0)  return 0;
      else {
            while(p!=Null){

                  c++;
                  p=p->next;
            }
            return c;}
}
```

Array versus Linked Lists

- Linked lists are more complex to code and manage than arrays, but they have some distinct advantages.
  - **Dynamic**: a linked list can easily grow and shrink in size.
    - We don't need to know how many nodes will be in the list. They are created in memory as needed.
    - In contrast, the size of a C++ array is fixed at compilation time.
  - **Easy and fast insertions and deletions**
    - To insert or delete an element in an array, we need to copy to temporary variables to make room for new elements or close the gap caused by deleted elements.
    - With a linked list, no need to move other nodes. Only need to reset some pointers.

End of Lecture