Linked List
Module 2.1

Sarika Dharangaonkar

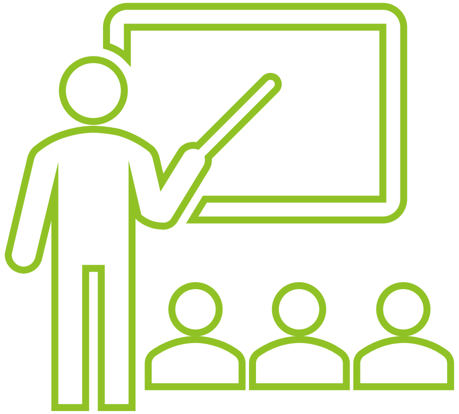
Assistant Professor,

Information Technology Department, KJSCE

Topics to be covered

- Introduction
- **Representation of Linked List**
- **Linked List vs Array**

Learning Objectives



At the end of the lecture, students will be able to

- Differentiate between linked list and array
- Represent Linked list in memory

Introduction

- **Array** is a linear collection of data elements in which the elements are stored in consecutive memory locations.

  Drawback:

  Need to specify the size of the array in advance

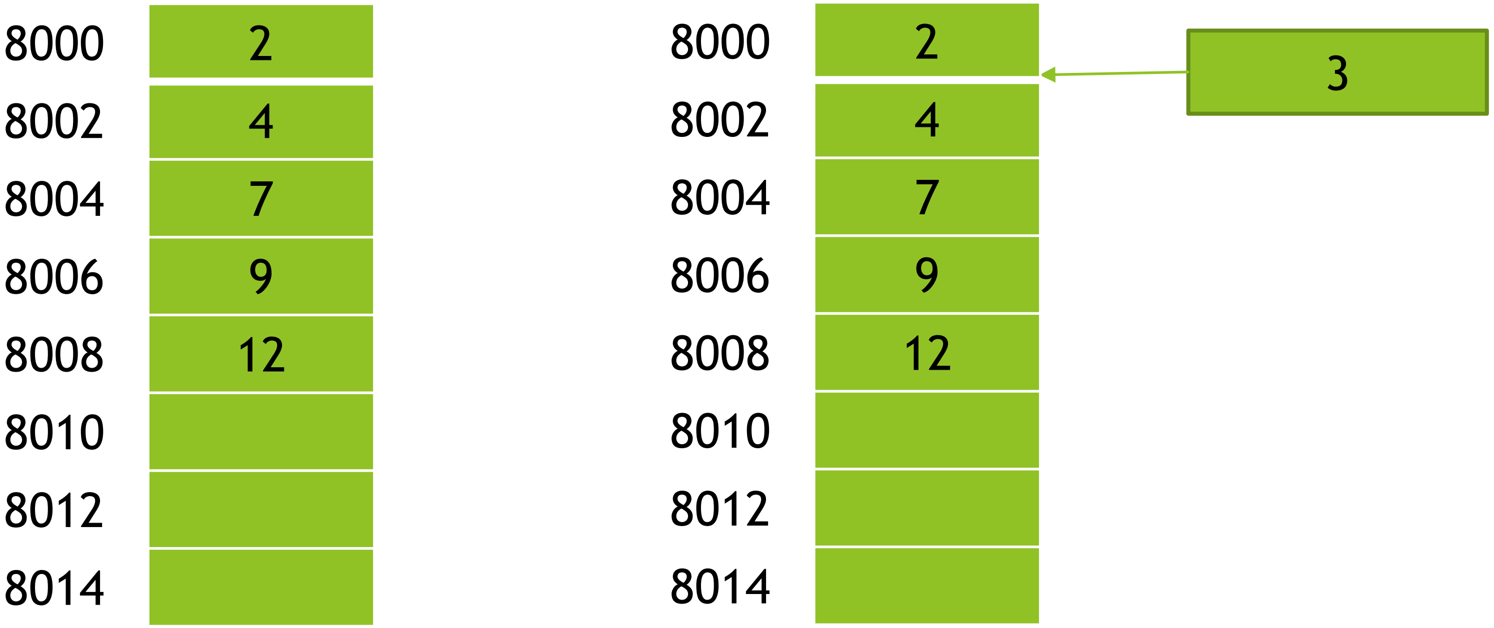
  Will restrict the number of elements that the array can store.

- **For efficient memory usage**

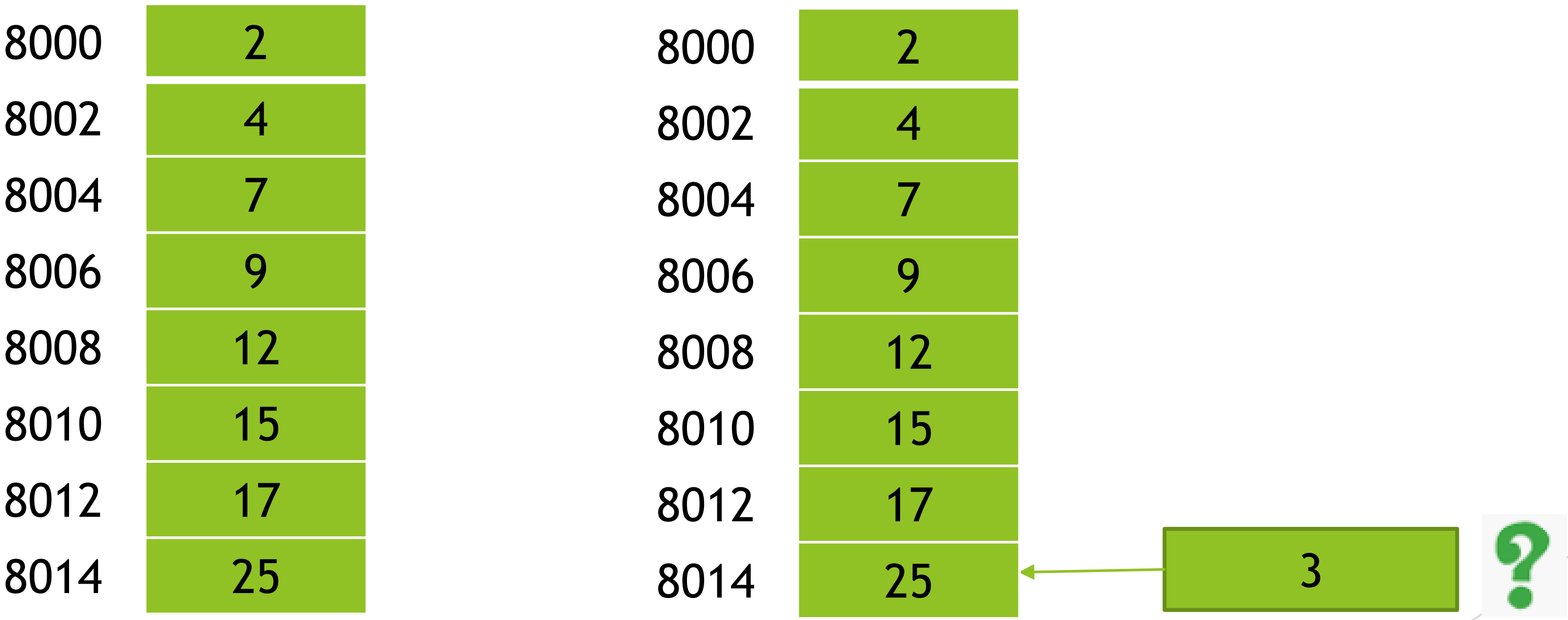
  The elements must be stored randomly at any location rather than in consecutive locations.

Array Disadvantage

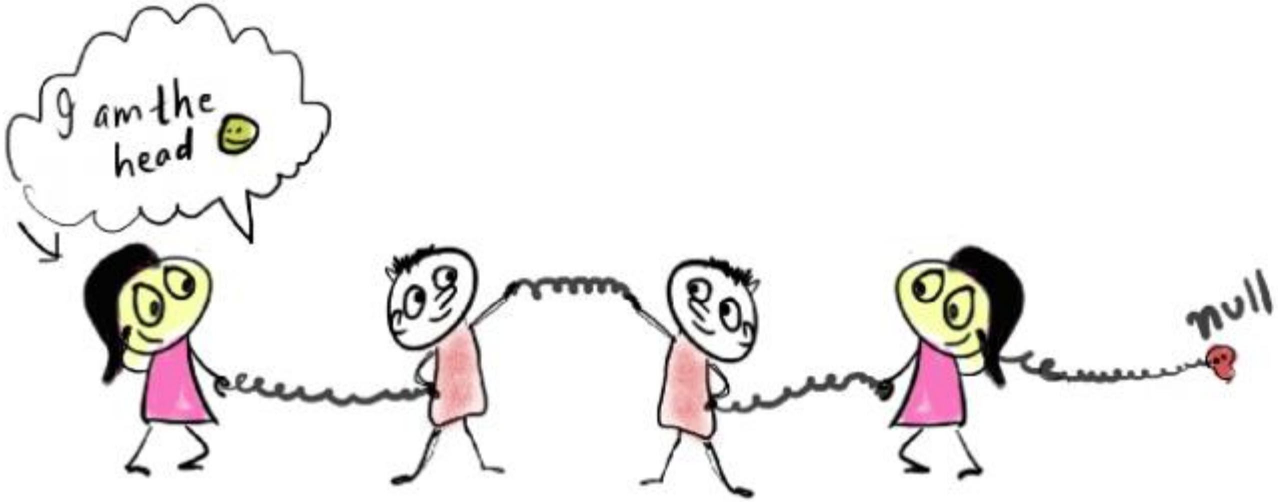
int n[8];



Array Disadvantage



Link List



Introduction

- **A linked list** does not store its elements in consecutive memory locations and the user can add any number of elements to it.

  Drawback:

  Does not allow random access of data.

  Elements in a linked list can be accessed only in a sequential manner.

  Extra space required for storing address of the next node.

  Advantage:

  Insertion and deletions can be done at any point in the list in a constant time.

Linked List

- Linked list is linear collection of data elements. These data elements are called *nodes*
- **Each node of the list has two elements:**
  1. The item being stored in the list
  2. A pointer to the next item in the list

Example of Data/Value in Node

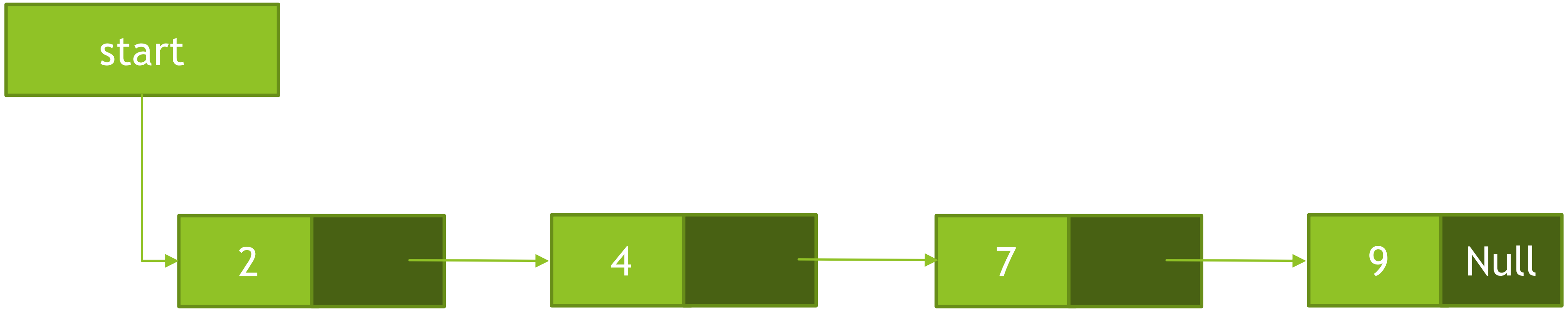
Item/Data	Pointer

apple	Pointer

78	Pointer

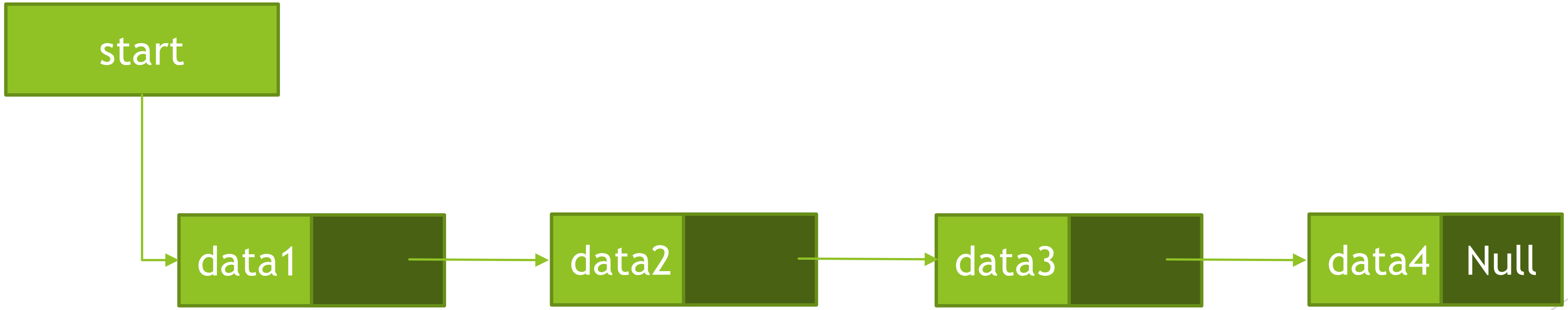
XYZ	90	Pointer

Link List Example



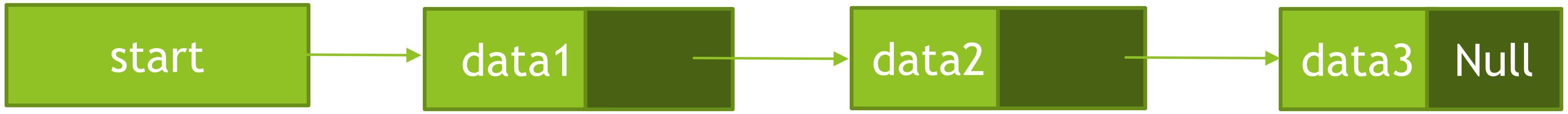
Linked List

- Linked list in which every node contains two parts
  - a **data item** and a **pointer** to the next node.
- Data item may include a simple data type, an array, or a structure.
- The last node will have no next node connected to it, it will store a special value called NULL.
- A NULL pointer denotes the end of the list



Importance of Start/Head pointer

- START pointer stores the address of the first node in the list.
- List traversal is possible only if we know where the list START
- The next part of the first node in turn stores the address of its succeeding node. Using this technique, the individual nodes of the list will form a chain of nodes.



- If START = NULL, then the linked list is empty and contains no nodes.

Linked list is self-referential data type

As in a linked list, every node contains a pointer to another node which is of the same type so it is called a *self-referential data type*.

Quiz

________ pointer stores the address of the first node in the list.

A. Data
B. Ptr
C. Next
D. Start

Quiz

__________does not allow random access of data.

A. Array
B. Linked List

Quiz

In my class there are 60 students attending my course of data structures.

Which linear data structure can be used to store the records of every student in the class. Also I should be able to access any students record in less time.

A. Array

B. Linked List

Quiz

If you want to store the names of list of participants appearing for the Debate Competition, which data structures will you prefer and why

A. Array

B. Linked List

References

- Data Structures using C, Reema Thareja, Oxford
- Data Structures: A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A., Forouzan, Second Edition, CENGAGE Learning

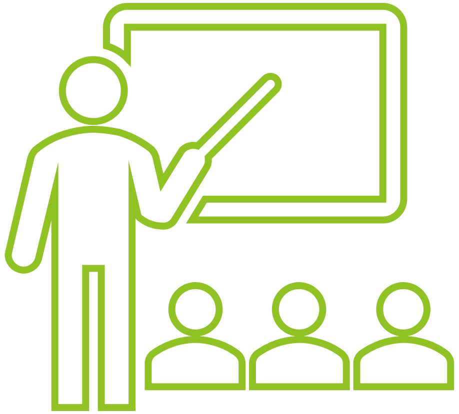
Linked List
Module 2.1

Sarika Dharangaonkar
Assistant Professor,
Information Technology Department, KJSCE

Topics to be covered

- Types of Linked List
- Singly **Linked List**
- **Basic Operations on Linked List**
- **Implementation of Singly Linked List**
  - **Create Node**
  - **Insert node at beginning of Linked List**
  - **Delete First node from Linked List**
  - **Display List**

Learning Objectives



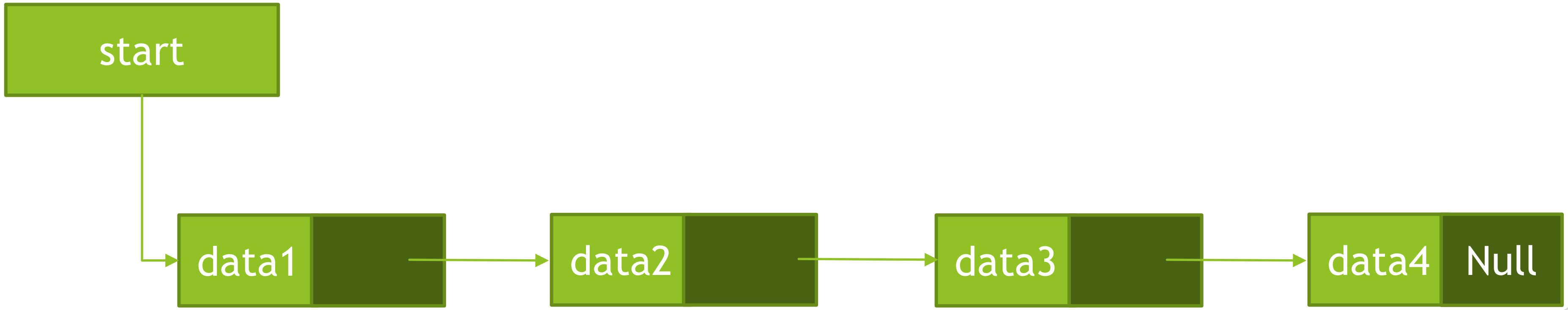
- Explain Singly Linked List
- List and perform the basic operations on the singly linked list
- Define a node and allocate a dynamic memory
- Perform Creation of node, INSERT node and Display linked list

Types of Linked List

- Singly Linked List
- Circular Linked List
- Doubly Linked

Singly Linked List

- A singly linked list is the simplest type of linked list in which every node contains some data and a pointer to the next node of the same data type.
- The node stores the address of the next node in sequence.
- A singly linked list allows traversal of data only in one way.



Basic List Operations

- ***Insertion*** is used to add a new element to the list.
- ***Deletion*** is used to remove an element from the list.
- ***Retrieval*** is used to get the information related to an element without changing the structure of the list.
- ***Traversal*** is used to traverse the list while applying a process to each element.

Operations on Linked List

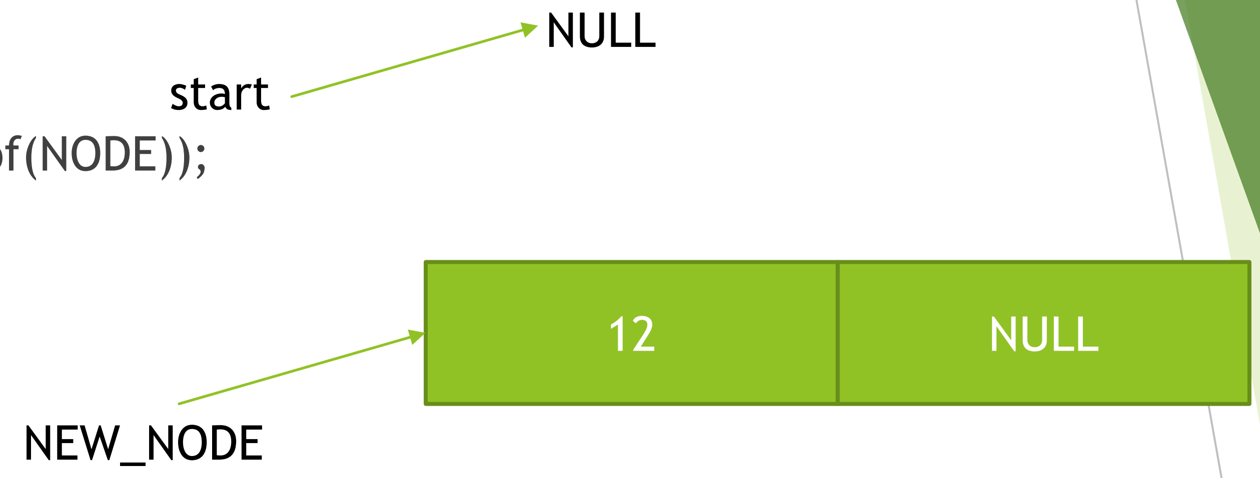
- Creation of Linked List
- Insertion of node
  - At the Beginning of the List
  - At the End of the List
  - In Between the List
- Deletion of node
  - From the Beginning of the List
  - From the End of the List
  - From in Between the List
- Display contents of List

Node of Linked List

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

Create a New Node

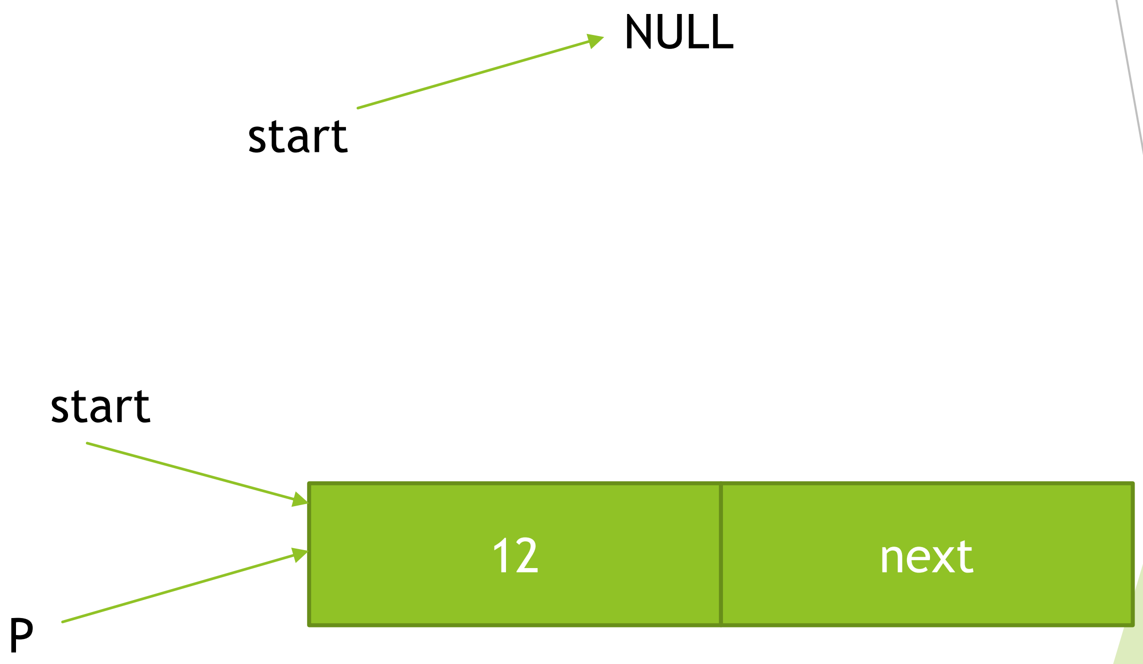
```
NODE *start=NULL,* NEW_NODE;
NEW_NODE=(NODE *)malloc(sizeof(NODE));
printf("\nEnter data item:");
scanf("%d",& NEW_NODE ->data);
NEW_NODE ->next=NULL
NODE *start=NEW_NODE
```



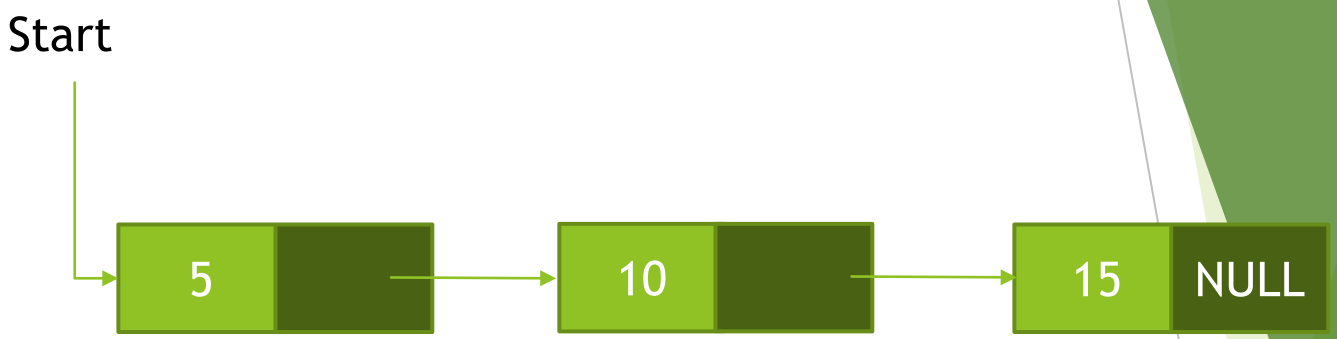
Insert the node in the empty List

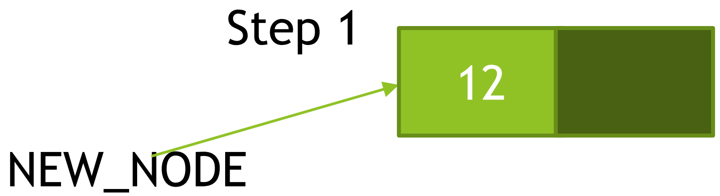
If List is Empty

```
if(start==NULL)
{
        start=p;
}
```



Insert node at beginning of Linked List



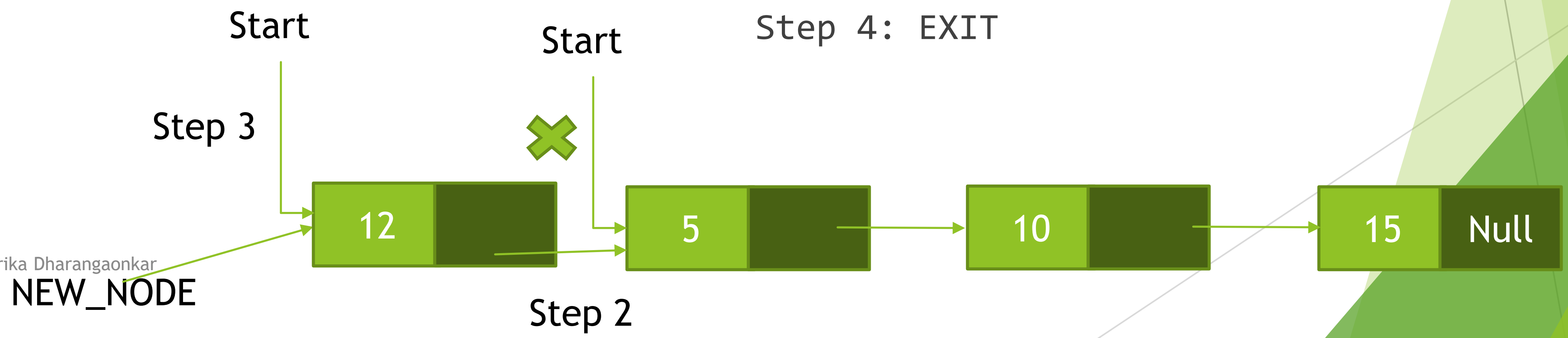


Step 1: Create NEW_NODE

Step 2: SET NEW_NODE NEXT = START

Step 3: SET START = NEW_NODE

Step 4: EXIT



Inserting a Node:

```
void insertStart (struct Node **head, int data)
{
 struct Node *newNode = (struct Node *) malloc (sizeof (struct Node));
 newNode - >  data = data;
 newNode - >  next = *head;
 //changing the new head to this freshly entered node
 *head = newNode;
}
```

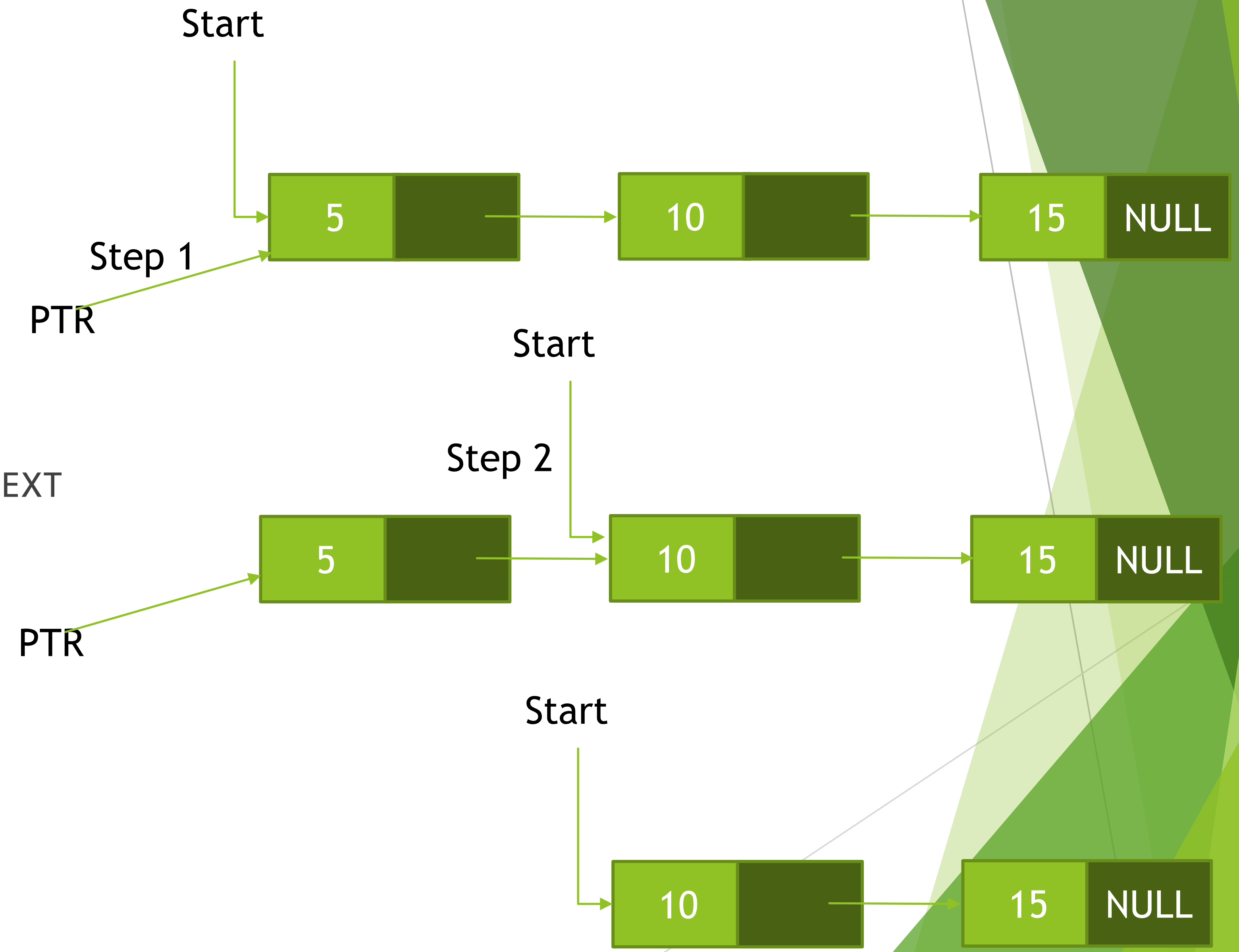
Traversing the Linked List

```
Step 1: [INITIALIZE] SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Display PTR->DATA
Step 4: SET PTR = PTR NEXT
[END OF LOOP]
Step 5: EXIT
```

```
void display(struct Node* node)
{
 printf("Linked List: ");

 // as linked list will end when Node is Null

  while(node!=NULL)
{
printf("%d ",node->data);
node = node->next;
 }
  printf("\n");
}
```

Delete the first node from Linked List

```
Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START NEXT
Step 4: FREE PTR
Step 5: EXIT
```



Deleting a Node:

```
void deleteStart(struct Node **head)
{
struct Node *temp = *head;
// if there are no nodes in Linked List can't delete
if (*head == NULL)
{
printf ("Linked List Empty, nothing to delete");
return;
}
// move head to next node
*head = (*head)->next;
free (temp);
}
```

References

- Data Structures using C, Reema Thareja, Oxford
- Data Structures: A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A., Forouzan, Second Edition, CENGAGE Learning

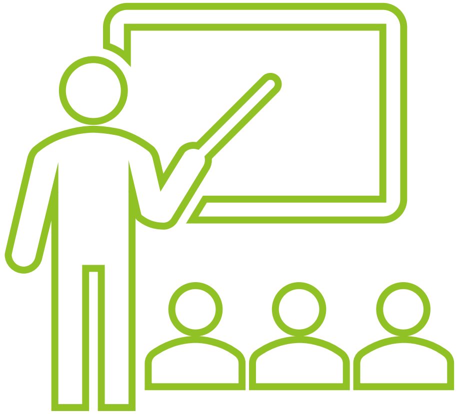
Linked List
Module 2.1

Sarika Dharangaonkar
Assistant Professor,
Information Technology Department, KJSCE

Topics to be covered

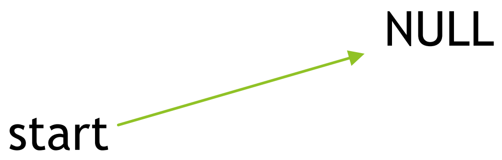
- **Implementation of Singly Linked List**
  - insert a new node at the end
  - insert a new node after a node that has value NUM
  - ***Deleting the Last Node from a Linked List***
  - ***Deleting the Node After a Given Node in a Linked List***

Learning Objectives

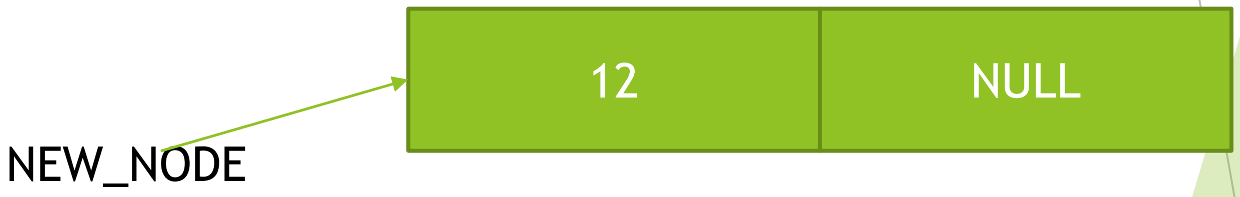


- Perform INSERT node, Delete node and Display linked list

Create a New Node




```
NODE *start=NULL,* NEW_NODE;
NEW_NODE=(NODE *)malloc(sizeof(NODE));
printf("\nEnter data item:");
scanf("%d",& NEW_NODE ->data);
NEW_NODE ->next=NULL
```



Insert a new node at the end of Linked List

Step 1: Create NEW_NODE

Step 2: SET PTR = START

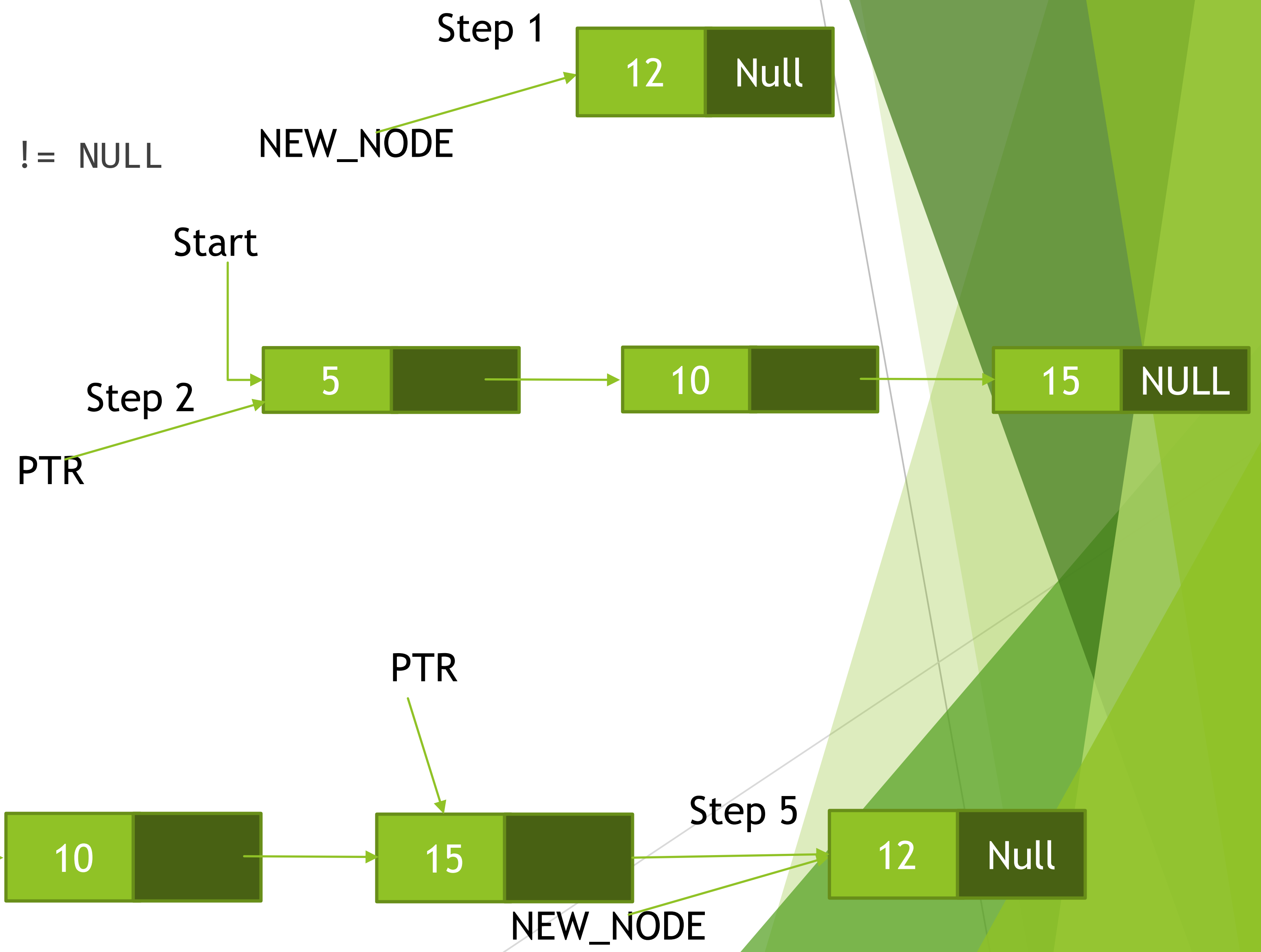
Step 3: Repeat Step 4 while PTR NEXT != NULL

Step 4: SET PTR = PTR NEXT

[END OF LOOP]

Step 5: SET PTR NEXT = NEW_NODE

Step 6: EXIT



```
// Function to insert a node at the end of the linked list
void insertAtEnd(struct Node** head, int newData)
{
 // Create a new node
struct Node* newNode = (struct Node*)malloc(sizeof(struct
Node));
newNode->data = newData;
newNode->next = NULL;
// If the list is empty, make the new node the head
if (*head == NULL)
{
 *head = newNode;
 return;
  }    // Traverse to the last node
 struct Node* current = *head;
while (current->next != NULL)
{
current = current->next;
 }
                                                // Connect the new node to the last node
                                                current->next = newNode;
                                                }
```

Inserting a node after a given node:

```
Initial Linked List: 10 -> 20 -> 30 -> 40 -> NULL

Step 1: Start from the head (10)
Step 2: Traverse to the node with value 20
Step 3: Create a new node (25)
Step 4: Set 'next' of 25 to point to 30
Step 5: Update 'next' of 20 to point to 25

Final Linked List: 10 -> 20 -> 25 -> 30 -> 40 -> NULL
```

Insert a new node after a node that has value NUM

Step 1: Create NEW_NODE

Step 2: SET PTR = START

Step 3: SET PREV = PTR

Step 4: Repeat Steps 8 and 9 while != NUM

Step 5: SET PREV = PTR

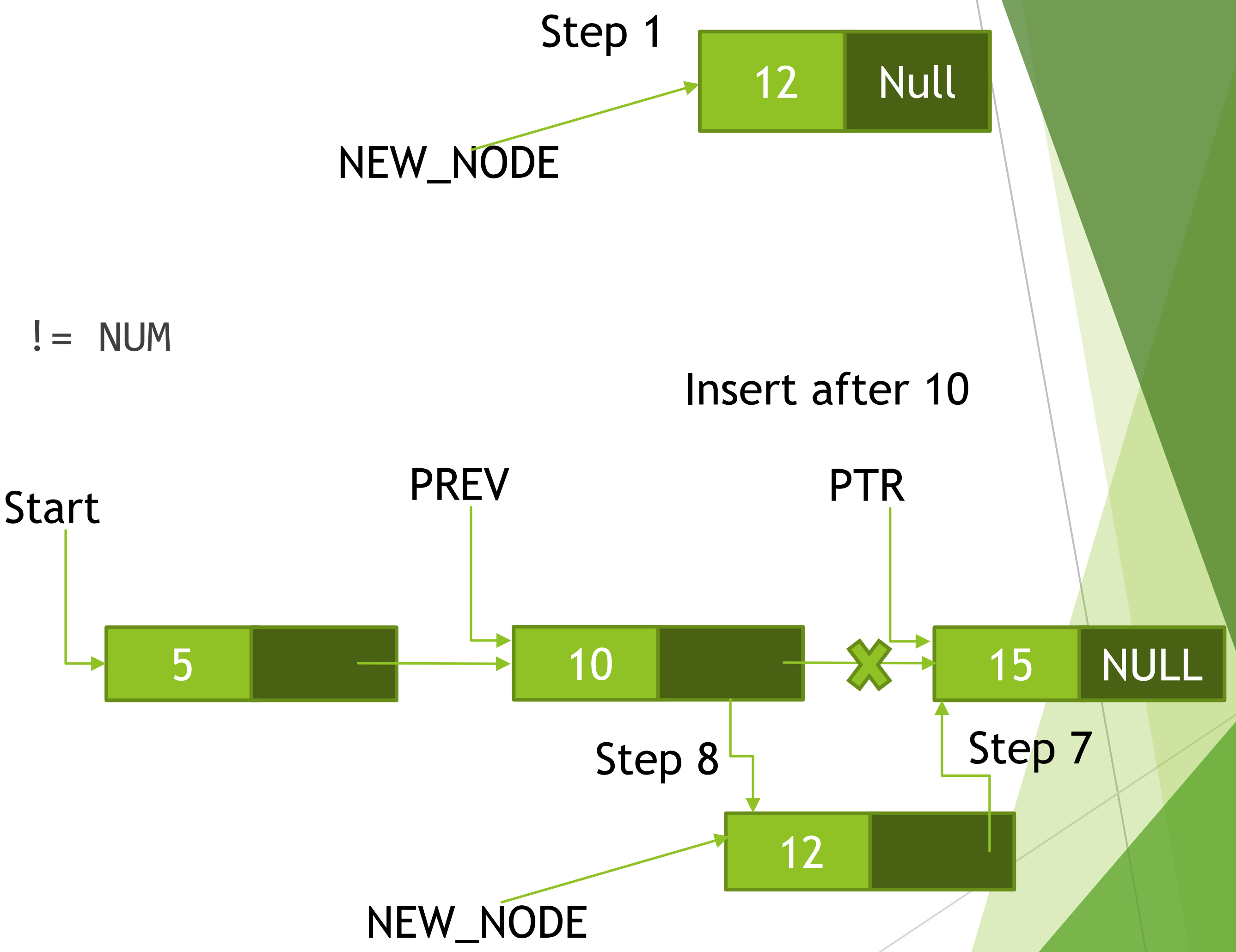
Step 6: SET PTR = PTR NEXT

[END OF LOOP]

Step 7: SET NEW_NODE NEXT = PTR

Step 8: PREV NEXT = NEW_NODE

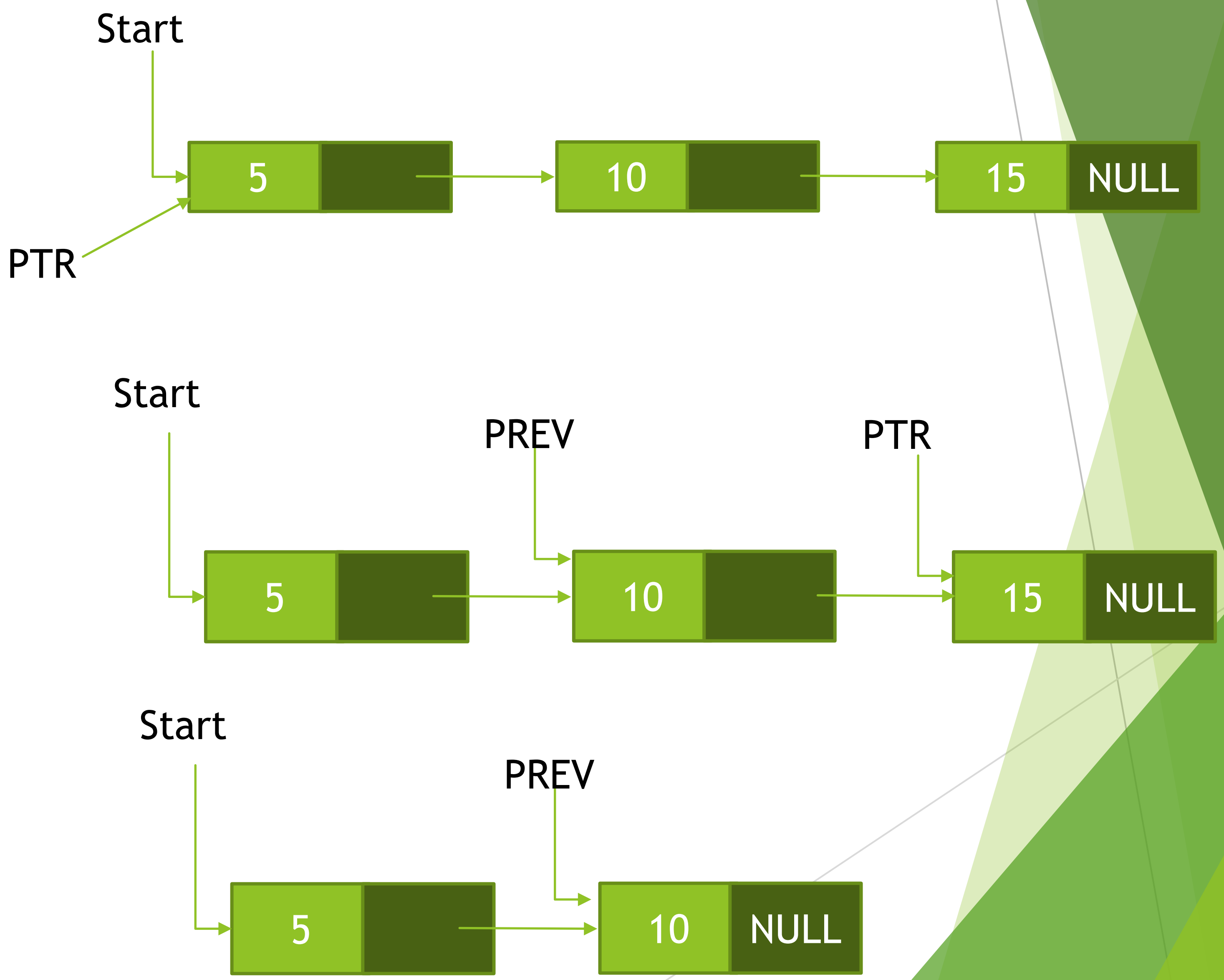
Step 9: EXIT



```
// Function to insert a new node with 'value' after the node with 'num'
void insertAfter(struct Node* start, int num, int value)
 {
 // Create a new node
struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
new_node->data = value;  // insert value in newnode
struct Node* ptr = start;  // Pointer to traverse the list
 struct Node* prev = ptr;  // Pointer to keep track of the previous node
// Traverse the list to find the node with 'num'
while (ptr != NULL && ptr->data != num)
{
prev = ptr;   // Keep track of the previous node
ptr = ptr->next;
}
Prev-> next = new_node; // The new node's next points to the node after 'ptr'

 new_node -> next = ptr; // 'ptr' now points to the new node
 }
```

Delete Last node from the Linked List

Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5
while PTR NEXT != NULL
Step 4: SET PREV = PTR
Step 5: SET PTR = PTR NEXT
[END OF LOOP]
Step 6: SET PREV NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT



```
void deleteEnd(struct Node** start)
{
// Step 1: Check if the linked list is empty
if (*start == NULL)
{
printf("UNDERFLOW: Linked list is empty.\n");
return;
 }
struct Node* ptr = *start;
 struct Node* prev = ptr;
// Step 3: Traverse the linked list to find the last
node
while (ptr->next != NULL)
{
prev = ptr;
 ptr = ptr->next;
}
```

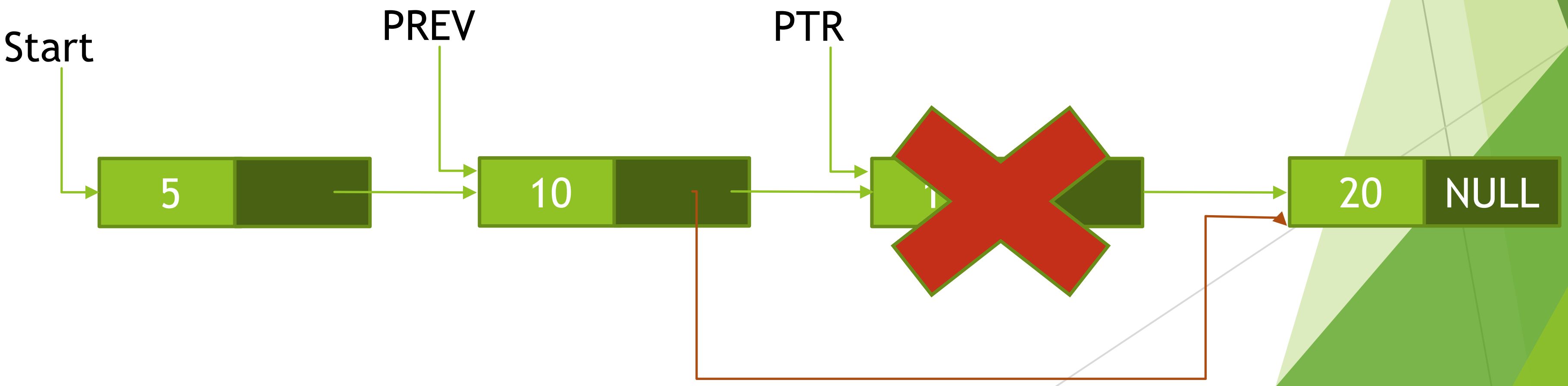


```
// Step 6: Set the previous node's next to NULL

prev->next = NULL;

 // Step 7: Free the memory of the last node

free(ptr);
printf("Last node deleted.\n"); }
```

Delete a node after a given node in the Linked List

```
Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 10
[END OF IF]
Step 2: SET PTR = START
Step 3: SET PREV = PTR
Step 4: Repeat Steps 5 and 6 while PREV DATA != NUM
Step 5: SET PREV = PTR
Step 6: SET PTR = PTR NEXT
[END OF LOOP]
Step 7: SET PREV NEXT = PTR NEXT
Step 8: FREE PTR
Step 9 : EXIT
```

Delete node after node with data 10



```c
void deleteNode(struct Node** start, int num)
{
// Step 1: Check if the linked list is empty
 if (*start == NULL)
{
 printf("UNDERFLOW: Linked list is empty.\n");
 return;
}
 struct Node* ptr = *start;
 struct Node* prev = ptr;
// Step 4: Traverse the linked list to find the node with value
'NUM'
while (prev->data != num && ptr != NULL)
{ prev = ptr;
// Step 5: Set PREV to PTR to keep track of the previous node
ptr = ptr->next;
// Step 6: Move PTR to the next node
}
 // If node with value 'NUM' found
if (ptr != NULL)
{
// Step 7: Disconnect the node to be
deleted
prev->next = ptr->next;
// Step 8: Free the memory of the
node to be deleted
free(ptr);
printf("Node with value %d deleted.\n",
num);
}
Else
 {
printf("Node with value %d not
found.\n", num); } }
```

Insert a node at Given position in Linked List

Example: Let's say we have a linked list: 10 -> 20 -> 40 -> NULL, and we want to insert a new node with the value 30 at position 3.

- Create a new node with value 30.
- Traverse to the node just before position i.e. 2 (20).
- Adjust pointers: Make the node with value 20 point to the new node, and the new node point to the node that used to be at position 3 (40).
- The linked list becomes: 10 -> 20 -> 30 -> 40 -> NULL.

pseudo-code

```
class Node:
data // Holds the data value of the node
next // Points to the next node in the linked list
function insertNodeAtPosition(head, data, pos):   // Create a new node with
the provided data
newNode = Node(data)
 if pos == 0:          // Inserting at the beginning of the list
newNode.next = head    // Point the new node's next to the current head
head = newNode         // Update the head to the new node
return head            // Initialize variables to traverse the list
current = head         // Start traversal from the head
position = 0           // Keeps track of the current position in the list
// Traverse the list to find the node just before the insertion position
while position < pos - 1 and current.next is not null:
current = current.next     // Move to the next node
position = position + 1    // Increment position
if current is null:     // Position is out of bounds
print("Invalid position")
return head
// Insert the new node at the specified position
newNode.next = current.next // Point the new node's next to the
node at the current position
current.next = newNode // Update the current node's next to the
new node return head
```

```
// Function to insert a node at a specific position
Node* insertNodeAtPosition(Node* head, int data, int pos)
 {
// Create a new node
Node* newNode = (Node*)malloc(sizeof(Node));-
newNode->data = data;
newNode->next = NULL;
if (pos == 0)              // Inserting at the beginning (position 0)
{
newNode->next = head;
head = newNode;
return head; }
 // Inserting at a non-zero position
Node* currentNode = head;
int currentPosition = 0;
// Traverse to the node just before the position to insert
while (currentPosition < pos - 1 && currentNode->next != NULL)
{
currentNode = currentNode->next; currentPosition++;
}
```

```
// Check if position is out of bounds
if (currentNode == NULL)
{
printf("Position is out of bounds.\n");
free(newNode);
// Free memory of the new node return head;
}
// Update pointers to insert the new node
newNode->next = currentNode->next;
currentNode->next = newNode;
 return head;
 }
```

Deleting a node at Given position

Example: Let's say we have a linked list: 10 -> 20 -> 30 -> 40 -> NULL, and we want to delete the node at position 3 (30).

- We start from the head (node with value 10) and move to the node just before position i.e. 2 (20).
- Adjust pointers: We make the node with value 20 point to the node after position 4 (40), effectively bypassing the node with value 30.
- We free the memory of the node with value 30.
- The linked list becomes: 10 -> 20 -> 40 -> NULL.

pseudo-code

```
class Node:
    data  // Holds the data value of the node
    next  // Points to the next node in the linked list

function deleteNodeAtPosition(head, pos):
    if head is null:  // Check if the list is empty
        print("List is empty")
        return null

    if pos == 0:  // Deleting the first node (position 0)
        temp = head  // Store the head node to free its memory later
        head = head.next  // Move head to the next node
        free(temp)  // Free memory of the deleted node
        return head

    // Initialize variables to traverse the list
    current = head  // Start traversal from the head
    position = 0  // Keeps track of the current position in the list

// Traverse the list to find the node just before the deletion position
    while position < pos - 1 and current.next is not null:
        current = current.next  // Move to the next node
        position = position + 1  // Increment position

    if current is null or current.next is null:  // Position is out of bounds
        print("Invalid position")
        return head

    temp = current.next  // Node to be deleted
    current.next = temp.next  // Update previous node's next pointer to skip the deleted node
    free(temp)  // Free memory of the deleted node

    return head
```

```
// Function to delete a node at a specific position
Node* deleteNodeAtPosition(Node* head, int pos)
 { // Check if the list is empty
if (head == NULL)
 {
printf("List is empty.\n");
return head; }
 // Deleting the first node (position 0)
if (pos == 0)
{
Node* temp = head;
head = head->next; // Move head to the next node
free(temp); // Free memory of the deleted node
return head; }
// Deleting a node at a non-zero position
Node* currentNode = head;
int currentPosition = 0;
```

```
// Traverse to the node just before the one to be deleted

while (currentPosition < pos - 1 && currentNode->next != NULL)
{
currentNode = currentNode->next;
currentPosition++; } // Check if position is out of bounds
if (currentNode == NULL || currentNode->next == NULL)
 {
printf("Position is out of bounds.\n");
return head;
 }
Node* temp = currentNode->next; // Node to be deleted
currentNode->next = temp->next; // Update previous node's next pointer
free(temp); // Free memory of the deleted node
return head;
}
```

References

- Data Structures using C, Reema Thareja, Oxford
- Data Structures: A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A., Forouzan, Second Edition, CENGAGE Learning

Linked List
Sem-III

Ms. Sarika Dharangaonkar

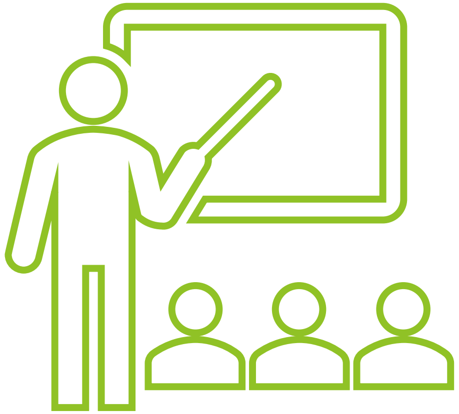
Assistant Professor,

Department of Information Technology, KJSCE

Topics to be covered

- Circular Linked List

Learning Objectives

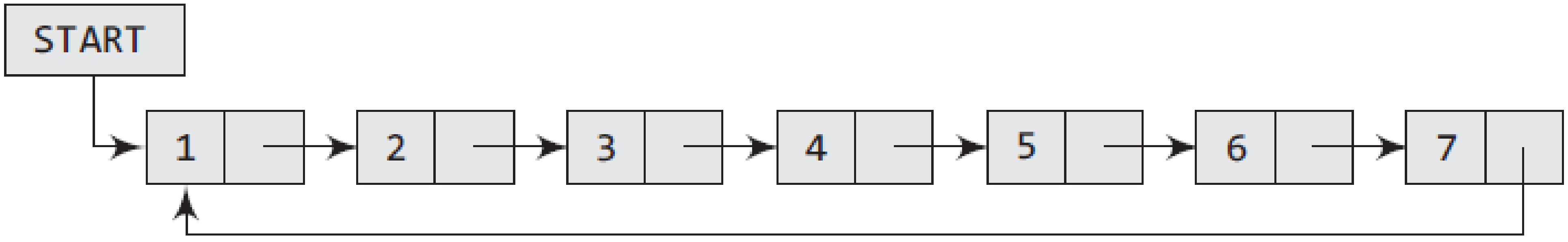


At the end of the lecture, students will be able to

- perform various operations such as insertion, deletion, searching and traversing over circular linked list

Introduction

- In a circular linked list, the last node contains a pointer to the first node of the list.
- While traversing a circular inked list, we can begin at any node and traverse the list in any direction, forward or backward, until we reach the same node where we started. Thus, a circular linked list has no beginning and no ending.
- The only downside of a circular linked list is the complexity of iteration. Note that there are no NULL values in the NEXT part of any of the nodes of list.



Introduction

- Circular linked lists are widely used in operating systems for task maintenance
- *When we are surfing the Internet, we can use the Back button and the Forward button to move to the previous pages that we have already visited. How is this done? The answer is simple. A circular linked list is used to maintain the sequence of the Web pages visited. Traversing this circular linked list either in forward or backward direction helps to revisit the pages again using Back and Forward buttons. Actually, this is done using either the circular stack or the circular queue.*
- We can traverse the list until we find the NEXT entry that contains the address of the first node of the list. This denotes the end of the linked list, that is, the node that contains the address of the first node is actually the last node of the list.

Circular Linked List

Circular Linked List Operations:

- Insertion
- Deletion
- Traversal

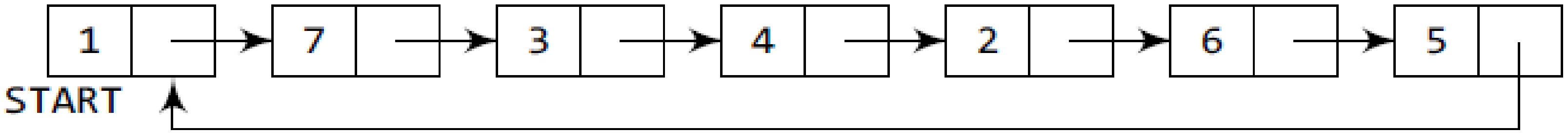
Inserting a New Node in a Circular Linked List

- Case 1: The new node is inserted at the beginning of the circular linked list.
- Case 2: The new node is inserted at the end of the circular linked list.

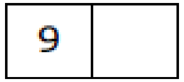
The new node is inserted at the beginning of the circular linked list.

```
Step 1: IF AVAIL = NULL
                Write OVERFLOW
                Go to Step 11
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = START
Step 6: Repeat Step 7 while PTR -> NEXT != START
Step 7:         PTR = PTR -> NEXT
         [END OF LOOP]
Step 8: SET NEW_NODE -> NEXT = START
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: SET START = NEW_NODE
Step 11: EXIT
```

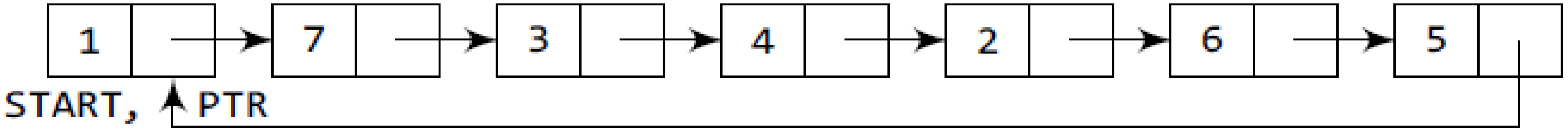
The new node is inserted at the beginning of the circular linked list.



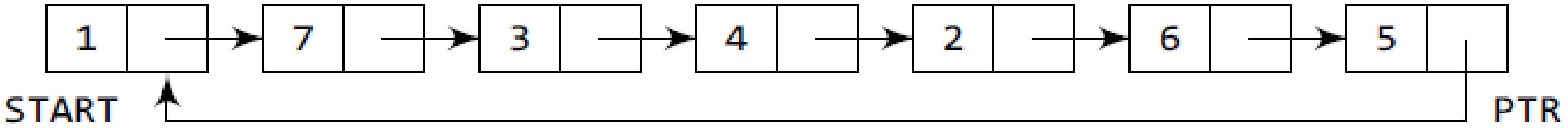
Allocate memory for the new node and initialize its DATA part to 9.



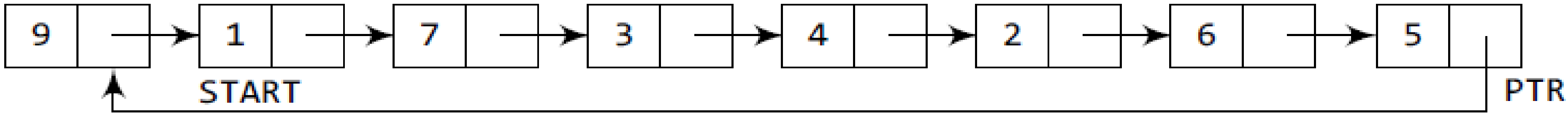
Take a pointer variable PTR that points to the START node of the list.



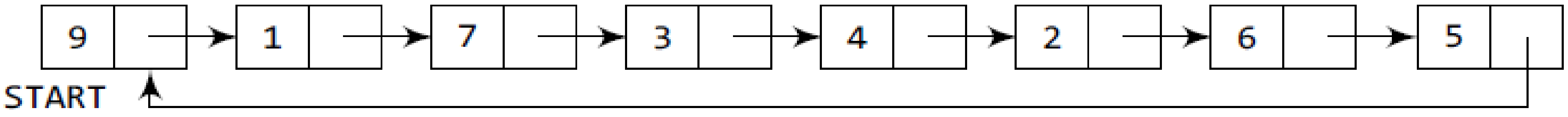
Move PTR so that it now points to the last node of the list.



Add the new node in between PTR and START.



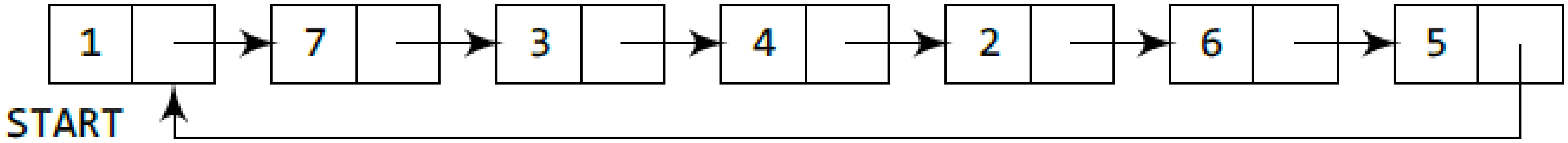
Make START point to the new node.



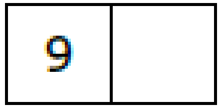
```
void insertAtBeginning(struct Node** start, int newData)
 {
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node)); // Create a new node
newNode->data = newData;      // Check if the circular linked list is empty
if (*start == NULL)
 {
newNode->next = newNode;     // Point to itself for the circular linkage
 }
else
{ struct Node* last = *start;
             // Traverse to the last node in the current circular linked list
while (last->next != *start) { last = last->next; }
newNode->next = *start;                // Set new node's next to the current start node
 last->next = newNode;                 // Update the last node's next to the new node
 }
*start = newNode;                      // Update the start pointer to the new node
 }
```

Inserting a Node at the End of a Circular Linked List

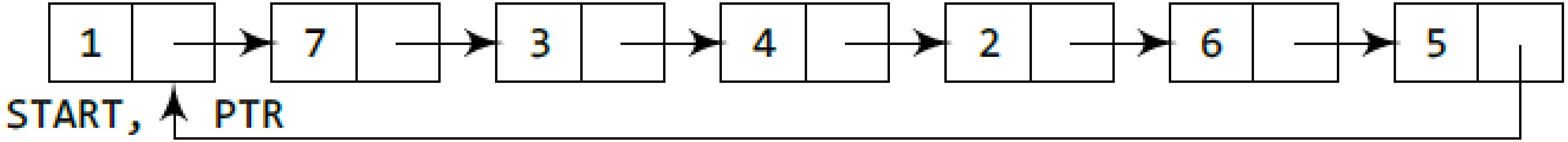
```
Step 1: IF AVAIL = NULL
              Write OVERFLOW
              Go to Step 10
        [END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = START
Step 6: SET PTR = START
Step 7: Repeat Step 8 while PTR -> NEXT != START
Step 8:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```



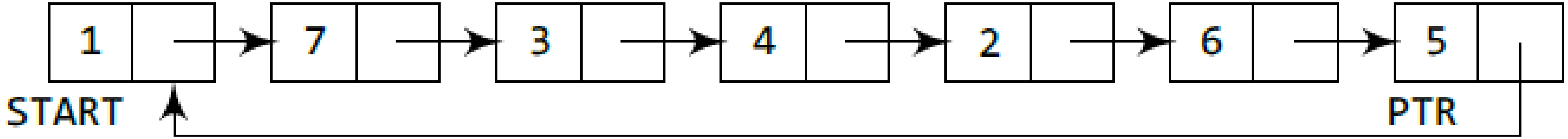
Allocate memory for the new node and initialize its DATA part to 9.



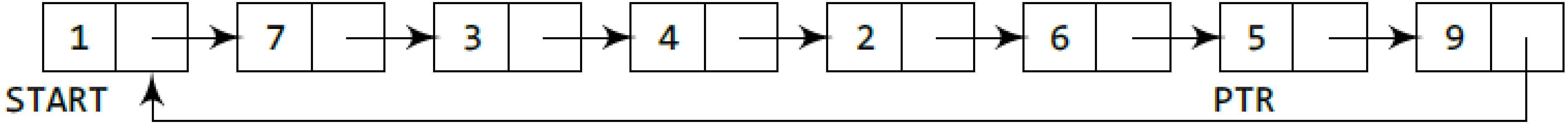
Take a pointer variable PTR which will initially point to START.



Move PTR so that it now points to the last node of the list.



Add the new node after the node pointed by PTR.



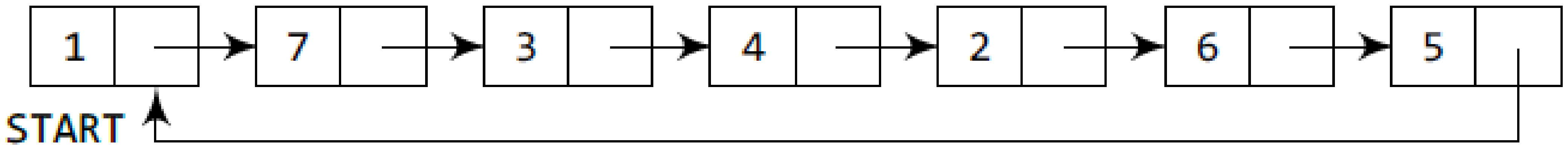
```
void insertAtEnd(struct Node** start, int newData)
{
struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));        // Create a new node
newNode->data = newData;                                 // Check if the circular linked list is empty
if (*start == NULL)
{
newNode->next = newNode;                                 // Point to itself for the circular linkage
*start = newNode;                             // Update the start pointer
} else
 {
struct Node* last = *start;                              // Traverse to the last node in the current circular linked list
while (last->next != *start)
{ last = last->next;
 }
last->next = newNode;                          // Update the last node's next to the new node
newNode->next = *start;                    // Set new node's next to the current start node
} }
```

Deleting a Node from a Circular Linked List

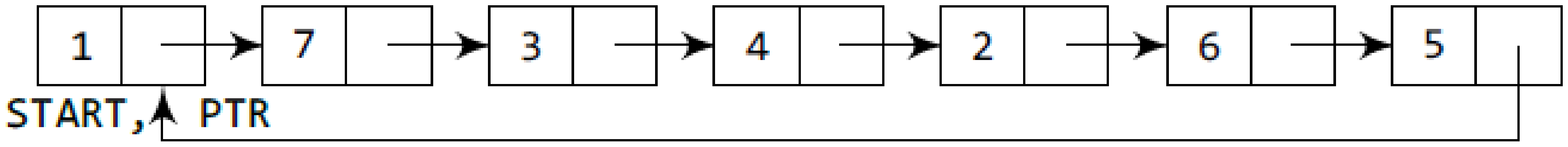
- Case 1: The first node is deleted.
- Case 2: The last node is deleted.

Deleting the First Node

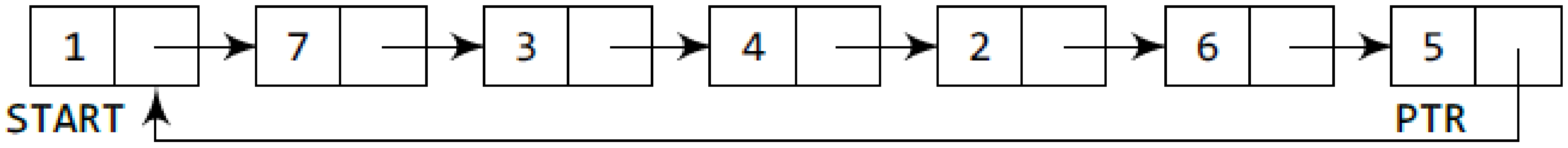
```
Step 1: IF START = NULL
              Write UNDERFLOW
              Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR -> NEXT != START
Step 4:       SET PTR = PTR -> NEXT
        [END OF LOOP]
Step 5: SET PTR -> NEXT = START -> NEXT
Step 6: FREE START
Step 7: SET START = PTR -> NEXT
Step 8: EXIT
```



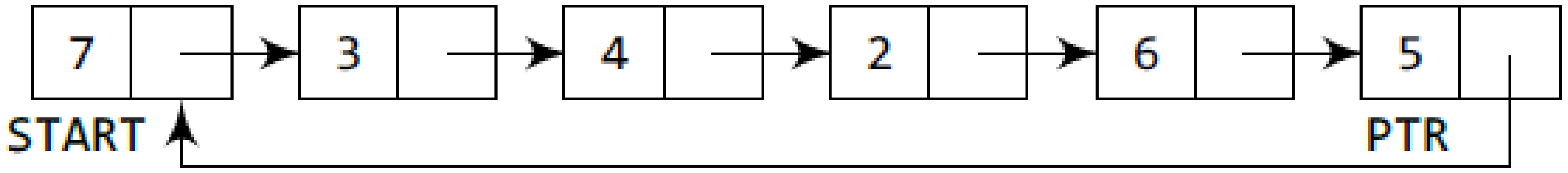
Take a variable PTR and make it point to the START node of the list.



Move PTR further so that it now points to the last node of the list.

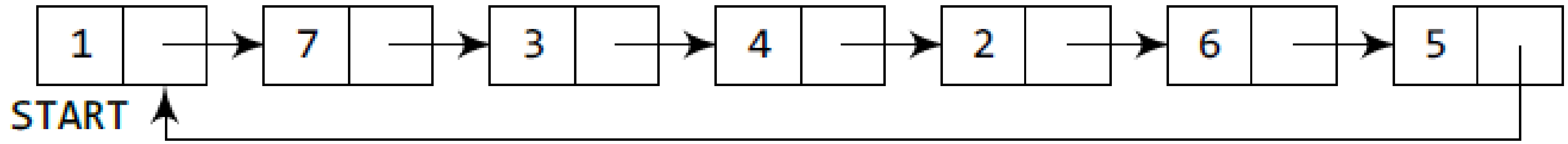


The NEXT part of PTR is made to point to the second node of the list and the memory of the first node is freed. The second node becomes the first node of the list.

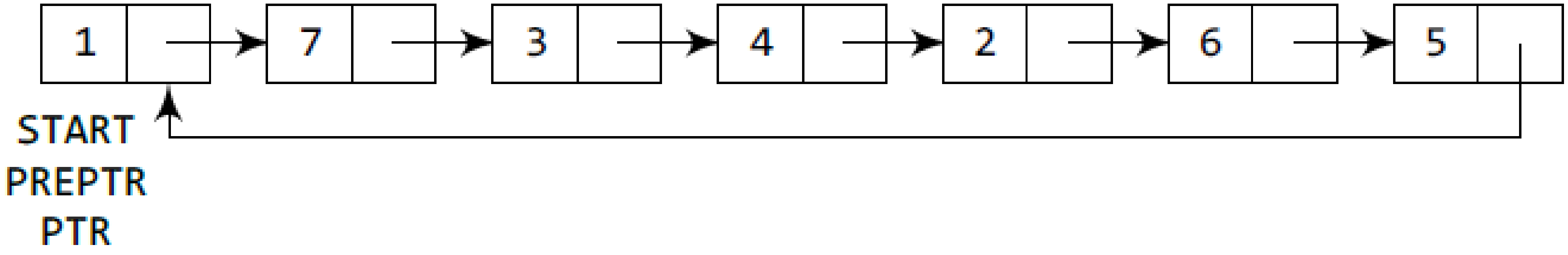


Deleting the last Node

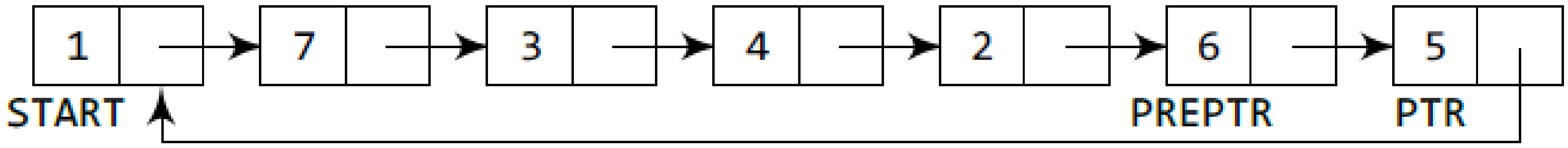
```
Step 1: IF START = NULL
               Write UNDERFLOW
               Go to Step 8
         [END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != START
Step 4:        SET PREPTR = PTR
Step 5:        SET PTR = PTR -> NEXT
         [END OF LOOP]
Step 6: SET PREPTR -> NEXT = START
Step 7: FREE PTR
Step 8: EXIT
```



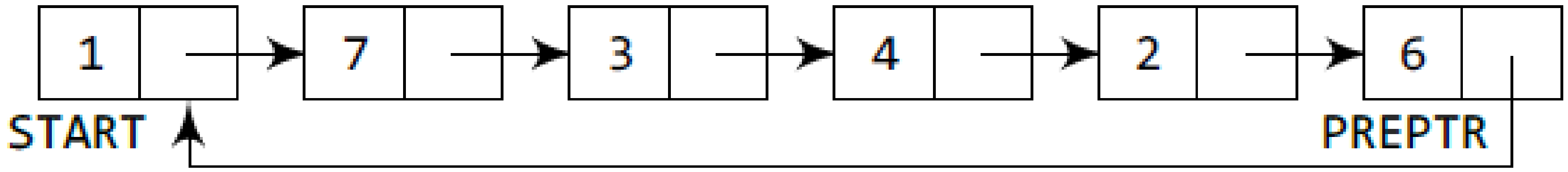
Take two pointers PREPTR and PTR which will initially point to START.



Move PTR so that it points to the last node of the list. PREPTR will always point to the node preceding PTR.



Make the PREPTR's next part store START node's address and free the space allocated for PTR. Now PREPTR is the last node of the list.



References

- Data Structures using C, Reema Thareja, Oxford
- Data Structures: A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A., Forouzan, Second Edition, CENGAGE Learning

Linked List
Sem-III

Ms. Sarika Dharangaonkar

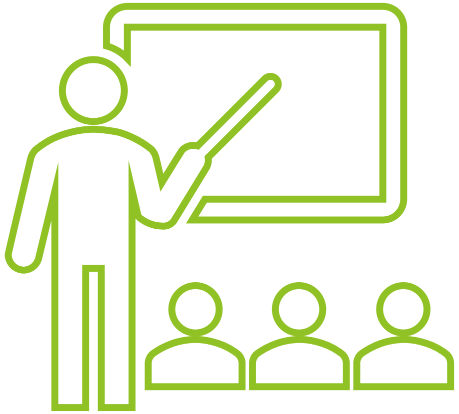
Assistant Professor,

Department of Information Technology, KJSCE

Topics to be covered

- Doubly Linked List

Learning Objectives



At the end of the lecture, students will be able to

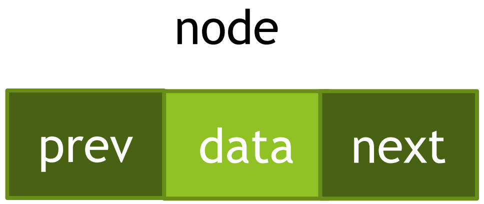
- Differentiate between singly and doubly linked list
- List the advantage and drawback of doubly linked list
- Create doubly linked list
- Perform basic operations on doubly linked list

Doubly Linked List

- A doubly linked list or a two-way linked list is a more complex type of linked list which contains a pointer to the next as well as the previous node in the sequence.
- Therefore, it consists of three parts—data, a pointer to the next node, and a pointer to the previous node

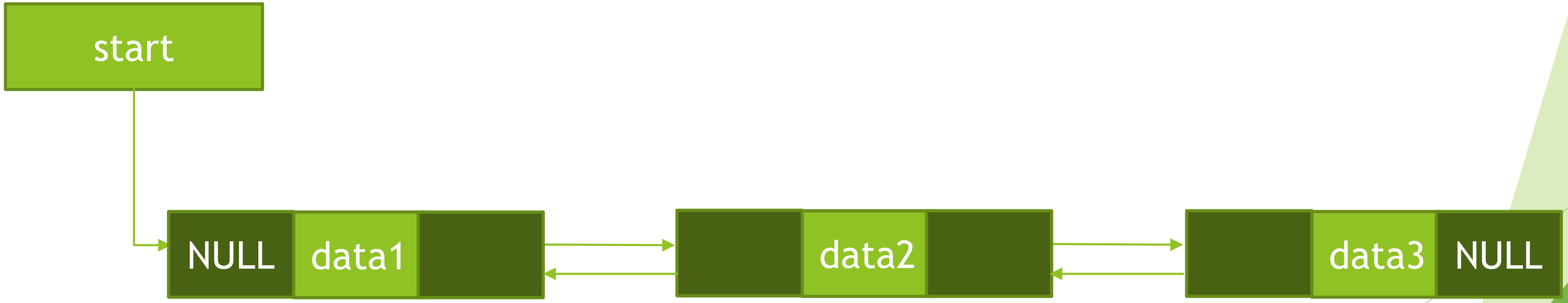
Doubly Linked List Node

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
};
```



Doubly Linked List

- The **`prev`** field of the first node and the **`next`** field of the last node will contain `NULL`.
- The **`prev`** field is used to store the address of the preceding node, which enables us to traverse the list in the backward direction.



Doubly Linked List

Drawback

- Thus, we see that a doubly linked list calls for more space per node and more expensive basic operations.
- List manipulations are slower (because more links must be changed)
- Greater chance of having bugs (because more links must be manipulated)

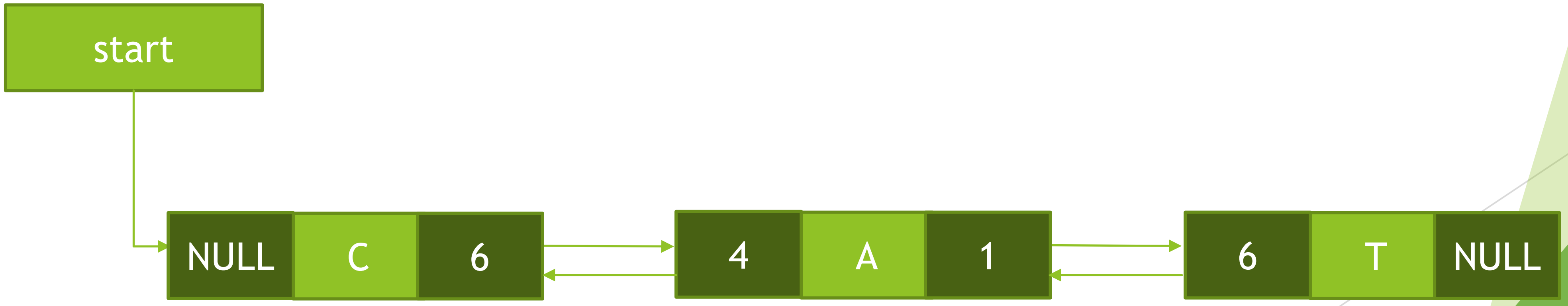
Advantage

- A doubly linked list provides the ease to manipulate the elements of the list as it maintains pointers to nodes in both the directions (forward and backward).
- The main advantage of using a doubly linked list is that it makes searching twice as efficient.

Memory Representation of Doubly Linked List

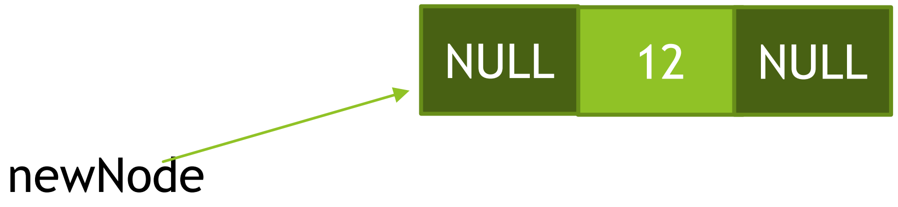
start

	Data	Prev	Next
1	T	6	-1
2			
3			
4	C	-1	6
5			
6	A	4	1
7			

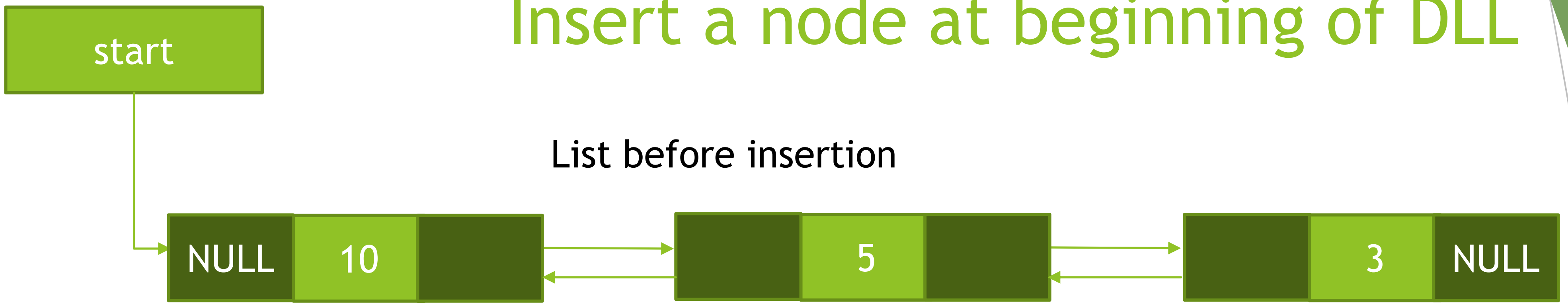


Create a New Node in Doubly Linked List

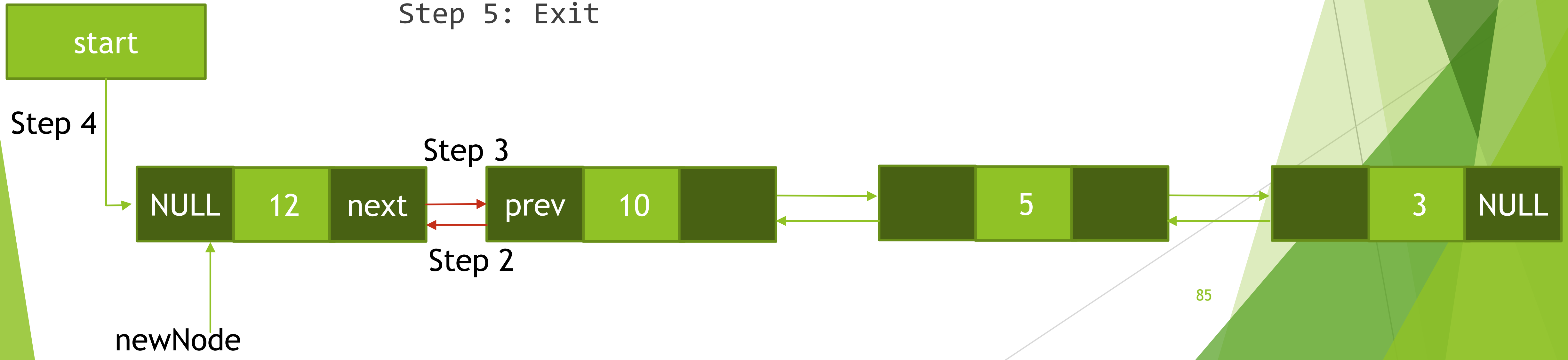
```
NODE * newNode;
newNode=(NODE *)malloc(sizeof(NODE));
printf("\nEnter data item:");
scanf("%d",&newNode ->data);
newNode ->prev=NULL
newNode ->next=NULL
```



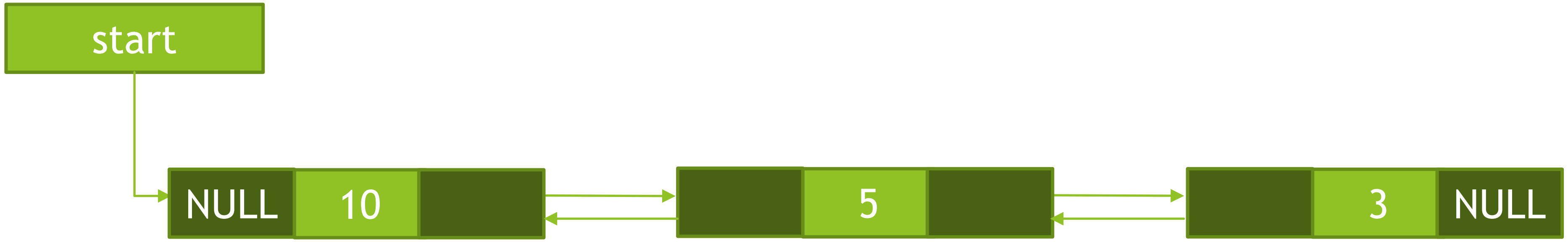
Insert a node at beginning of DLL



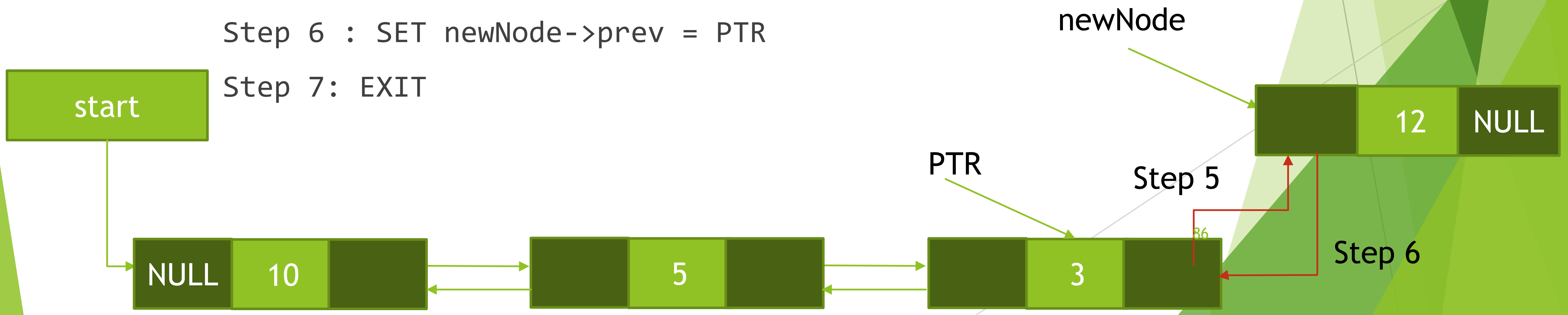

```
Step 1: create a new node
Step 2: SET START->prev = newNode
Step 3: SET newNode->next = start
Step 4: Set START = newNode
Step 5: Exit
```



Insert a node at the end of DLL




```
Step 1: create newNode
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->next != NULL
Step 4: SET PTR = PTR->next
[END OF LOOP]
Step 5: SET PTR->NEXT = newNode
Step 6 : SET newNode->prev = PTR
Step 7: EXIT
```



Insert a new node after given node

```
Step 1: create newNode
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->data != NUM
Step 4: SET PTR = PTR->next
[END OF LOOP]
Step 5: SET newNode->next = PTR->next
Step 6: SET newNode->prev = PTR
Step 7: SET PTR->next = newNode
Step 8: SET PTR->next->prev = newNode
Step 9: EXIT
```

Delete First Node of doubly linked list

```
Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 6
[END OF IF]
Step 2: SET PTR = START
Step 3: SET START = START->next
Step 4: SET START->prev = NULL
Step 5: FREE PTR
Step 6: EXIT
```

Delete last node of doubly linked list

```
Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 7
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->next != NULL
Step 4: SET PTR = PTR->next
[END OF LOOP]
Step 5: SET PTR->prev->next = NULL
Step 6: FREE PTR
Step 7: EXIT
```

Delete a node after a given node

```
Step 1: IF START = NULL
Write UNDERFLOW
Go to Step 9
[END OF IF]
Step 2: SET PTR = START
Step 3: Repeat Step 4 while PTR->data != NUM
Step 4: SET PTR = PTR->next
[END OF LOOP]
Step 5: SET TEMP = PTR->next
Step 6: SET PTR->next = TEMP->next
Step 7: SET TEMP->next->prev = PTR
Step 8: FREE TEMP
Step 9: EXIT
```

Display a List in forward direction

```
Step 1: SET PTR = START
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Display PTR->data
Step 4: SET PTR = PTR->next
[END OF LOOP]
Step 5: EXIT
```

Display a List in backward direction

```
Step 1: SET PTR = START
Step 2: Repeat Step 3 while PTR->next != NULL
Step 3: SET PTR = PTR->next
[END OF LOOP]
Step 4: Repeat Steps 5 and 6 while PTR != NULL
Step 5: Display PTR->data
Step 6: SET PTR = PTR->prev
[END OF LOOP]
Step 7: EXIT
```

Program on Doubly Linked List

Quiz

Which of the following is false about a doubly linked list?
a) We can navigate in both the directions
b) It requires more space than a singly linked list
c) The insertion and deletion of a node take a bit longer
d) Implementing a doubly linked list is easier than singly linked list

References

- Data Structures using C, Reema Thareja, Oxford
- Data Structures: A Pseudocode Approach with C, Richard F. Gilberg & Behrouz A., Forouzan, Second Edition, CENGAGE Learning