Unit 3

Prof. A.M. Joshi

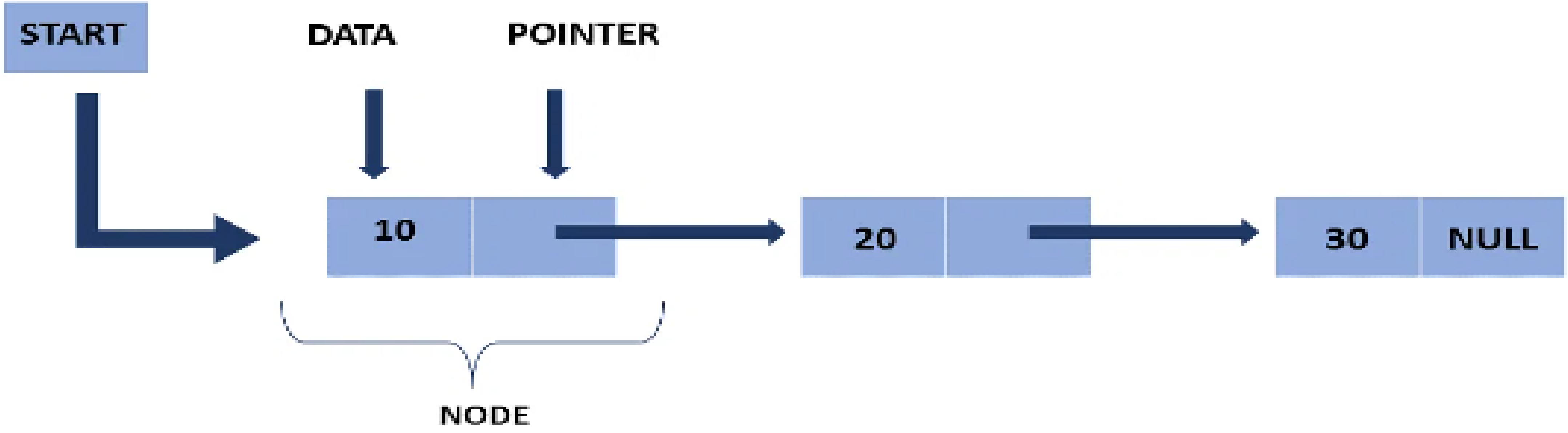
Linked List

What is a Linked List?

- A linked list is a linear data structure that stores a collection of data elements dynamically.
- Nodes represent those data elements, and links or pointers connect each node.
- Each node consists of two fields, the information stored in a linked list and a pointer that stores the address of its next node.
- The last node contains null in its second field because it will point to no node.
- A linked list can grow and shrink its size, as per the requirement.
- It does not waste memory space.

Representation of a Linked List

- This representation of a linked list depicts that each node consists of two fields.
- The first field consists of data, and the second field consists of pointers that point to another node.
- Here, the start pointer stores the address of the first node, and at the end, there is a null pointer that states the end of the Linked List.



Types of Linked Lists

- The linked list mainly has three types, they are:

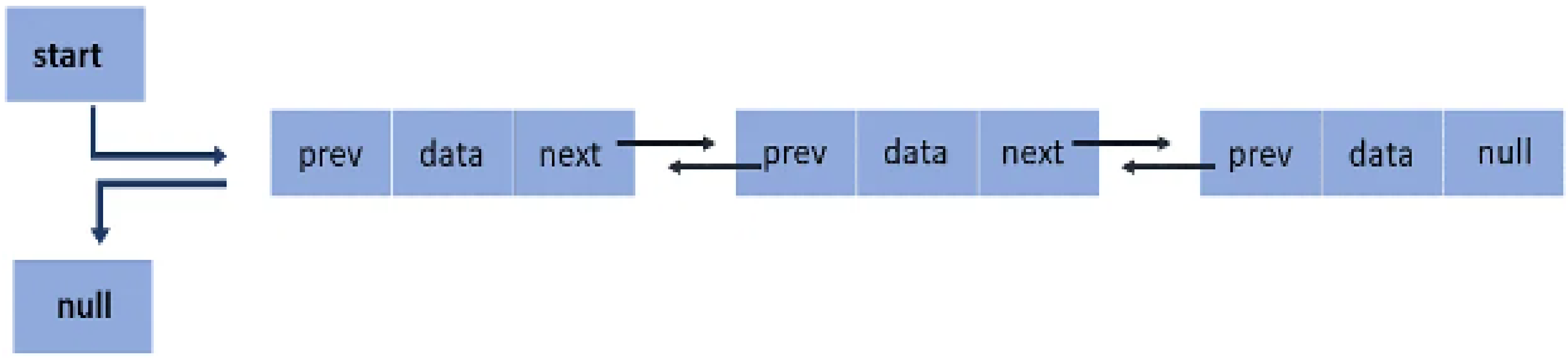
1. Singly Linked List
2. Doubly Linked List
3. Circular Linked List

Singly Linked List

- A singly linked list is the most common type of linked list.
- Each node has data and an address field that contains a reference to the next node.

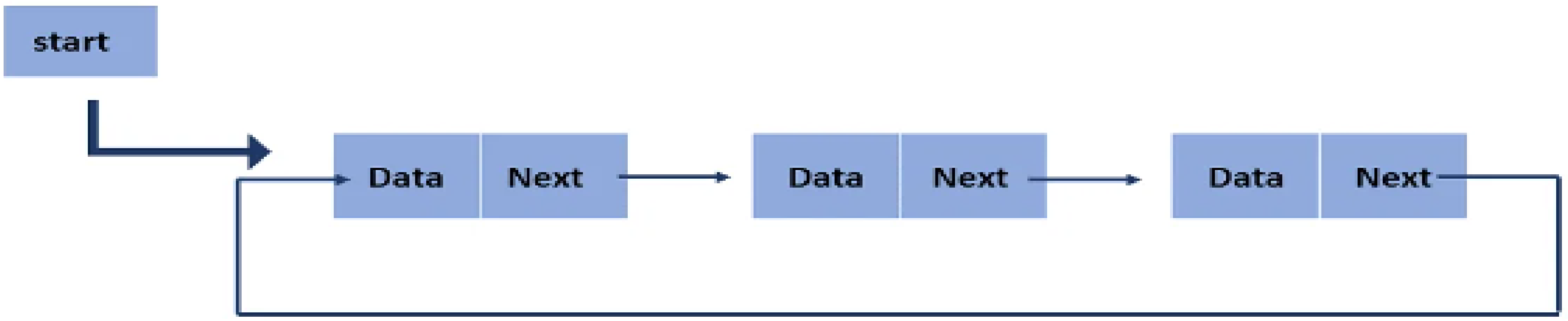
Doubly Linked List

- There are two pointer storage blocks in the doubly linked list.
- The first pointer block in each node stores the address of the previous node.
- Hence, in the doubly linked inventory, there are three fields that are the previous pointers, that contain a reference to the previous node.
- Then there is the data, and last you have the next pointer, which points to the next node.
- Thus, you can go in both directions (backward and forward).



Circular Linked List

- The circular linked list is extremely similar to the singly linked list.
- The only difference is that the last node is connected with the first node, forming a circular loop in the circular linked list.
- Circular link lists can either be singly or doubly-linked lists.
- The next node's next pointer will point to the first node to form a singly linked list.
- The previous pointer of the first node keeps the address of the last node to form a doubly-linked list.



Why linked list data structure needed?

- Here are a few advantages of a linked list that is listed below, it will help you understand why it is necessary to know.

1. Dynamic Data structure: The size of memory can be allocated or de-allocated at run time based on the operation insertion or deletion.
2. Ease of Insertion/Deletion: The insertion and deletion of elements are simpler than arrays since no elements need to be shifted after insertion and deletion, Just the address needed to be updated.
3. Efficient Memory Utilization: As we know Linked List is a dynamic data structure the size increases or decreases as per the requirement so this avoids the wastage of memory.
4. Implementation: Various advanced data structures can be implemented using a linked list like a stack, queue, graph, hash maps, etc.

Essential Operation on Linked Lists

1. Traversing: To traverse all nodes one by one.

2. Insertion: To insert new nodes at specific positions.

3. Deletion: To delete nodes from specific positions.

4. Searching: To search for an element from the linked list.

Traversal

- In this operation, you will display all the nodes in the linked list.
- When the temp is null, it means you traversed all the nodes, and you reach the end of the linked list and get out from the while loop.

```
struct node * temp = start;

printf(“\n list empty are-”);

while (temp!= NULL)

{

 printf(‘%d “, temp -> data)

 temp=temp -> next;

}
```

Insertion

- You can add a node at the beginning, middle, and end.

Insert at the Beginning

- Create a memory for a new node.
- Store data in a new node.
- Change next to the new node to point to start.
- Change starts to tell the recently created node.

```
struct node *NewNode;

NewNode=malloc(sizeof(struct node));

NewNode -> data = 40;

NewNode -> next= start;

start= NewNode;
```

Insert at the End

- Insert a new node and store data in it.
- Traverse the last node of a linked list.
- Change the next pointer of the last node to the newly created node.

```
struct node *NewNode;
NewNode=malloc(sizeof(struct node));
NewNode-> data = 40;
NewNode->next = NULL;
struct node *temp = start;
while( temp->next ! = NULL){
temp=temp -> next;
    }
 temp -> next = NewNode;
```

Insert at the Middle

- Allocate memory and store data in the new node.
- Traverse the node, which is just before the new node.
- Change the next pointer to add a new node in between.
- NewNode -> next = temp -> next;
- This line sets the next pointer of the NewNode to the same node that temp is pointing to. This effectively connects the NewNode to the node that was originally at the desired insertion position.
- .

```
struct node *NewNode;
NewNode= malloc(sizeof(struct node));
NewNode -> data = 40;
struct node - > temp = start;
for(int i=2; i<position; i++)
{
if (temp -> next!= NULL)
temp = temp -> next;
}
NewNode -> next = temp -> next;
temp -> next = NewNode;
```

Deletion

- You can also do deletion in the linked list in three ways either from the end, beginning, or from a specific position.

1. Delete from the Beginning
2. Delete from the End
3. Delete from the Middle

Delete from the Beginning

- The point starts at the second node.

```
start = start -> next;
```

Delete from the End

- Traverse the second last element in the linked list.
- Change its next pointer to null.

```
struct node * temp = start;

while(temp -> next -> next!= NULL)
{

temp=temp -> next;

}

temp -> next = NULL;
```

Delete from the Middle

- Traverse the element before the element to be deleted.
- Change the next pointer to exclude the node from the linked list.

```
for (int i = 2; i<position; i++){

if (temp -> next ! = NULL)

temp = temp -> next;
}
temp->next = temp->next->next;
}
```

Searching

- The search operation is done to find a particular element in the linked list.

- If the element is found in any location, then it returns.

- Else, it will return null.

Cont...

```
int searchNode(struct node *head,int key)
{
    struct node *temp = head;

    //iterate the entire linked list and print the
data
    while(temp != NULL)
    {
        //key found return 1.
        if(temp->data == key)
            return 1;
```

```
//key found return 1.
        if(temp->data == key)
            return 1;
temp = temp->next;
    }
    //key not found
    return -1;
```