LINKED LIST

AGENDA

1. Introduction
2. Singly Linked List
3. Circular Linked List
4. Doubly Linked List
5. Questions

WHAT IS LINKED LIST

- A linked list is a **linear data structure**, in which the elements are not stored at contiguous memory locations.
- The elements in a linked list are linked using **pointers**

WHAT IS ‘NODE’ ?

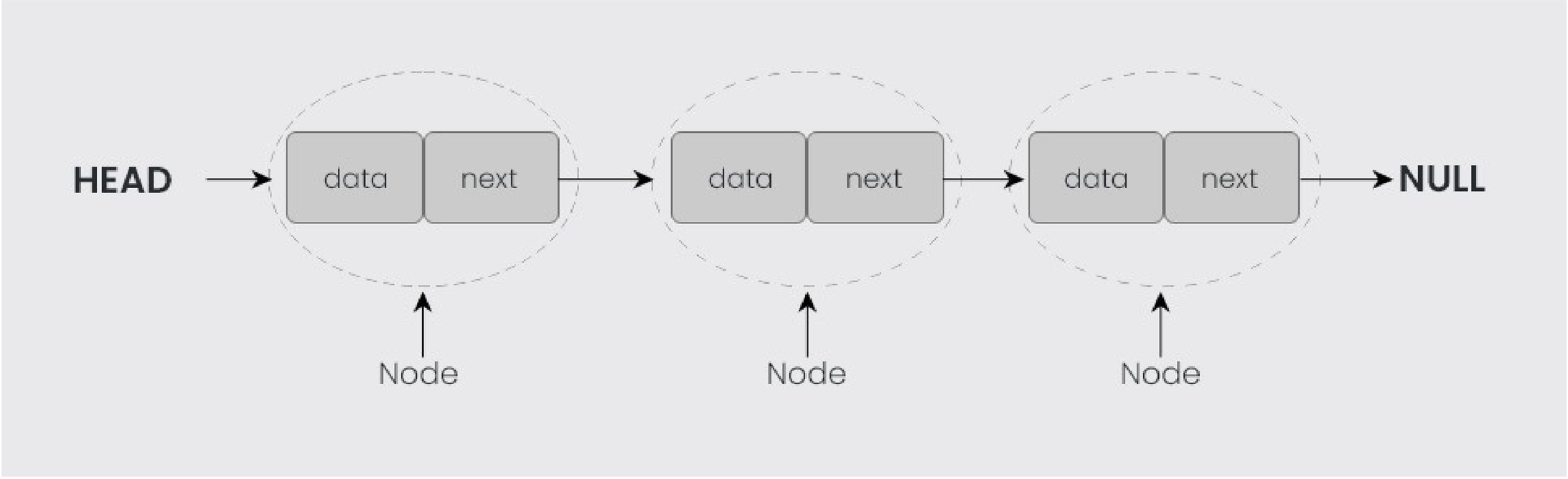
- Node contains a **data** fields and **reference** (link) to the next node in list.
- Node consists of **two** things

1. Data
2. Pointer

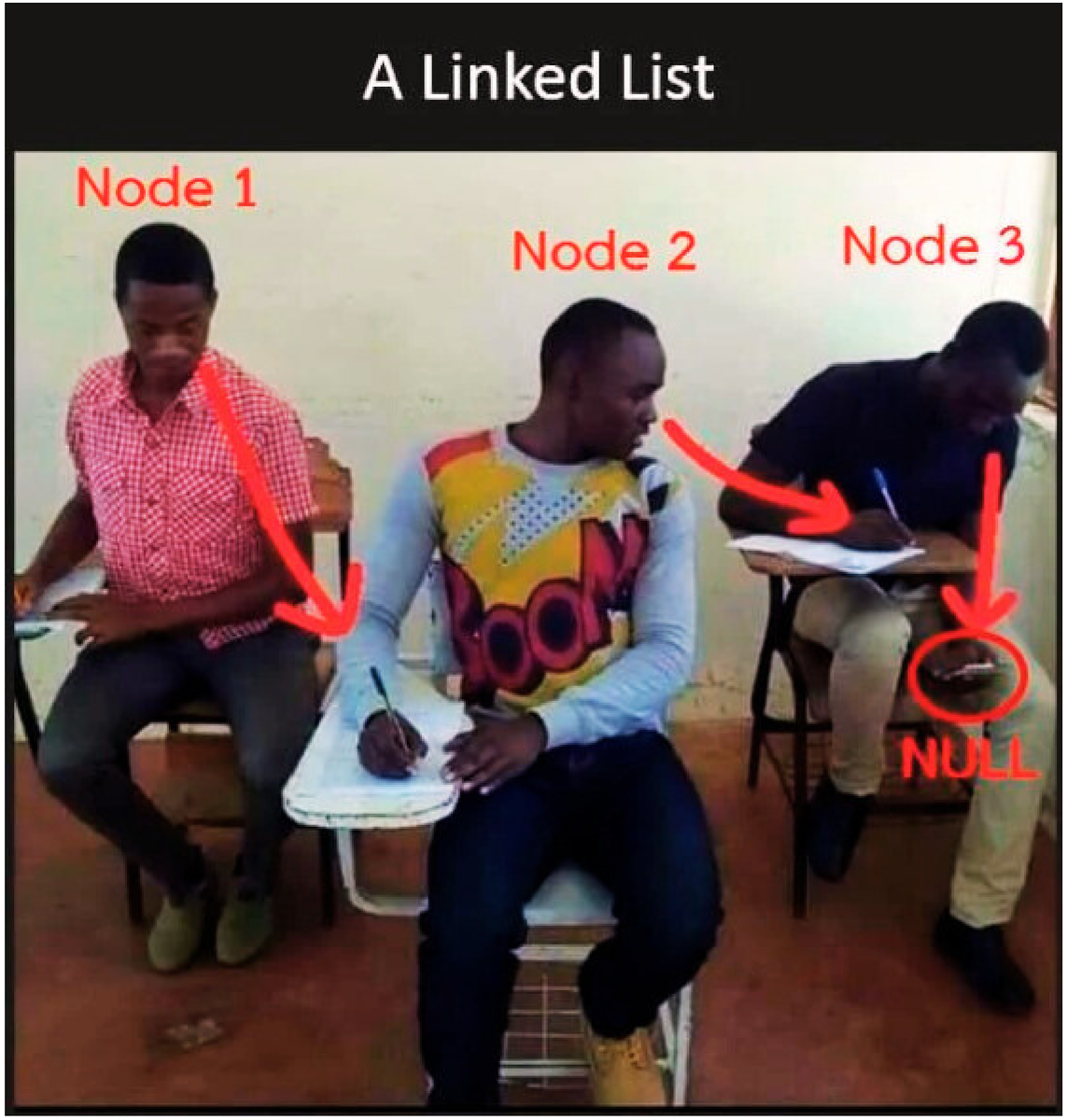
• Array Vs Linked List

ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

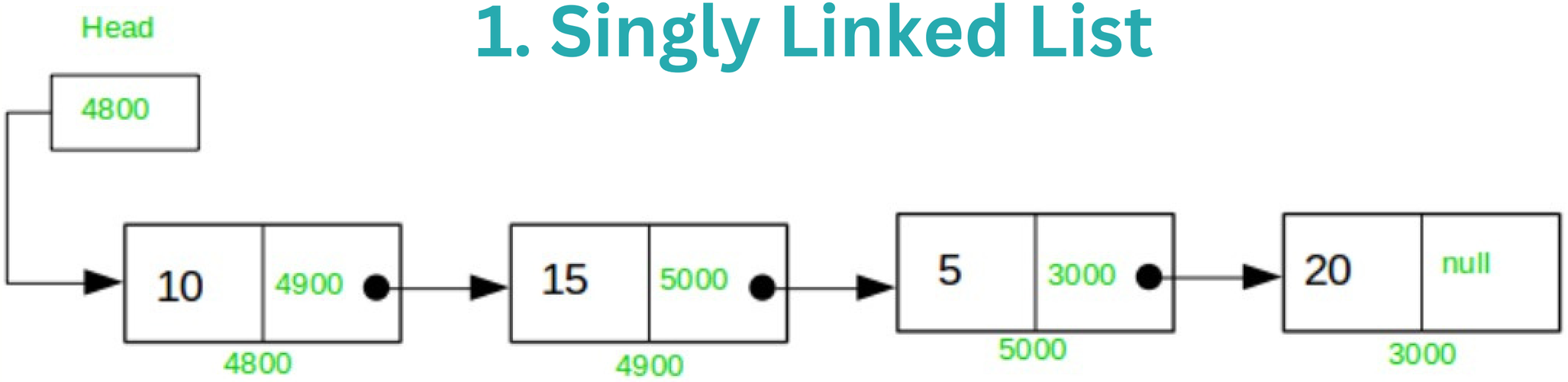
Structure of Linked List



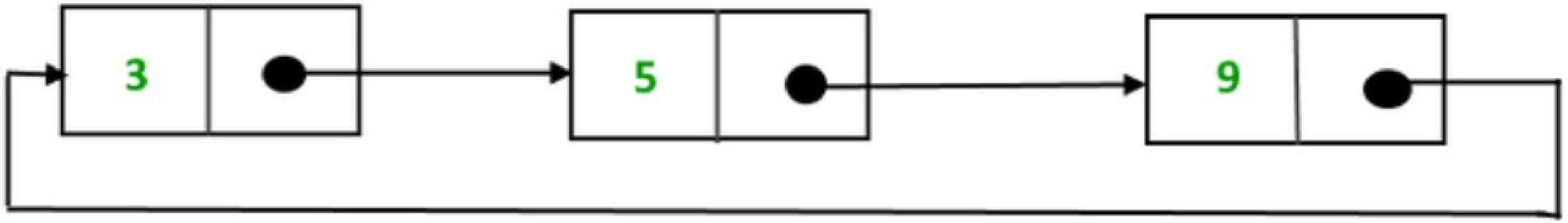
What do you See from this picture ?



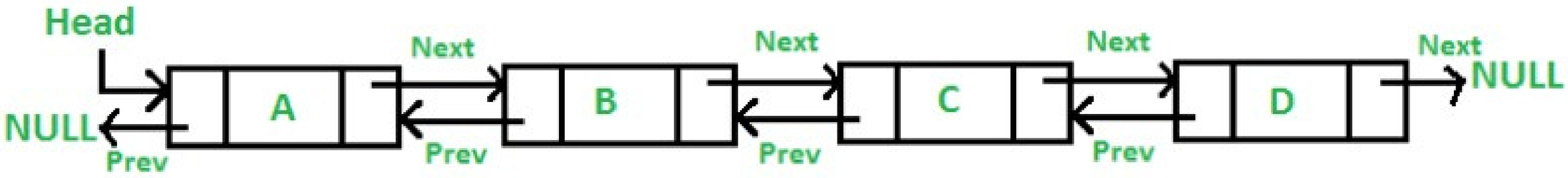
1. Singly Linked List



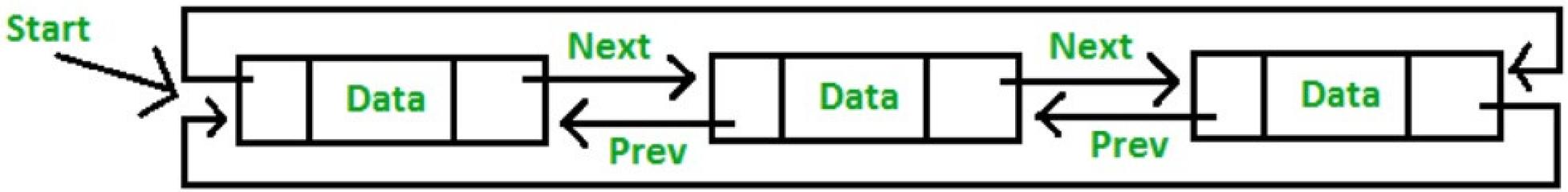
2.Circular Singly Linked List



3. Doubly Linked List



4. Circular Doubly Linked List



Implementation of SLL

Basics

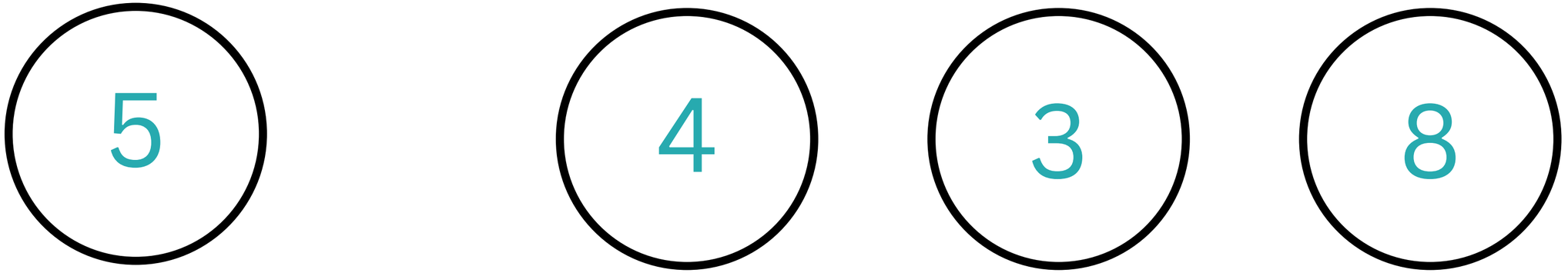
```
class Node{
    public :
    int val;
    Node* next;
    Node(int val){
      this->val=val;
      this->next=NULL;
    }
}
```

Insertion at Head:

Case 1 : Size = 0

Insertion at Head:

Case 2 : Size > 0



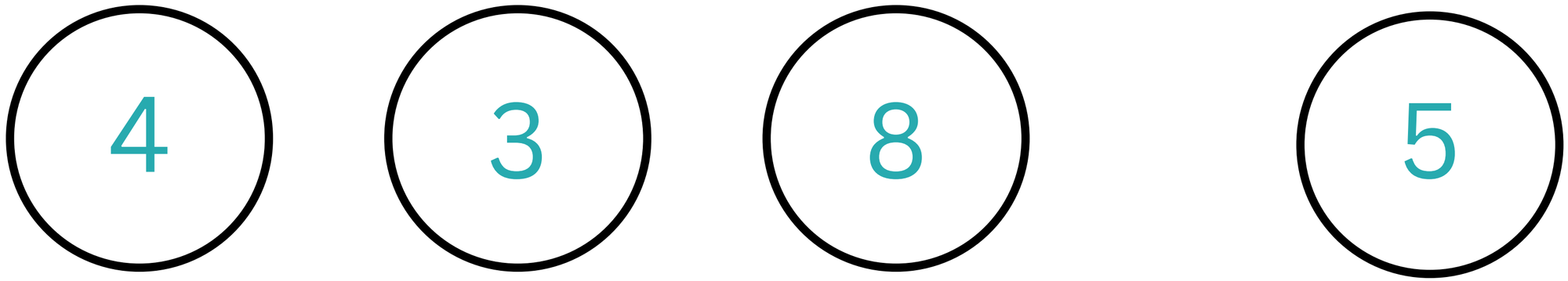
T.C. : O(1)
S.C. : O(1)

Insertion at Tail:

Case 1 : Size = 0

Insertion at Tail:

Case 2 : Size > 0



T.C. : O(1)

S.C. : O(1)

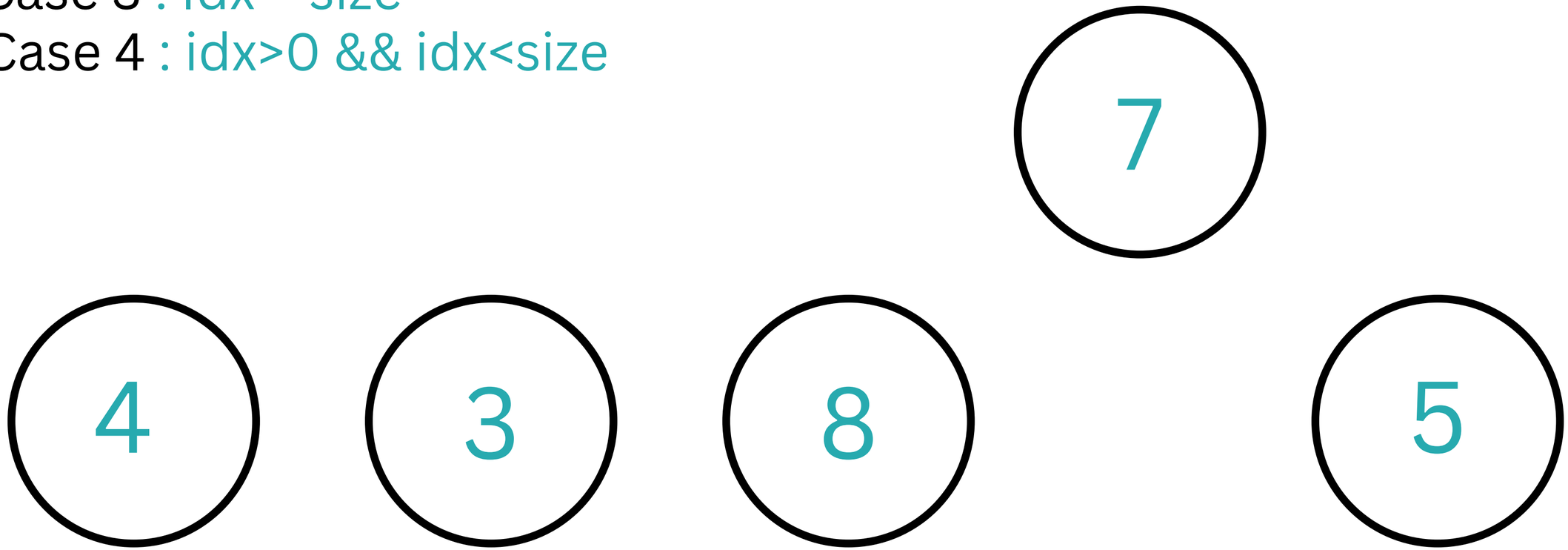
Insertion at Any Index:

Case 1 : (idx<0 || idx>size)

Case 2 : Idx = 0

Case 3 : Idx = size

Case 4 : idx>0 && idx<size



T.C. : O(n)

S.C. : O(1)

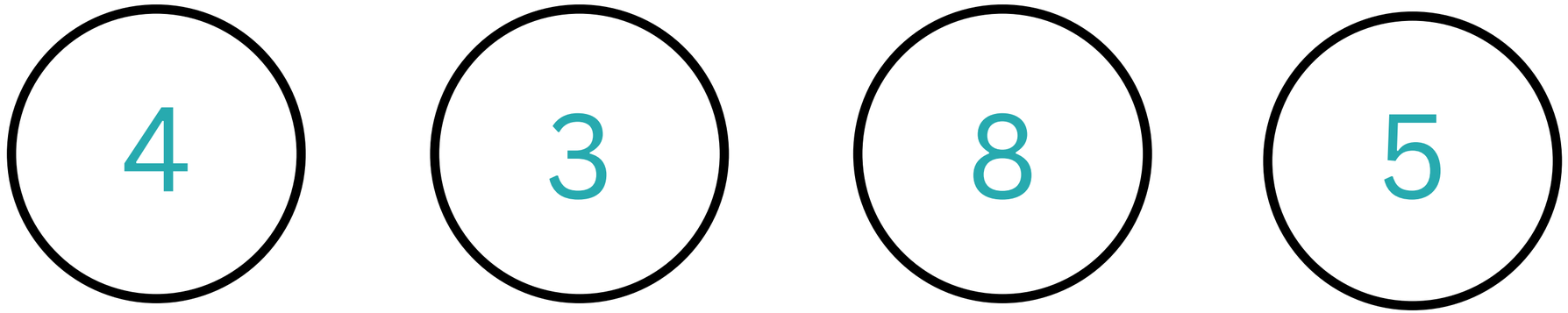
Deletion at Head:

Case 1 : size = 0

Case 2 : size = 1

Deletion at Head:

Case 3 : size > 0



T.C. : O(1)

S.C. : O(1)

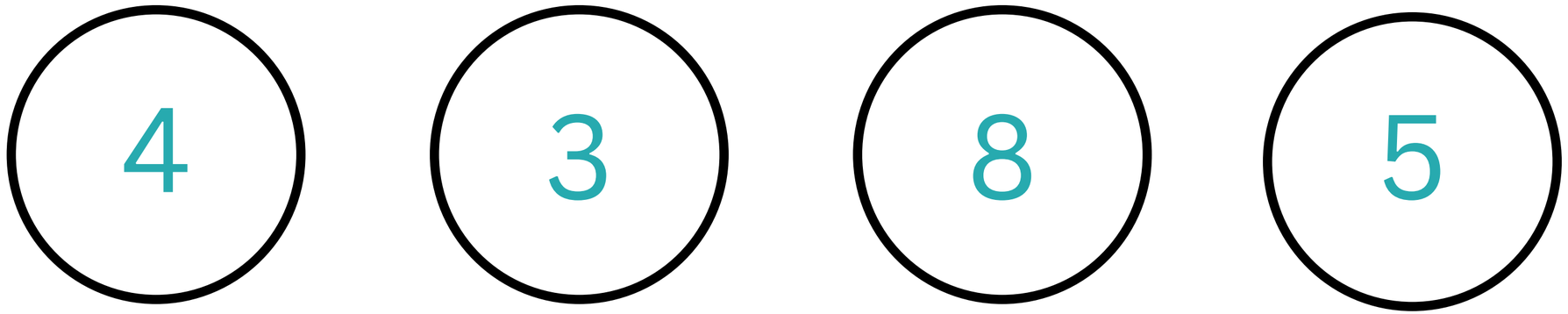
Deletion at Tail :

Case 1 : size = 0

Case 2 : size = 1

Deletion at Tail :

Case 3 : size > 0



T.C. : O(n)

S.C. : O(1)

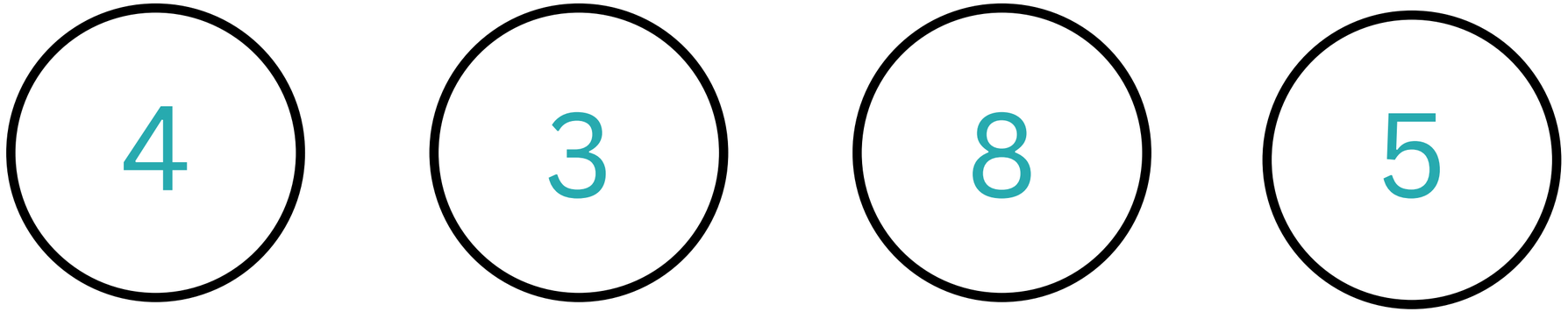
Deletion at Any Index :

Case 1 : Invalid Idx (idx <0 && idx>=size)

Case 2 : idx = 0

Case 3 : idx = size-1

Case 4 : id >0 && idx < size-1



T.C. : O(n)

S.C. : O(1)

Insertion

At Head : O(1)

1. size==0
2. size>0

At Tail : O(1)

1. size==0
2. size>0

At Any idx : O(n)

1. idx<0 || idx>size
2. idx==0
3. idx==size
4. idx>0 && idx<size

Deletion

At Head : O(1)

1. size==0
2. size>0

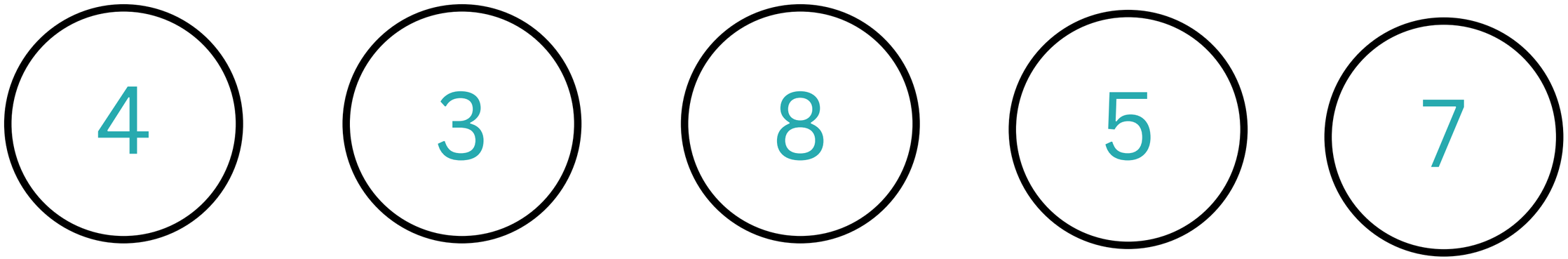
At Tail : O(n)

1. size==0
2. size>0

At Any idx : O(n)

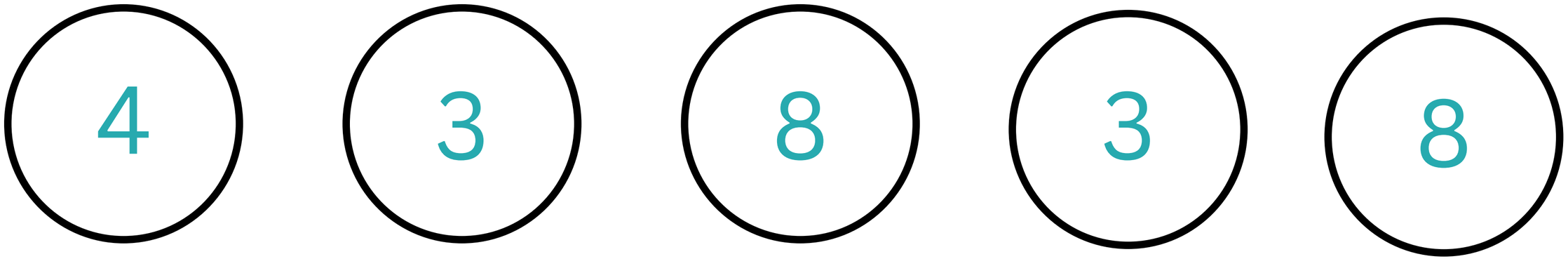
1. idx<0 || idx>=size
2. idx==0
3. idx==size-1
4. idx>0 && idx<size-1

Leetcode : 206. Reverse Linked List

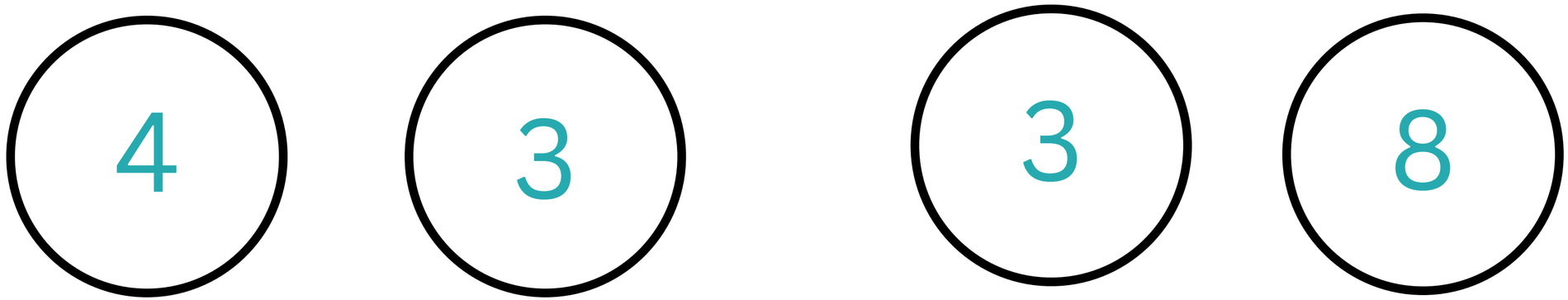


Leetcode : 237. Delete Node in a Linked List

Leetcode : 234. Pallindrome Linked List



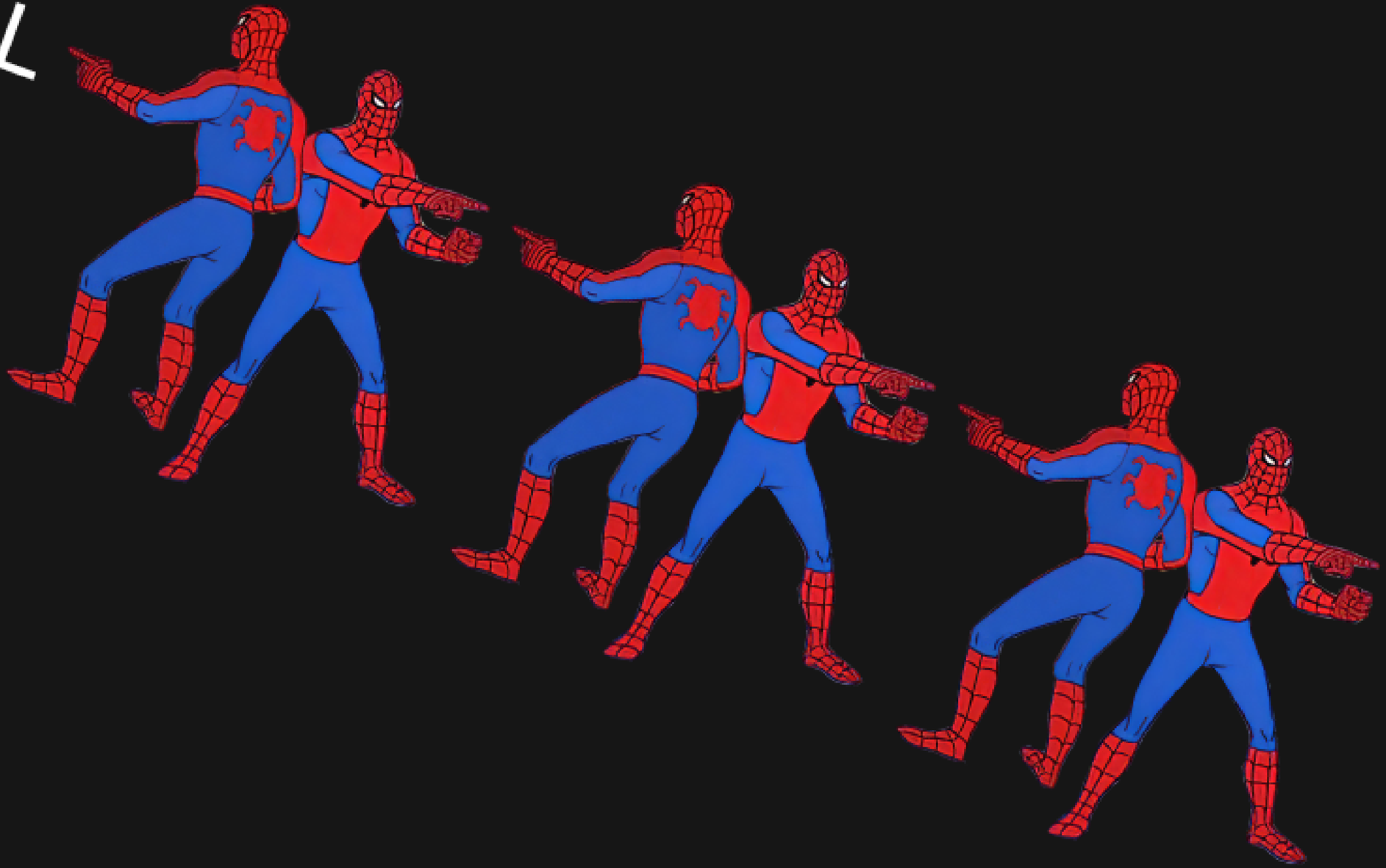
Leetcode : 234. Pallindrome Linked List



DOUBLY LINKED LIST

A list with forward and reverse gears

NULL



NULL

Structure

Representation

```
class Node {
public:
    int data;
    Node* next;
    Node* prev;
    Node(int data) {
        this->data = data;
        this->next = NULL;
        this->prev = NULL;
    }
};
```

Insertion at Head

1. Create a new node with the given data
2. Set the **next** pointer of the new node to point to the head node
3. Set the **prev** pointer of the head node to point to the new node
4. Update the head of the list to be the new node

Insertion at Head

```
void insertStart(int data) {
    Node* ptr = new Node(data);
    ptr→next = head;
    if (!isEmpty())
        head→prev = ptr;
    head = ptr;
}
```

Insertion at Tail

1. Create a new node with the given data
2. Traverse the list to find the current last node
3. Set the **next** pointer of the current last node to point to the new node
4. Set the **prev** pointer of the new node to point to the current last node

Insertion at Tail

```
void insertEnd(int data) {
    Node* ptr = new Node(data);
    if (isEmpty())
        head = ptr;
    else {
        Node* temp = head;
        while (temp→next ≠ NULL)
            temp = temp→next;
        temp→next = ptr;
        ptr→prev = temp;
    }
}
```

Insertion at any Index

1. Create a new node with the given data
2. Traverse the list to find the node immediately before the specified index
3. Update the **next** and **prev** pointers of the new node and the adjacent nodes

Insertion at any Index

```
void insertAtIndex(int data, int index) {
    Node* ptr = new Node(data);
    Node* temp = head;
    for (int i = 0; i < index; i++)
        temp = temp->next;
    temp->next->prev = ptr;
    ptr->next = temp->next;
    temp->next = ptr;
    ptr->prev = temp;
}
```

Deletion at Start

1. If the list is empty, return immediately
2. Store the head pointer in a **temp** variable
3. Update the **head** pointer to point to the next node
4. Update the **prev** pointer of the head node to NULL
5. Free the memory allocated for the **temp** node

Deletion at Head

```
void deleteStart(int data) {
    if (isEmpty())
        cout << "List khali hai!";
    else {
        Node *temp = head;
        head = head→next;
        head→prev = NULL;
        delete temp;
    }
}
```

Deletion at End

1. If the list is empty, return immediately
2. Traverse the linked list until you reach the second last node
3. Store the second last node in a **temp** variable
4. Free the memory allocated for the last node
5. Update the **next** pointer of **temp** to NULL

Deletion at End

```
void deleteEnd() {
    if (isEmpty())
        cout << "List khali hai!";
    else {
        Node *temp = head;
        while (temp->next->next != NULL)
            temp = temp->next;
        delete temp->next;
        temp->next = NULL;
    }
}
```

Deletion at any Index

1. If the list is empty or if the index is out of bounds, return immediately
2. Traverse the list until you reach the node at the given index
3. Store the node to be deleted in a **temp** variable
4. Update the **prev** and **next** pointers of the surrounding nodes
5. Free the memory allocated for the **temp** node

Deletion at any Index

```
void deleteAtIndex(int index) {
    if (isEmpty())
        cout << "List khali hai!";
    else {
        struct Node *temp = head;
        for (int i = 0; i < index; i++)
            temp = temp->next;
        temp->next = temp->next->next;
        delete temp->next->prev;
        temp->next->prev = temp;
    }
}
```

Traversal

```
void traverse() {
    Node* temp = head;
    cout << "Mera list: ";
    while (temp ≠ NULL) {
        cout << temp→data << " ";
        temp = temp→next;
    }
}
```

Applications

- Text Editors
- Web Browsers
- Music Players
- Image Viewer
- Keyboard Buffer
- Database Management Systems
- Network Routing

CIRCULAR LINKED LIST

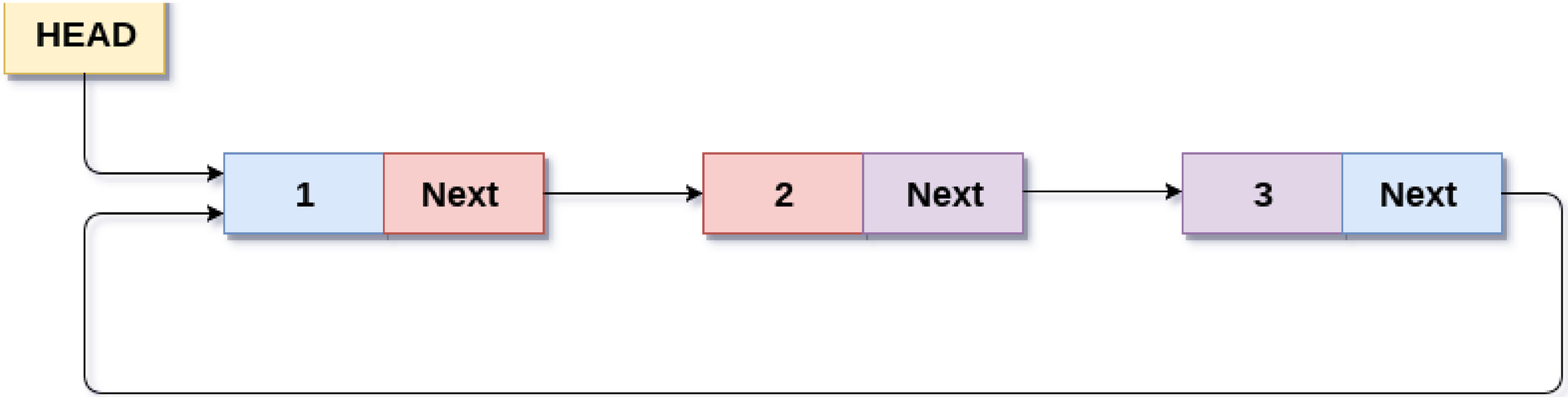
CIRCULAR LINKED LIST:

- In a circular Singly linked list, the **last node of the list contains a pointer to the first node** of the list. We can have a circular singly linked list as well as a circular doubly linked list.

- We traverse a circular singly linked list until we reach the same node where we started. The circular singly-liked list has **no beginning and no ending.**

- **No null** value is present in the next part of any of the nodes.

Structure



Standard Operations :

Standard Operations :

- [ ] Traversal
- [ ] Insertion of Node
- [ ] Deletion of Node

Traversal

- If the head is **NULL**, simply **return** because the list is **empty.**
- Create a node temp and make it point to the head of the list.
- Now, with the help of a **do-while loop**, keep printing the temp – > data and increment the temp, while temp is not equal to the head of the list.
- As we have chosen the head as our starting point, we are **terminating** the loop when we are reaching the **head again.**

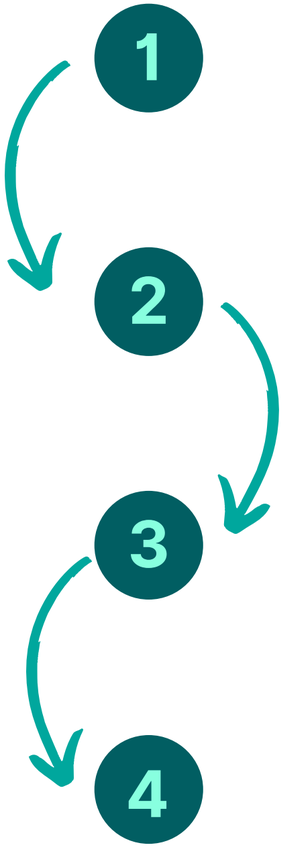
Code:

```c
void printList(struct Node *head)
{
    struct Node *temp = head;
    if (head != NULL)
    {
        do
        {
            printf("%d ",temp->data);
            temp = temp->next;
        }
        while (temp != head);
    }
```

Let's Solve

SOME

PROBLEMS



Cycle detection in linked list.

Finding length of cycle.

Finding starting node of the cycle.

Removal of cycle.

Cycle Detection in Linked List

Finding Length of Cycle

Finding Starting Node of Cycle

Removal of Cycle

Thank You

HAPPY LEARNING !