

B. Tech. I CSE (Sem-2)  
Data Structures  
CS102

# Linked List

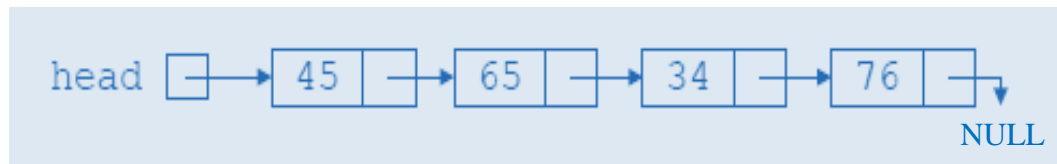
\*Some slides are kept with logical or syntax error. So, if you are absent in theory class, please also refer the slides provided at the end of the presentation.

Dr. Rupa G. Mehta  
Dr. Dipti P. Rana  
Department of Computer Science and Engineering  
SVNIT, Surat

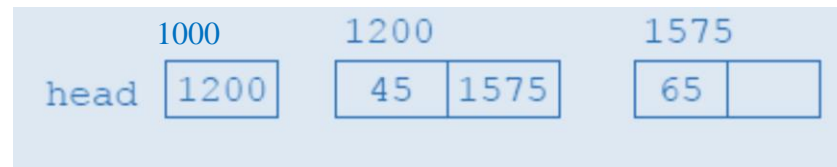
# To Learn...

- Linked list
- Array versus linked list
- Linked lists in C
- Types of linked lists
  - Single linked list
  - Circular linked list
  - Doubly linked list

# Linked List



Example of Linked list

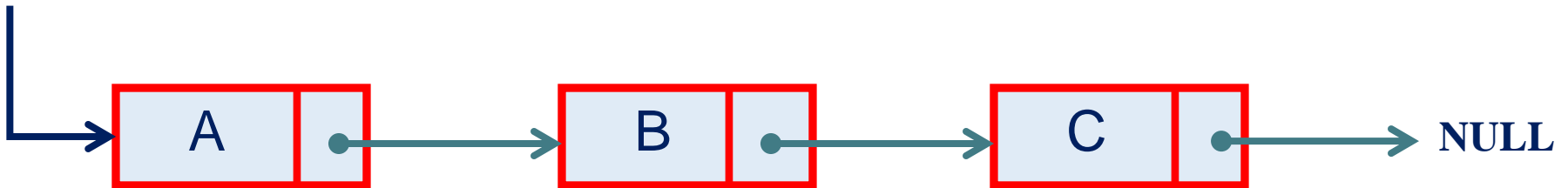


Example Linked list and Values of the Links

# Linked List

- A linked list is a **data structure** which allows to store data dynamically and manage data efficiently
- Linked list representation

header

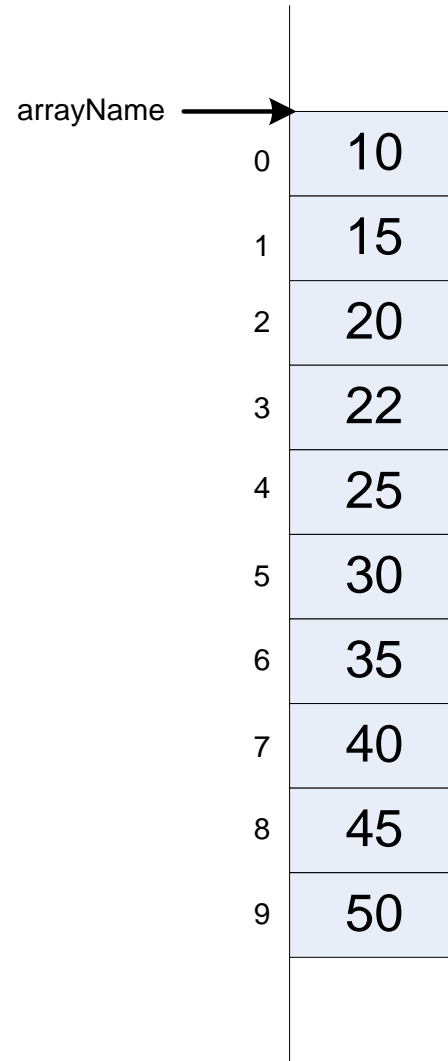


# Linked List: Features

- There is a pointer (called **header**) points the first element (also called node)
- Successive nodes are connected by **pointers**
- Last element points to **NULL**
- It can grow or shrink in size during execution of a program
- It can be made just as long as required
- It does not waste memory space, consume exactly what it needs

# Arrays versus Linked Lists

# Array: Contagious Storage



arrayName →	
0	10
1	15
2	20
3	22
4	25
5	30
6	35
7	40
8	45
9	50

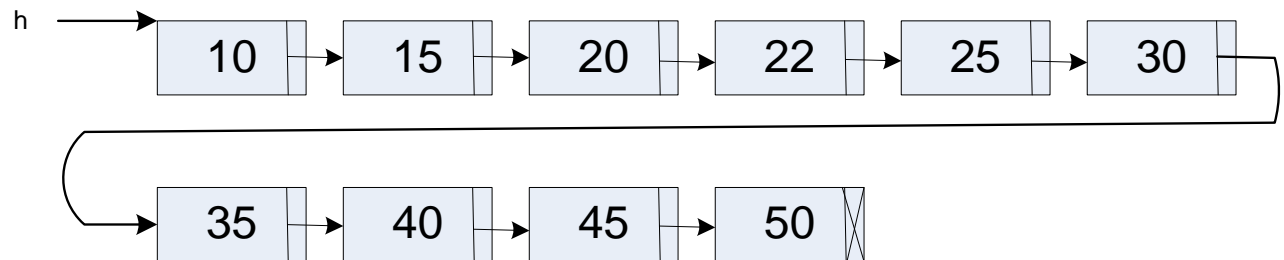
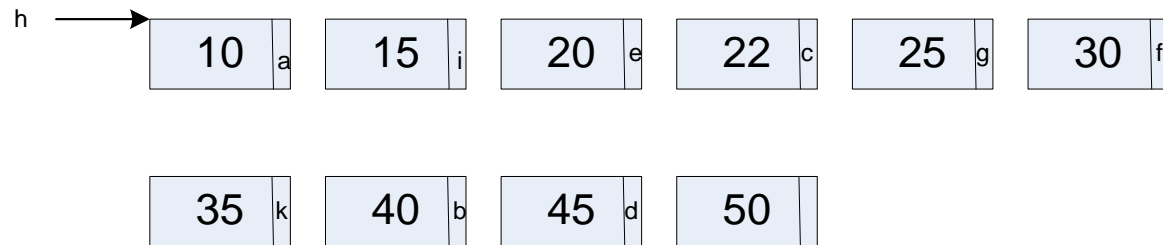
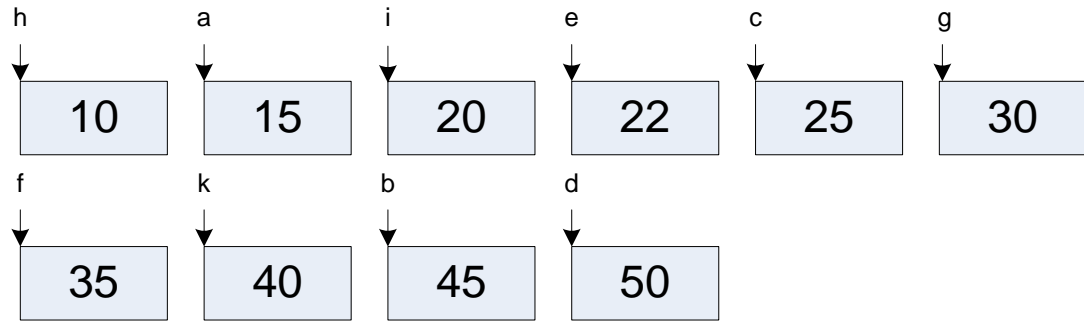
# Array versus Linked Lists

- **In arrays**
  - Elements are stored in a contiguous memory locations
- Arrays are static data structure unless we use dynamic memory allocation
- Arrays are suitable for
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.



# Linked List: Non-Contiguous Storage

a	15
b	45
c	25
d	50
e	22
f	35
g	30
h	10
i	20
j	
k	40



# Array versus Linked Lists

- **In Linked lists**

- adjacency between any two elements are maintained by means of links or pointers
- It is essentially a dynamic data structure
- Linked lists are suitable for
  - Inserting an element at any position.
  - Deleting an element from any where.
  - Applications where sequential access is required.
  - In situations, where the number of elements cannot be predicted beforehand.

# Linked Lists in C

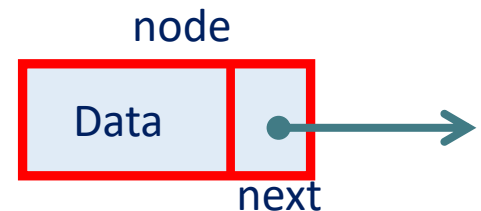
# Defining a Node of a Linked List

Each structure of the list is called a **node**, and consists of two fields:

- Item (or) data
- Address of the next item in the list (or) pointer to the next node in the list

**How to define a node of a linked list?**

```
struct node
{
    int data;           /* Data */
    struct node *next; /* pointer*/
} ;
```



## **Note:**

Such structures which contain a member field pointing to the same structure type are called **self-referential structures**.

# Types of Lists

Single Linked List

Doubly Linked List

Circular Linked List

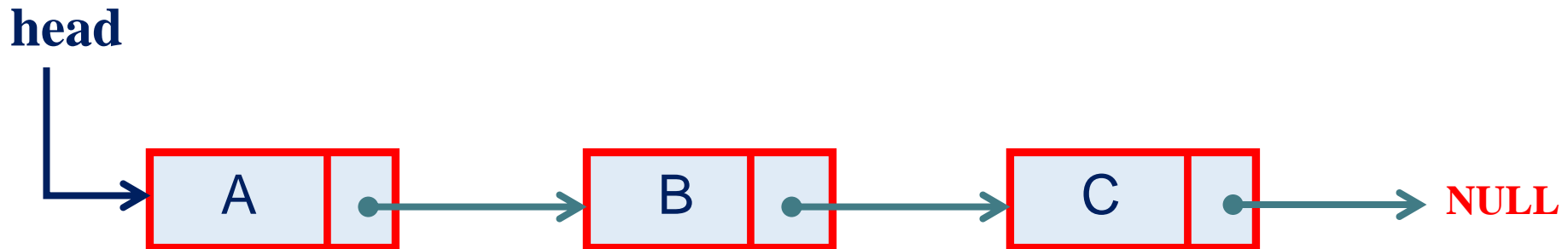
# Single Linked List

# Types of Lists: Single Linked List

Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

## Single linked list (or simply linked list)

- A head pointer addresses the first element of the list.
- Each element points at a successor element.
- The last element has a link value NULL.



# Example 1: Creating a Single Linked List

Linked list to store and print roll number, name and age of 3 students.

```
#include <stdio.h>
struct stud
{
    int roll;
    char name[30];
    int age;
    struct stud *next;
};
main()
{
    struct stud n1, n2, n3;
    struct stud *p;
    scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
    scanf ("%d %s %d", &n2.roll, n2.name, &n2.age);
    scanf ("%d %s %d", &n3.roll, n3.name, &n3.age);
```



# Example 1: Creating a Single Linked List

```
n1.next = &n2 ;
n2.next = &n3 ;
n3.next = NULL ;
/* Now traverse the list and print the elements */
p = &n1 ;          /* point to 1st element */
while (p != NULL)
{
    printf ("\n %d %s %d", p->roll, p->name, p->age);
    p = p->next;
}
}
```

# Example 1: Illustration

**The structure:**

```
struct stud
{
    int roll;
    char name[30];
    int age;
    struct stud *next;
};
```

**Also assume the list with three nodes n1, n2 and n3 for 3 students.**

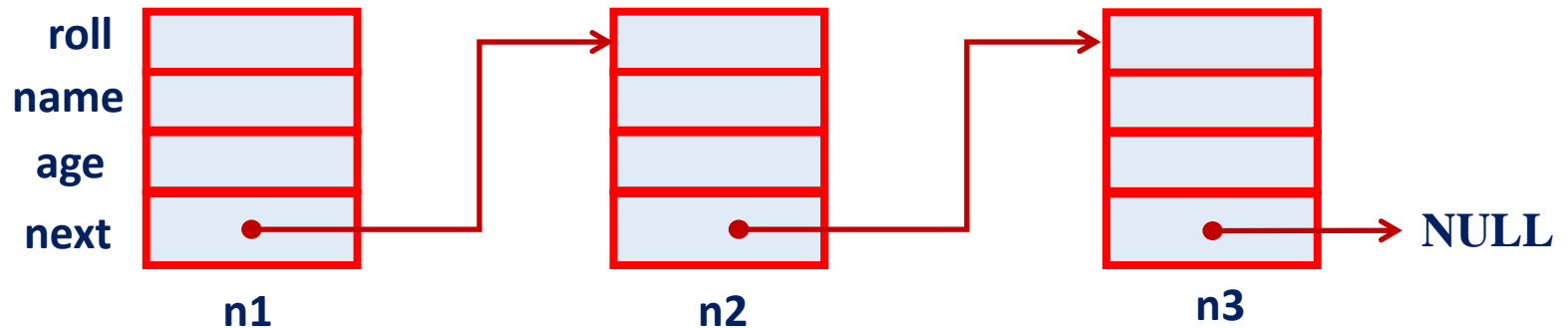
```
struct stud n1, n2, n3;
```

# Example 1: Illustration

To create the links between nodes, it is written as:

```
n1.next = &n2 ;  
n2.next = &n3 ;  
n3.next = NULL ;    /* No more nodes follow */
```

- Now the list looks like:



# Example 2: Creating a Single Linked List

C-program to store 10 values on a linked list reading the data from keyboard.

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

struct node {
    int data;                //Data part
    struct node *next;      //Address part
}*header;

void createList(int n);      /* Functions to create a list*/

int main()
{
    int n;
    printf("Enter the total number of nodes: ");
    scanf("%d", &n);
    createList(n);
    return 0;
}
```

# Example 2: Creating a Single Linked List

```
void createList(int n)
{
    struct node *newNode, *temp;
    int data, i;

    /* A node is created by allocating memory to a structure */
    newNode = (struct node *)malloc(sizeof(struct node));

    /* If unable to allocate memory for head node */
    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        printf("Enter the data of node 1: ");
        scanf("%d", &data);

        newNode->data = data; //Links the data field with data
        newNode->next = NULL; //Links the address field to NULL
        header = newNode;    //Header points to the first node
        temp = newNode;      //First node is the current node
    }
}
```

# Example 2: Creating a Single Linked List

```
for(i=2; i<= n; i++)
{
    /* A newNode is created by allocating memory */
    newNode = (struct node *)malloc(sizeof(struct node));

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
        break;
    }
    else
    {
        printf("Enter the data of node %d: ", i);
        scanf("%d", &data);

        newNode->data = data; //Links the data field of newNode with data
        newNode->next = NULL; //Links the address field of newNode with NULL

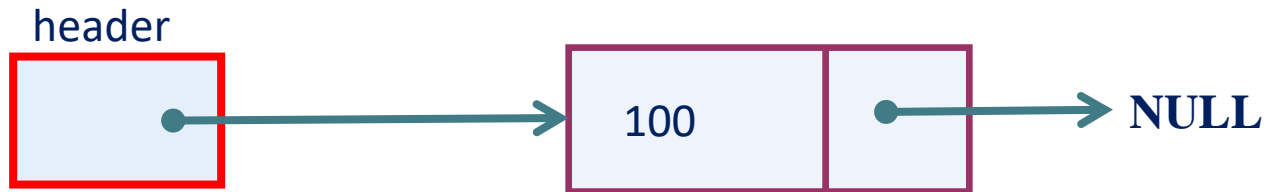
        temp->next = newNode; //Links previous node i.e. temp to the newNode
        temp = temp->next;
    }
}
}
```

# Example 2: Illustration

- To start with, we have to create a **node** (the first node), and make **header** point to it.

```
newNode = (struct node *)malloc(sizeof(struct node));  
newNode->data = data;    //Links the data field with data  
newNode->next = NULL;    //Links the address field to NULL  
header = newNode;  
temp = newNode;
```

It creates a single node. For example, if the data entered is 100 then the list look like



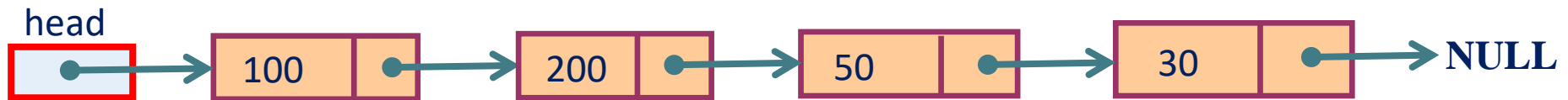
# Creating a Single Linked List

If we need **n number of nodes** in the linked list:

- Allocate n newNodes, one by one.
- Read in the data for the newNodes.
- Modify the links of the newNodes so that the chain is formed.

```
newNode = (struct node *)malloc(sizeof(struct node));  
newNode->data = data;           //Links the data field of newNode with data  
newNode->next = NULL;           //Links the address field of newNode with NULL  
  
temp->next = newNode;           //Links previous node i.e. temp to the newNode  
temp = temp->next;
```

It creates **n** number of nodes . For e.g. if the data entered is 200, 50, 30 then the list look like





# Example 3: Creating a Single Linked List

C-program to copy an array to a single linked list.

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

struct node {
    int data;                //Data part
    struct node *next;       //Address part
};

int main()
{
    struct node *header, *newNode, *temp;
    int data, i, n, a[100];
    printf("Enter the total number of data: ");
    scanf("%d", &n);
    // Write code here to initialize the array a with n elements //
```

...

# Example 2: Creating a Single Linked List

```
/* A node is created by allocating memory to a structure */
newNode = (struct node *)malloc(sizeof(struct node));

/* If unable to allocate memory for head node */
if(newNode == NULL)
{
    printf("Unable to allocate memory.");
}
else
{
    newNode->data = a[0]; //Links the data field with data
    newNode->next = NULL; //Links the address field to NULL
    header = newNode;    //Header points to the first node
    temp = header;
```

# Example 2: Creating a Single Linked List

```
for(i = 1; i <= n; i++)
{
    /* A newNode is created by allocating memory */
    newNode = (struct node *)malloc(sizeof(struct node));

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
        break;
    }
    else
    {
        newNode->data = a[i]; //Links the data field of newNode with a[i]
        newNode->next = NULL; //Links the address field of newNode with NULL

        temp->next = newNode; //Links previous node i.e. temp to the newNode
        temp = temp->next;
    }
}
}
```

# Operations on Linked Lists

# Operations on Single Linked List

- Traversing a list
  - Printing, finding minimum, etc.
- Insertion of a node into a list
  - At front, end
  - Anywhere with given index, order etc.
- Deletion of a node from a list
  - At front, end
  - Anywhere with given index, value, etc.
- Comparing two linked lists
  - Similarity, intersection, etc.
- Merging two linked lists into a larger list
  - Union, concatenation, etc.
- Ordering a list
  - Reversing, sorting, etc.

# Traversing a Linked List

# Single Linked List: Traversing

Once the linked list has been constructed and **header** points to the first node of the list,

- Follow the pointers.
- Display the contents of the nodes as they are traversed.
- Stop when the **next** pointer points to **NULL**.

The function **traverseList**(struct Node \*) is given in the next slide. This function to be called from **main()** function as:

```
int main()
{
    // Assume header, the pointer to the linked list is given as an input
    printf("\n Data in the list \n");
    traverseList(header);
    return 0;
}
```

# Single Linked List: Traversing

```
void traverseList(struct Node *header)
{
    struct node *temp;

    /* If the list is empty i.e. head = NULL */
    if(header == NULL)
    {
        printf("List is empty.");
    }
    else
    {
        temp = header;
        while(temp != NULL)
        {
            printf("Data = %d\n", temp->data); //Prints the data of current node
            temp = temp->next;                //Advances the position of current node
        }
    }
}
```



# Insertion in a Linked List

# Single Linked List: Insertion

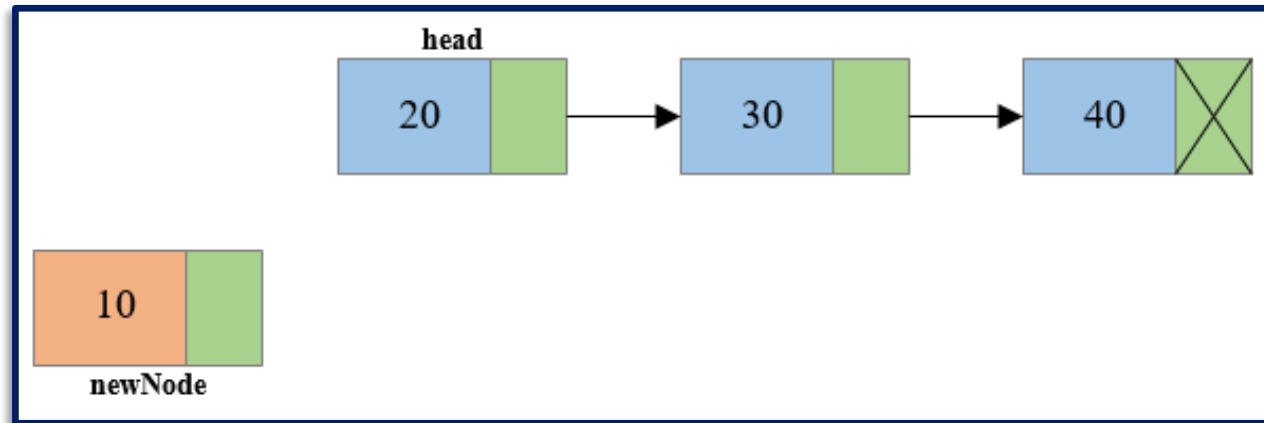
## Insertion steps:

- Create a new node
- Start from the header node
- Manage links to
  - Insert at front
  - Insert at end
  - Insert at any position

# Insertion at Front

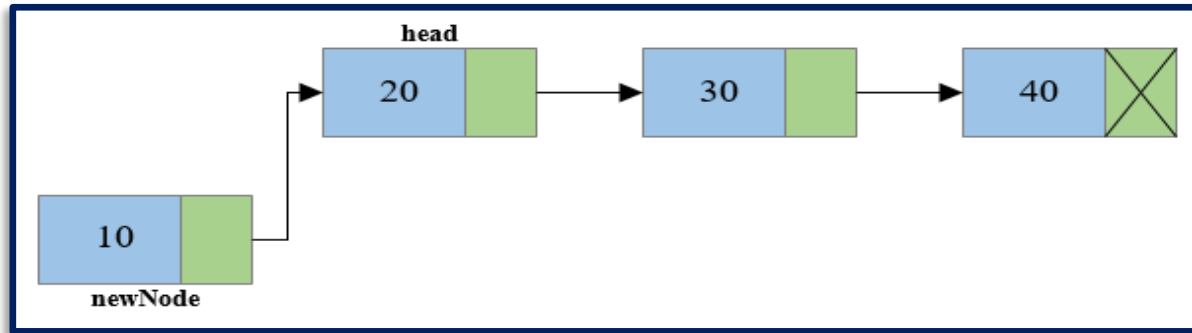
**Steps to insert node at the beginning of singly linked list**

**Step 1:** Create a new node.

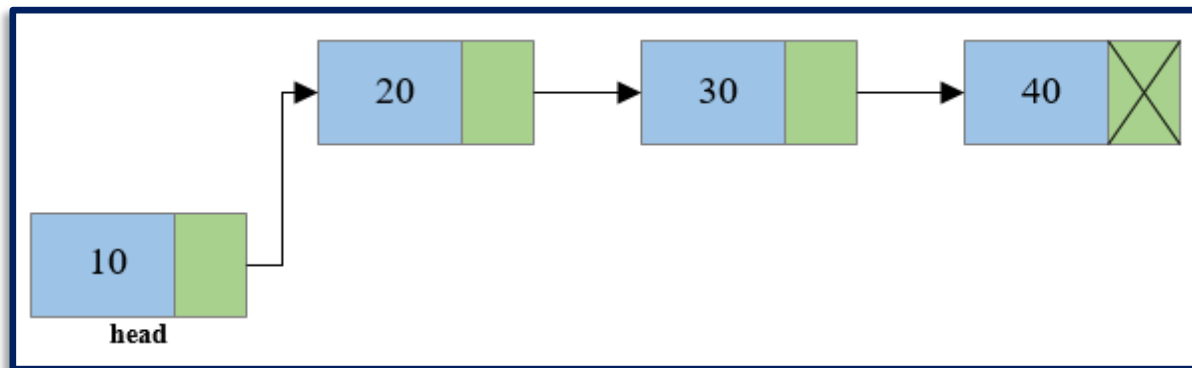


# Insertion at Front

**Step 2:** Link the newly created node with the head node, i.e. the **newNode** will now point to **head** node.



**Step 3:** Make the new node as the head node, i.e. now **head** node will point to **newNode**.



# Insertion at Front

```
/*Create a new node and insert at the beginning of the linked list.*/

void insertNodeAtBeginning(int data)
{
    struct node *newNode;
    newNode = (struct node*)malloc(sizeof(struct node));

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data;    //Links the data part
        newNode->next = head;    //Links the address part

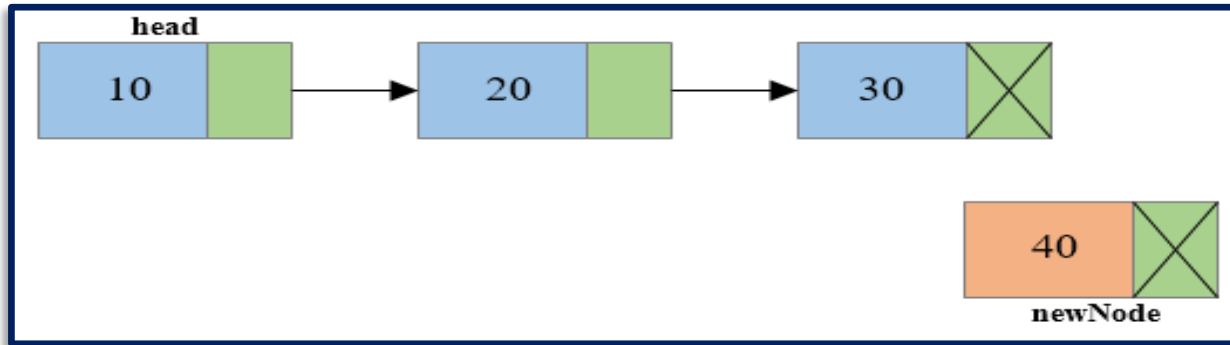
        head = newNode;         //Makes newNode as first node

        printf("DATA INSERTED SUCCESSFULLY\n");
    }
}
```

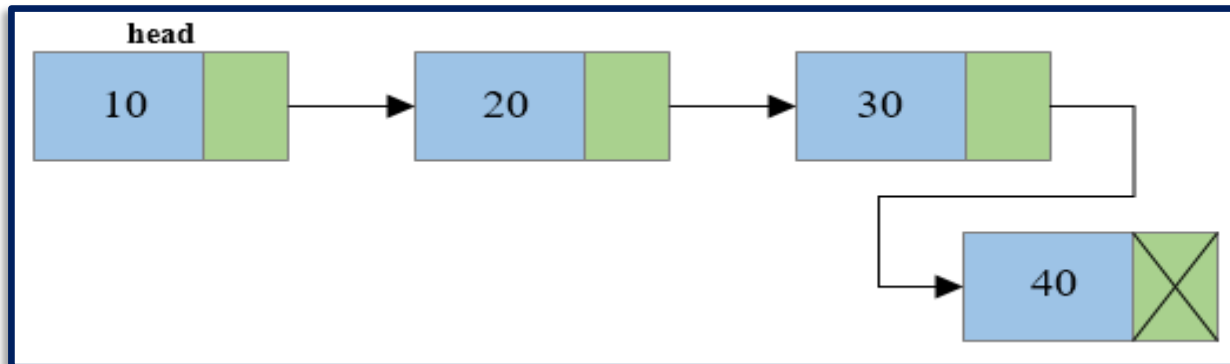
# Single Linked List: Insertion at End

## Steps to insert node at the end of Singly linked list

**Step 1:** Create a new node and make sure that the address part of the new node points to NULL. i.e. `newNode->next=NULL`



**Step 2:** Traverse to the last node of the linked list and connect the last node of the list with the new node, i.e. last node will now point to new node. (`lastNode->next = newNode`).



# Insertion at End

```
/* Create a new node and insert at the end of the linked list. */
void insertNodeAtEnd(int data)
{
    struct node *newNode, *temp;
    newNode = (struct node*)malloc(sizeof(struct node));
    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data; //Links the data part
        newNode->next = NULL;
        temp = head;

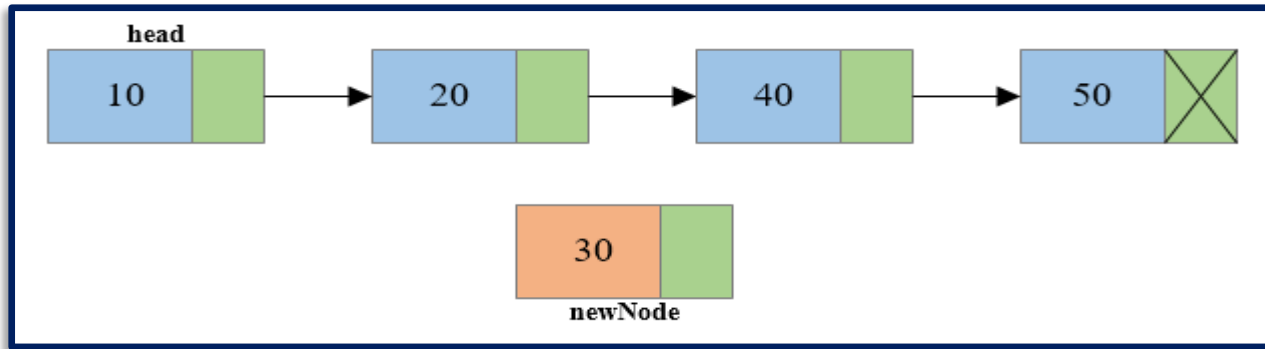
        while(temp->next != NULL) //Traverse to the last node
            temp = temp->next;

        temp->next = newNode; //Links the address part
        printf("DATA INSERTED SUCCESSFULLY\n");
    }
}
```

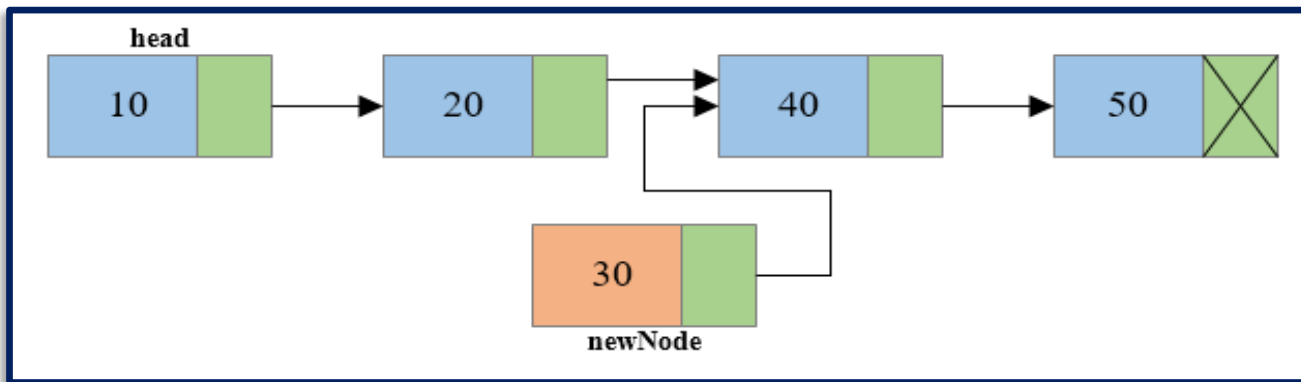
# Single Linked List: Insertion at any Position

## Steps to insert node at any position of Singly Linked List

**Step 1:** Create a new node.



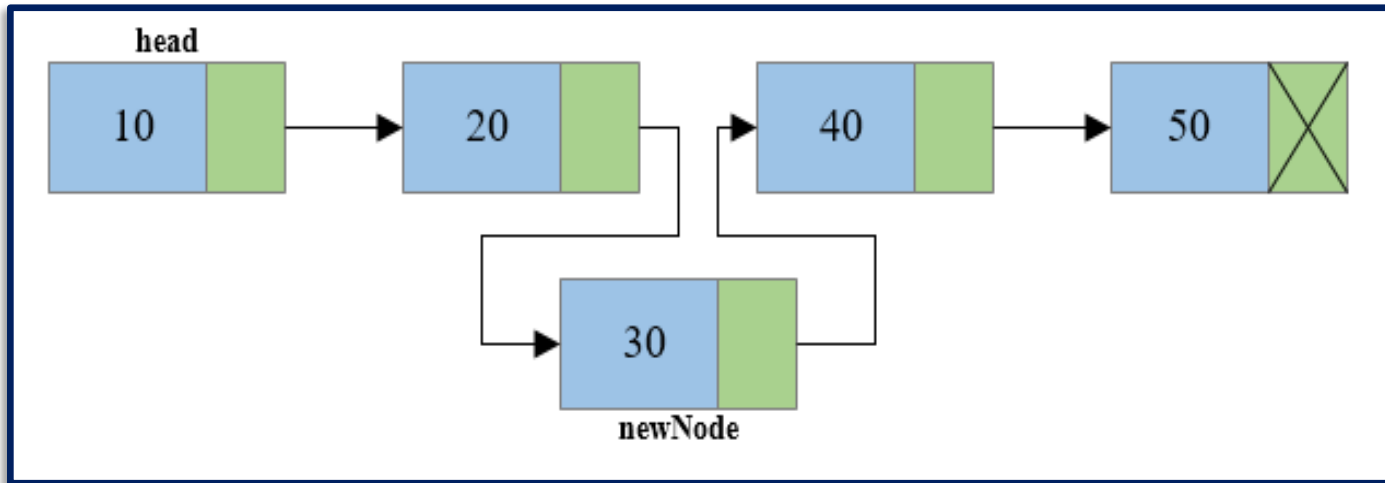
**Step 2:** Traverse to the  $n-1^{\text{th}}$  position of the linked list and connect the new node with the  $n+1^{\text{th}}$  node. ( $\text{newNode} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$ ) where temp is the  $n-1^{\text{th}}$  node.





# Single Linked List: Insertion at any position

**Step 3:** Now at last connect the  $n-1^{\text{th}}$  node with the new node i.e. the  $n-1^{\text{th}}$  node will now point to new node. (`temp->next = newNode`) where temp is the  $n-1^{\text{th}}$  node.



# Insertion at any Position

```
/* Create a new node and insert at middle of the linked list.*/  
  
void insertNodeAtMiddle(int data, int position)  
{  
    int i;  
    struct node *newNode, *temp;  
  
    newNode = (struct node*)malloc(sizeof(struct node));  
  
    if(newNode == NULL)  
    {  
        printf("Unable to allocate memory.");  
    }  
    else  
    {  
        newNode->data = data;    //Links the data part  
        newNode->next = NULL;  
  
        temp = head;
```

# Insertion at any Position

```
for(i=2; i<=position-1; i++) /* Traverse to the n-1 position */
{
    temp = temp->next;

    if(temp == NULL)
        break;
}
if(temp != NULL)
{
    /* Links the address part of new node */
    newNode->next = temp->next;

    /* Links the address part of n-1 node */
    temp->next = newNode;

    printf("DATA INSERTED SUCCESSFULLY\n");
}
else
{
    printf("UNABLE TO INSERT DATA AT THE GIVEN POSITION\n");
}
}
```

# Deletion from a Linked List

# Single Linked List: Deletion

## Deletion steps

- Start from the header node
- Manage links to
  - Delete at front
  - Delete at end
  - Delete at any position
- freeingup the node as free space.

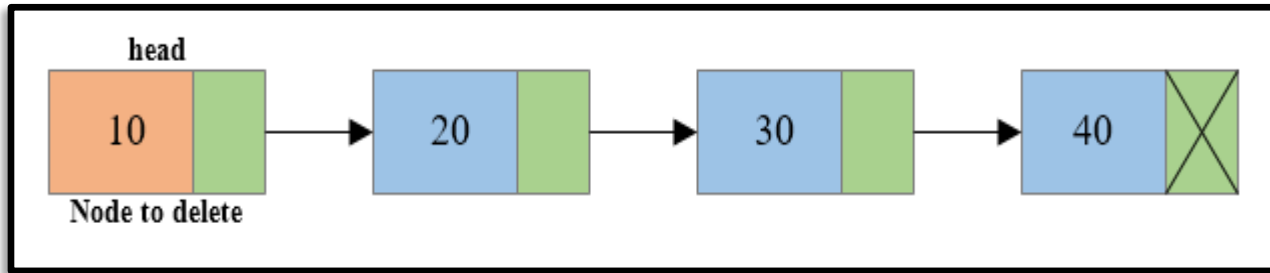
# Free Memory after Deletion

- Do not forget to **free()** memory location dynamically allocated for a node **after deletion** of that node.
- It is the programmer's responsibility to free that memory block.
- Failure to do so may create a **dangling pointer** – a memory, that is not used either by the programmer or by the system.
- The content of a free memory is not erased until it is overwritten.

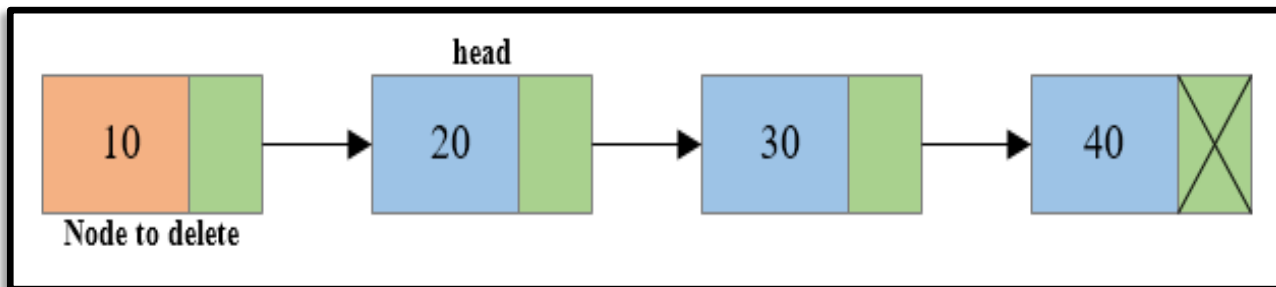
# Single Linked List: Deletion at Front

## Steps to delete first node of Singly Linked List

**Step 1:** Copy the address of first node i.e. **head** node to some temp variable say **toDelete**.

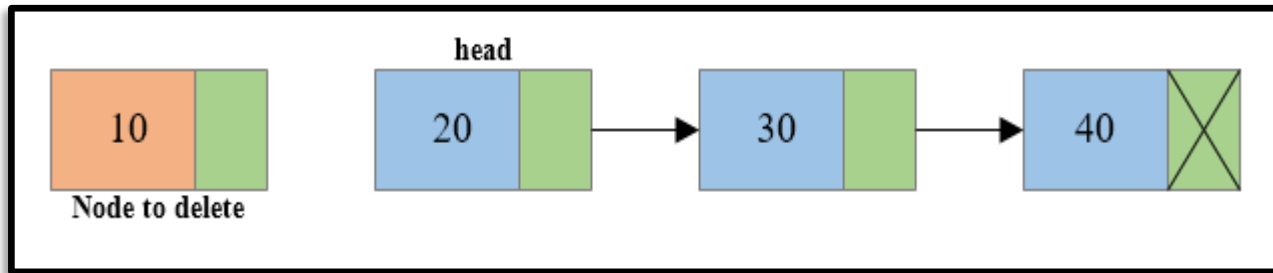


**Step 2:** Move the head to the second node of the linked list ( $\text{head} = \text{head} \rightarrow \text{next}$ ).

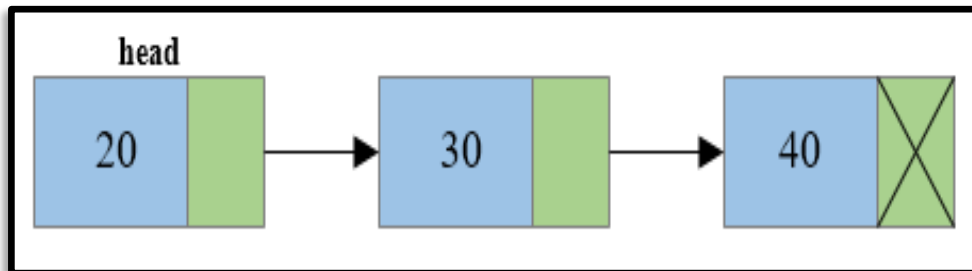


# Single Linked List: Deletion at front

**Step 3:** Disconnect the connection of first node to second node.



**Step 4:** Free the memory occupied by the first node.





# Deletion at Front

```
/* Delete the first node of the linked list */
void deleteFirstNode()
{
    struct node *toDelete;

    if(head == NULL)
    {
        printf("List is already empty.");
    }
    else
    {
        toDelete = head;
        head = head->next;

        printf("\nData deleted = %d\n", toDelete->data);

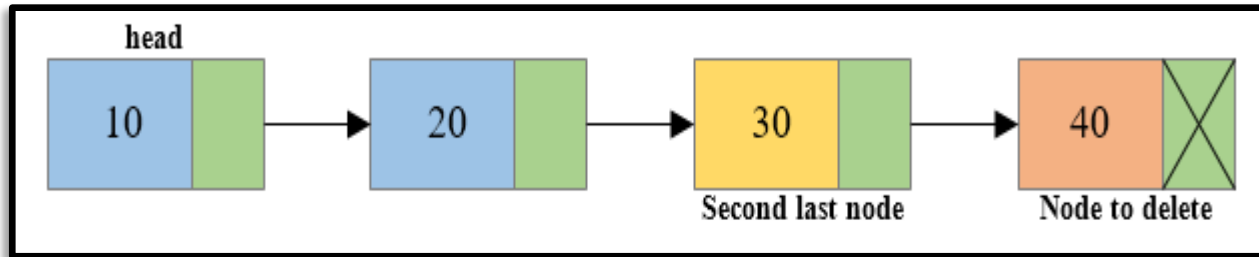
        /* Clears the memory occupied by first node*/
        free(toDelete);

        printf("SUCCESSFULLY DELETED FIRST NODE FROM LIST\n");
    }
}
```

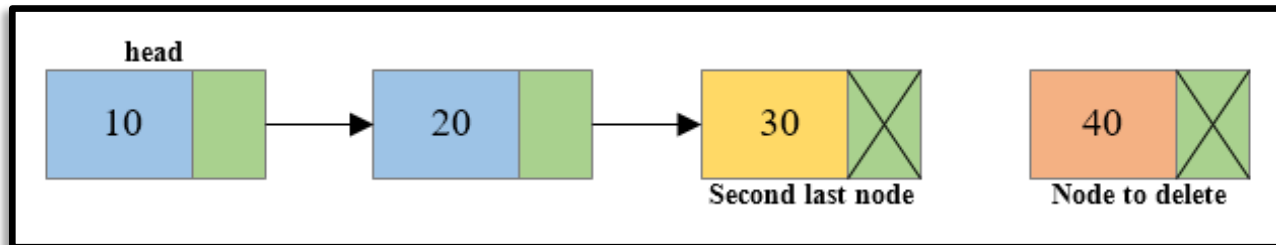
# Single Linked List: Deletion at End

## Steps to delete last node of a Singly Linked List

**Step 1:** Traverse to the last node of the linked list keeping track of the second last node in some temp variable say **secondLastNode**.

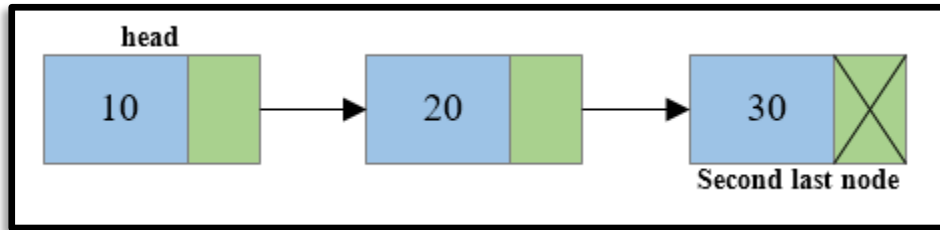


**Step 2:** If the last node is the head node then make the head node as NULL else disconnect the second last node with the last node i.e. `secondLastNode->next = NULL`



# Single linked list: Deletion at End

**Step 3:** Free the memory occupied by the last node.



# Deletion at End

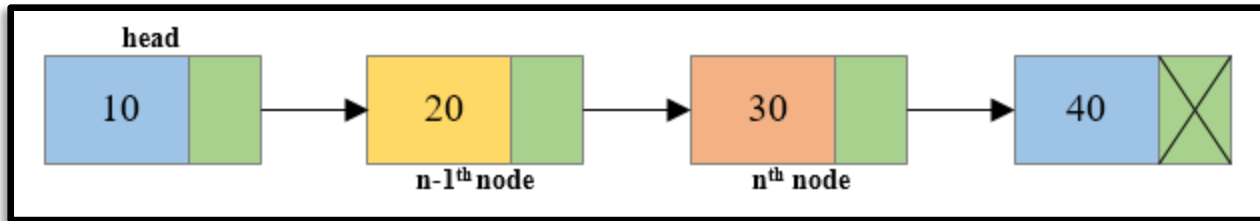
```
/* Delete the last node of the linked list */
void deleteLastNode()
{
    struct node *toDelete, *secondLastNode;
    toDelete = head;
    secondLastNode = head;

    while(toDelete->next != NULL) /* Traverse to the last node of the list*/
    {
        secondLastNode = toDelete;
        toDelete = toDelete->next;
    }
    if(toDelete == head)
    {
        head = NULL;
    }
    else
    {
        /* Disconnects the link of second last node with last node */
        secondLastNode->next = NULL;
    }
    /* Delete the last node */
    free(toDelete);
}
```

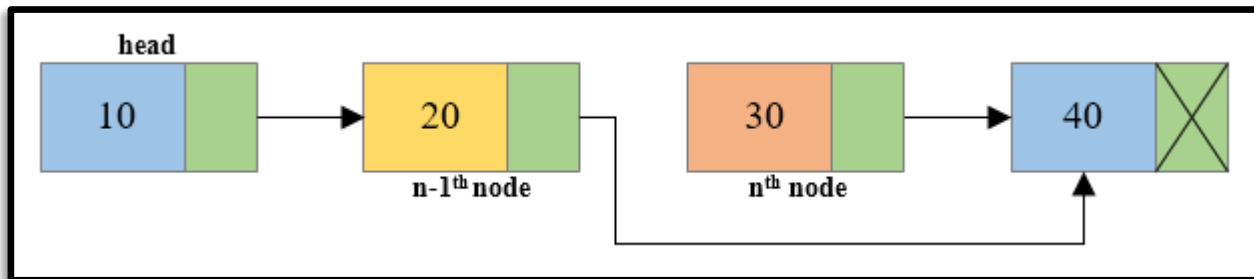
# Single Linked List: Deletion at any Position

## Steps to delete a node at any position of Singly Linked List

**Step 1:** Traverse to the  $n^{\text{th}}$  node of the singly linked list and also keep reference of  $n-1^{\text{th}}$  node in some temp variable say **prevNode**.

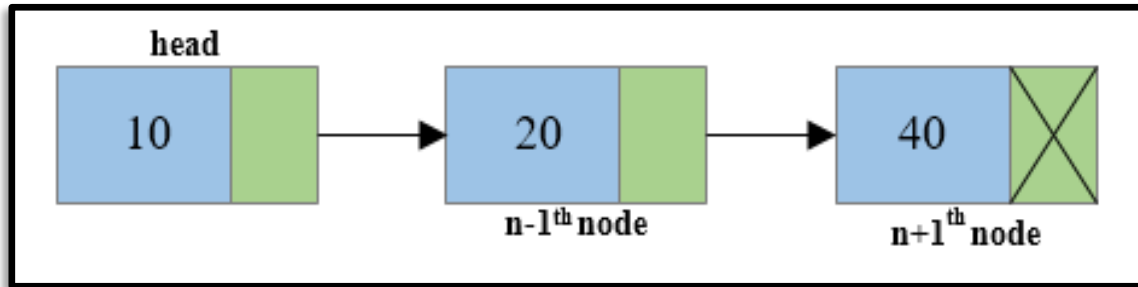


**Step 2:** Reconnect  $n-1^{\text{th}}$  node with the  $n+1^{\text{th}}$  node i.e.  $\text{prevNode} \rightarrow \text{next} = \text{toDelete} \rightarrow \text{next}$  (Where **prevNode** is  $n-1^{\text{th}}$  node and **toDelete** node is the  $n^{\text{th}}$  node and  $\text{toDelete} \rightarrow \text{next}$  is the  $n+1^{\text{th}}$  node).



# Single Linked List: Deletion at any Position

**Step 3:** Free the memory occupied by the  $n^{\text{th}}$  node i.e. **toDelete** node.



# Deletion at any Position

```
/* Delete the node at any given position of the linked list */
void deleteMiddleNode(int position)
{
    int i;
    struct node *toDelete, *prevNode;
    if(head == NULL)
    {
        printf("List is already empty.");
    }
    else
    {
        toDelete = head;
        prevNode = head;

        for(i=2; i<=position; i++)
        {
            prevNode = toDelete;
            toDelete = toDelete->next;

            if(toDelete == NULL)
                break;
        }
    }
}
```

# Deletion at any Position

```
if(toDelete != NULL)
{
    if(toDelete == head)
        head = head->next;

    prevNode->next = toDelete->next;
    toDelete->next = NULL;

    /* Deletes the n node */
    free(toDelete);

    printf("SUCCESSFULLY DELETED NODE FROM MIDDLE OF LIST\n");
}
else
{
    printf("Invalid position unable to delete.");
}
}
```



# Cross Check 1

- How to delete the node when you do not have the head pointer?
  - Given Information
    1. A singly linked list
    2. One Pointer which is pointing to the node which is required to be deleted
  - Not Given
    - Any information about the head pointer or any other node
  - Requirement
    - You need to check and write a function to delete that node from the linked list
    - Function will take only one argument, i.e., a pointer to the node which is to be deleted
  - Assumption
    1. No head reference is given to you
    2. Guaranteed that the node to be deleted is not the last node

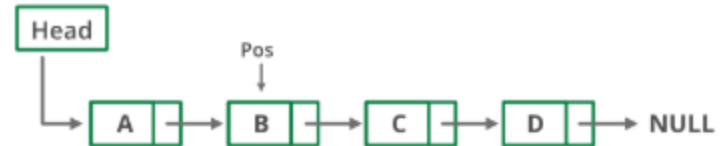
# Cross Check 1: Solution

- Conventional deletion method would fail here
  - It would be a simple deletion problem from the singly linked list if the head pointer was given
  - Because for deletion you must know the previous node and you can easily reach there by traversing from the head pointer
  - Conventional deletion is impossible without knowledge of the previous node of a node that needs to be deleted.

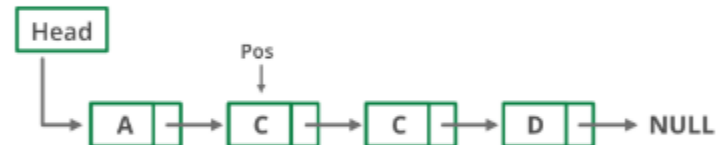
# Cross Check 1: Solution

- Then how to delete the node when you do not have the head pointer
  1. Copy the data of the next node to the data field of the current node to be deleted
  2. Now move one step forward
    - So, Next node has become the current node and the current has become the previous node
  3. Now we can easily delete the current node by conventional deletion methods.

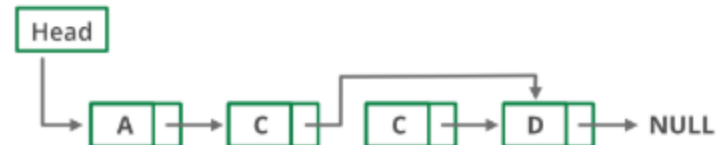
Initially :



Step 1:



Step 2:



**Conclusion**  
**ONLY POSSIBLE,**  
**if the node to be deleted is not the last node**

Operation	Operation Type	Description	Time Complexity	Space Complexity
Insertion	At Beginning	Insert a new node at the start of a linked list.	O (1)	O (1)
	At the End	Insert a new node at the end of the linked list.	O (N)	O (1)
	At Specific Position	Insert a new node at a specific position in a linked list.	O (N)	O (1)
Deletion	From Beginning	Delete a node from the start of a linked list	O (1)	O (1)
	From the End	Delete a node at the end of a linked list.	O (N)	O (1)
	A Specific Node	Delete a node from a specific position of a linked list.	O (N)	O (1)
Traversal		Traverse the linked list from start to end.	O (N)	O (1)

# Comparing Two Linked Lists

# Single Linked List: Comparing two Lists

## Comparing two linked list includes

- Identifying whether the given two linked list are **identical**.
- Two Linked Lists are identical when they have **same data and arrangement** of data is also same.
- Checking whether the lists have **same values** when the **arrangement** is not same.

# Comparing two Linked Lists

```
/* Return true if linked lists a and b are identical, otherwise false */
bool areIdentical(struct node *a, struct node *b)
{
    while (a != NULL && b != NULL)
    {
        if (a->data != b->data)
            return false;
        /* If we reach here, then a and b are not NULL and their data is same, so
           move to next nodes in both lists */
        a = a->next;
        b = b->next;
    }
    //If linked lists are identical, then 'a' and 'b' must be NULL at this point.
    return (a == NULL && b == NULL);
}

int main()
{
    struct node *a, *b;
    a = createList(5); // e.g: a: 5->4->3->2->1
    b = createList(5); // e.g: b: 5->4->3->2->1
    areIdentical(a, b)? printf("Identical"): printf("Not identical");
    return 0;
}
```

# Self Check

## Write a function to:

- Concatenate or merge two given list into one big list.  
`node *concatenate(node *a, node *b) ;`
- Compare two given list with same data but different arrangement.  
e.g:    a: 5->4->3->2->1  
          b: 1->2->3->4->5
- Count the number of nodes in the given list using iterative method and recursive method.



# Ordering Linked List

# Single Linked List: Reversing

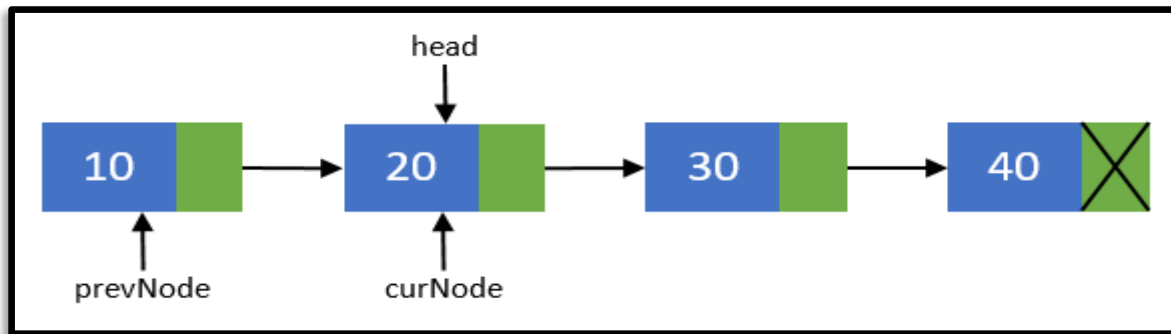
Reversing a list can be performed in two ways:

- **Iterative** method
- **Recursive** method

## Steps to reverse a Singly Linked List using Iterative method

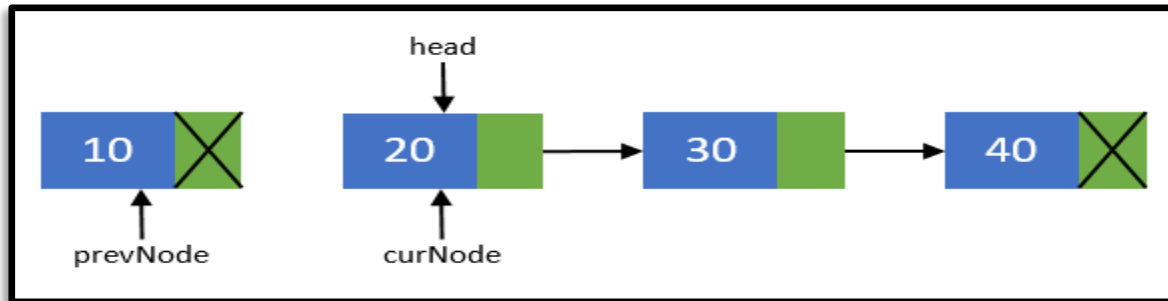
**Step 1:** Create two more pointers other than **head** namely **prevNode** and **curNode** that will hold the reference of previous node and current node respectively.

- Make sure that prevNode points to first node i.e.  $\text{prevNode} = \text{head}$ .
- head should now point to its next node i.e.  $\text{head} = \text{head} \rightarrow \text{next}$ .
- curNode should also points to the second node i.e.  $\text{curNode} = \text{head}$ .

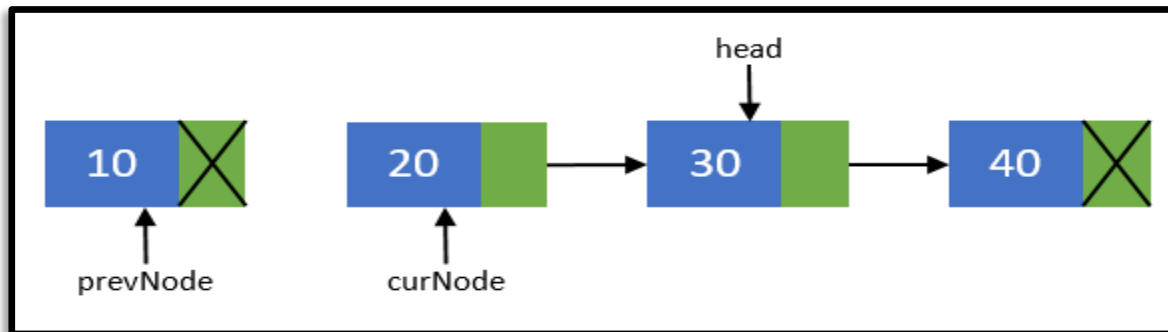


# Reversing a List

**Step 2:** Now, disconnect the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation `prevNode->next = NULL`.

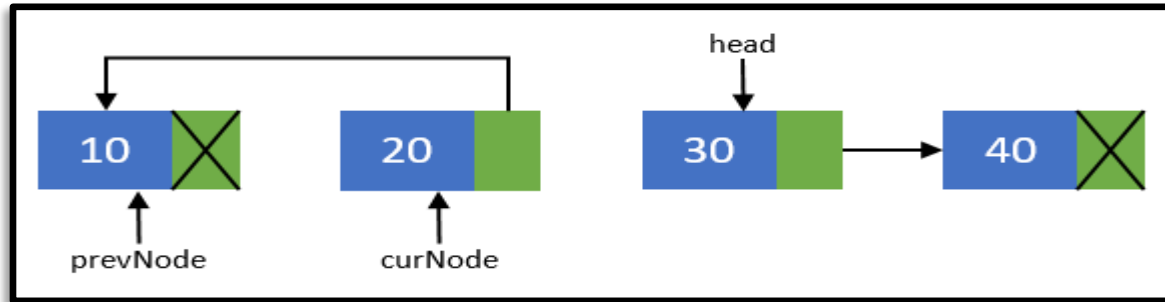


**Step 3:** Move the head node to its next node i.e. `head = head->next`.

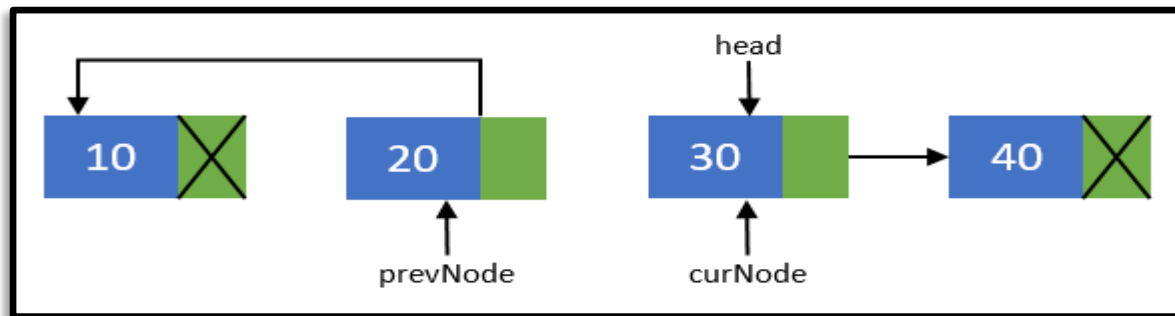


# Reversing a List

**Step 4:** Now, re-connect the current node to its previous node  
i.e. `curNode->next = prevNode;`



**Step 5:** Point the previous node to current node and current node to head node. Means they should now point to `prevNode = curNode;` and `curNode = head.`

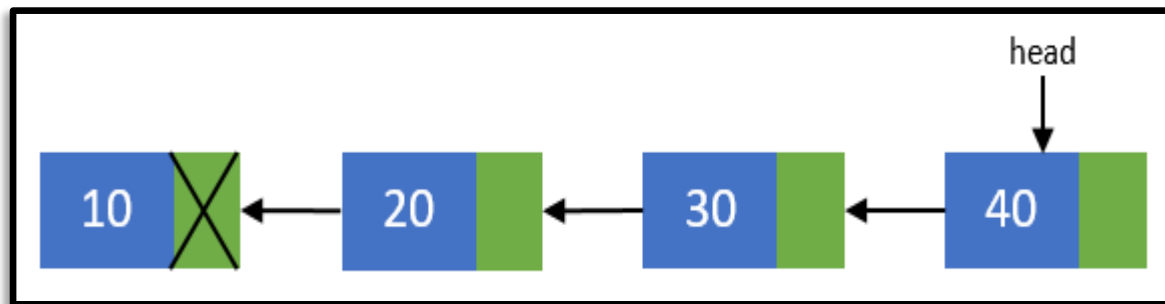


# Reversing a List

**Step 6:** Repeat steps 3-5 till head pointer becomes **NULL**.

**Step 7:** Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the head pointer should point to prevNode pointer.

- Perform `head = prevNode;` And finally you end up with a reversed linked list of its original.



# Reversing a List: Iterative Method

```
void reverseList()
{
    struct node *prevNode, *curNode;
    if(head != NULL)
    {
        prevNode = head;
        curNode = head->next;
        head = head->next;

        prevNode->next = NULL;    //Makes the first node as last node

        while(head != NULL)
        {
            head = head->next;
            curNode->next = prevNode;

            prevNode = curNode;
            curNode = head;
        }

        head = prevNode;    //Makes the last node as head
        printf("SUCCESSFULLY REVERSED LIST\n");
    }
}
```

# Reversing a List: Recursive Method

```
/* Function to reverse the linked list */
void Recursive_Reverse(struct node* head)
{
    if (head == NULL) // boundary condition to stop recursion
        return;

    Recursive_Reverse(head->next); // print the list after head node

    printf("%d  ", head->data); // After everything else is printed, print head
}

/* It should be called from the main() function as */

int main()
{
    int n = 10;
    createList(n); // creates 10 nodes in the linked list
    Recursive_Reverse(head);

    return 0;
}
```

# Single Linked List: Sorting

**The linked list can be ordered using any of the following sorting algorithms:**

- Insertion sort
- Selection sort
- Merge sort
- Quick sort
- Bubble sort, etc.

Here, we discuss **insertion** sort for ordering linked list.



# Sorting a List using Insertion Sort

```
// function to sort a singly linked list using insertion sort
void insertionSort(struct node **head_ref)
{
    // Initialize sorted linked list
    struct node *sorted = NULL;

    // Traverse the given linked list and insert every node to be sorted
    struct node *current = *head_ref;
    while (current != NULL)
    {
        struct node *next = current->next;

        // insert current in sorted linked list
        sortedInsert(&sorted, current);

        current = next; // Update current
    }

    // Update head_ref to point to sorted linked list
    *head_ref = sorted;
}
```

# Sorting a List using Insertion Sort

```
/* function to insert a new_node in a list.*/
void sortedInsert(struct node** head_ref, struct node* new_node)
{
    struct node* current;

    /* Special case for the head end */
    if (*head_ref == NULL || (*head_ref)->data >= new_node->data)
    {
        new_node->next = *head_ref;
        *head_ref = new_node;
    }
    else
    {
        /* Locate the node before the point of insertion */
        current = *head_ref;
        while (current->next!=NULL && current->next->data < new_node->data)
        {
            current = current->next;
        }
        new_node->next = current->next;
        current->next = new_node;
    }
}
```

# Sorting a List using Insertion Sort

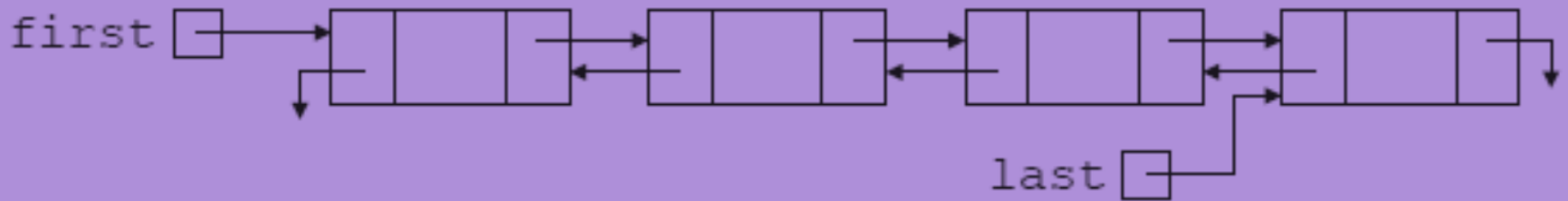
```
int main()
{
    int n=5;
    createList(n);
    insertionSort(&head);
    printf("\nData after sorting the list \n");
    displayList();
    return 0;
}
```

## Output:

```
Enter the data of node 1: 6
Enter the data of node 2: 88
Enter the data of node 3: 42
Enter the data of node 4: 21
Enter the data of node 5: 1
SINGLY LINKED LIST CREATED SUCCESSFULLY
```

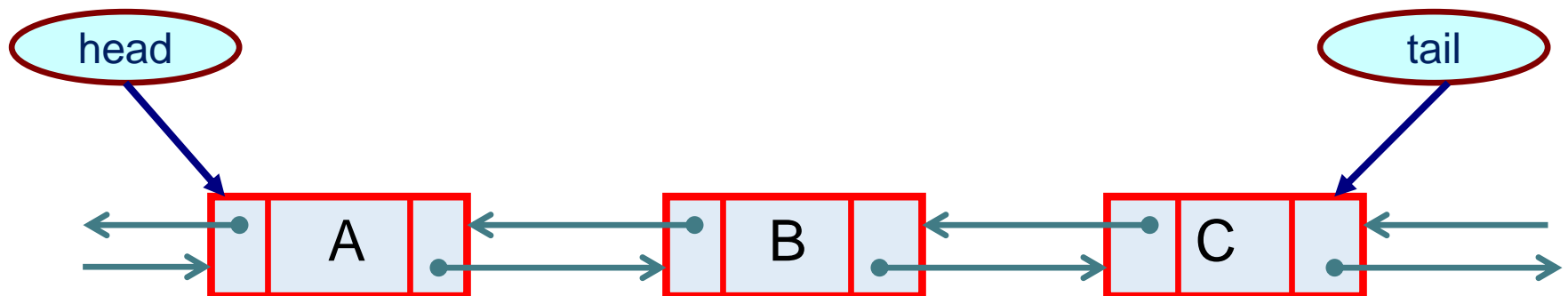
```
Data after sorting the list
Data = 1
Data = 6
Data = 21
Data = 42
Data = 88
```

# Doubly Linked List



# Doubly Linked List

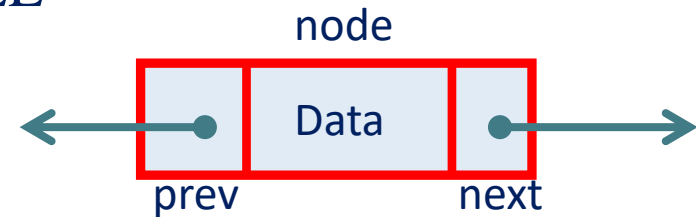
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually two pointers are maintained to keep track of the list, *head* and *tail*.



# Defining a Node of a Doubly Linked List

Each node of Doubly Linked List (DLL) consists of three fields:

- Item (or) Data
- Pointer of the next node in DLL
- Pointer of the previous node in DLL

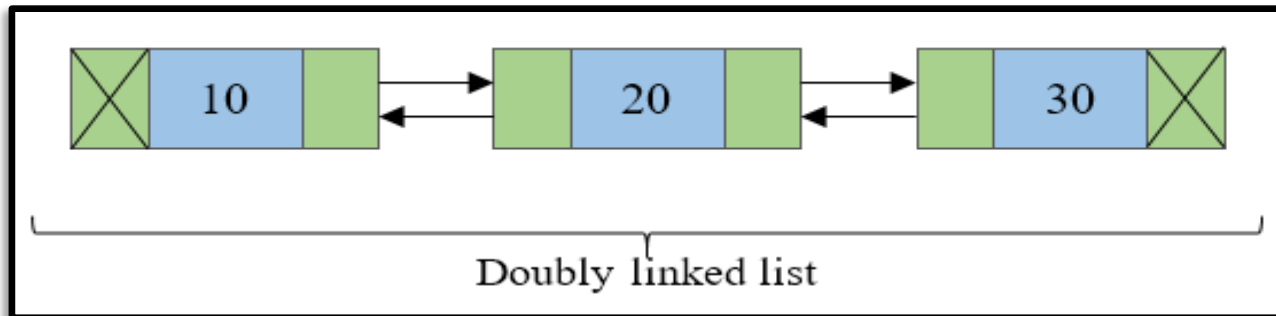


**How to define a node of a doubly linked list (DLL)?**

```
struct node
{
    int data;
    struct node *next; // Pointer to next node in DLL
    struct node *prev; // Pointer to previous node in DLL
};
```

# Doubly Linked List

- A collection of nodes linked together in a sequential way.
- Almost similar to singly linked list except it contains **two address or reference fields**, where one of the address field contains reference of the next node and other contains reference of the previous node.
- **First and last node** of a linked list contains a terminator generally a **NULL** value, that determines the start and end of the list.
- Sometimes also referred as **bi-directional linked list** since it allows traversal of nodes in both direction.
- Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.



# Doubly versus Single Linked List

## **Advantages over singly linked list**

- 1) A DLL can be traversed in both forward and backward direction.
- 2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

## **Disadvantages over singly linked list**

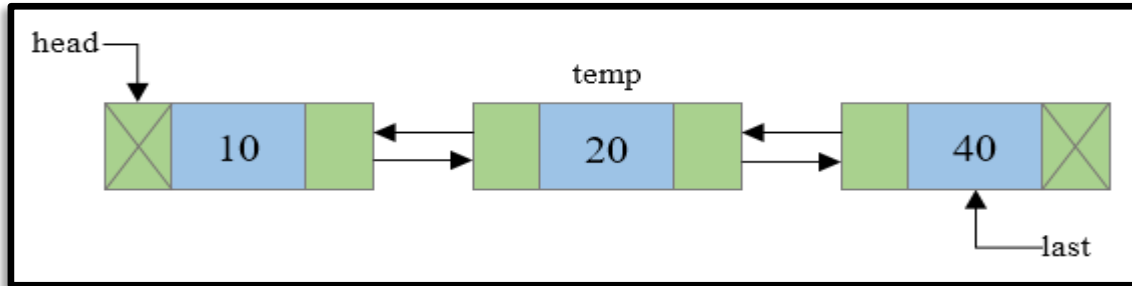
- 1) Every node of DLL Require extra space for an previous pointer.
- 2) All operations require an extra pointer previous to be maintained.



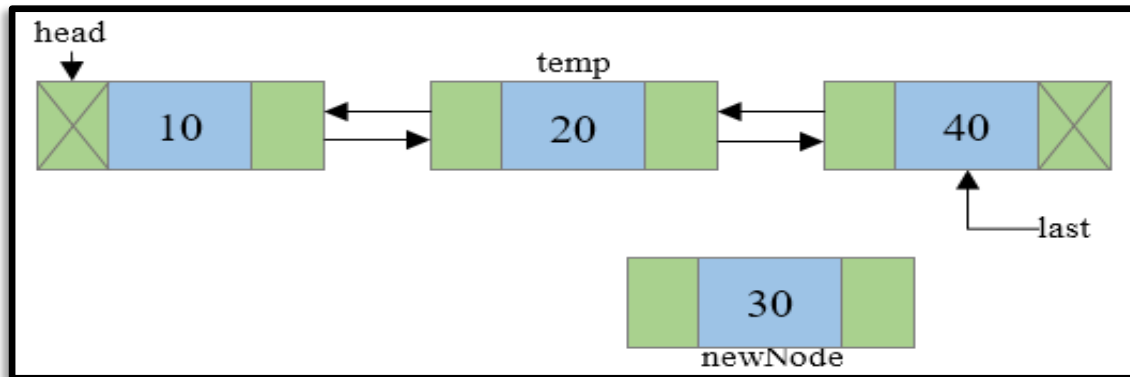
# Doubly Linked List: Insertion at any Position

**Steps to insert a new node at  $n^{\text{th}}$  position in a Doubly linked list.**

**Step 1:** Traverse to  $N-1$  node in the list, where  $N$  is the position to insert. Say **temp** now points to  $N-1^{\text{th}}$  node.

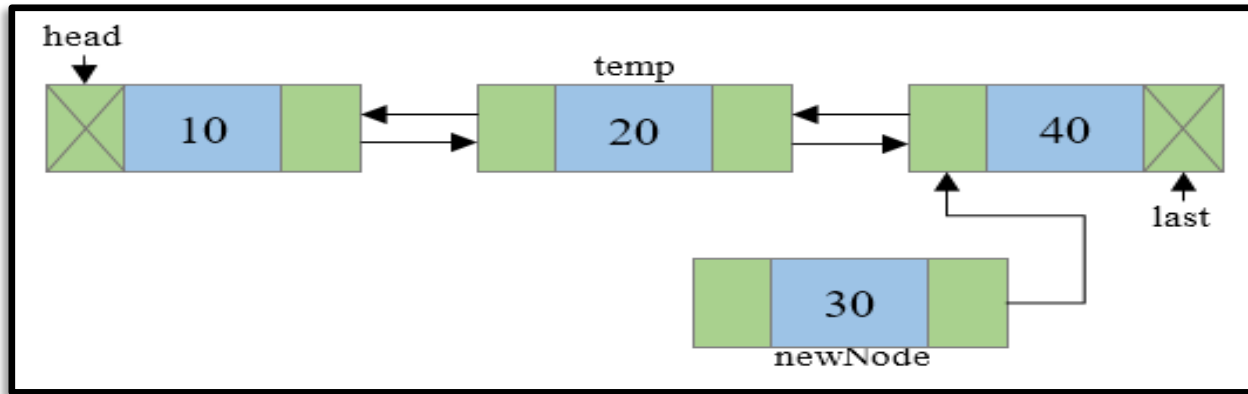


**Step 2:** Create a **newNode** that is to be inserted and assign some data to its data field.

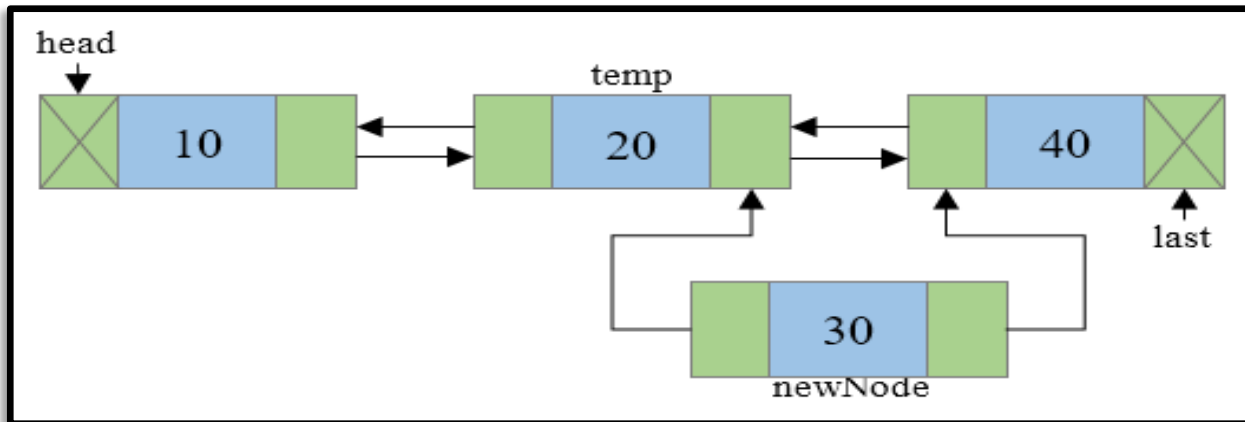


# Doubly Linked List: Insertion at any Position

**Step 3:** Connect the next address field of **newNode** with the node pointed by next address field of **temp** node.

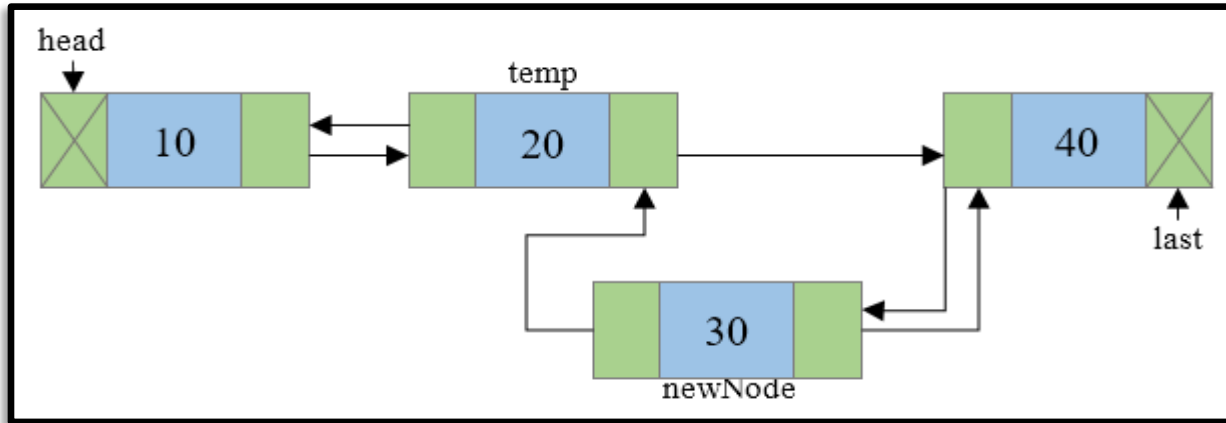


**Step 4:** Connect the previous address field of **newNode** with the **temp** node.

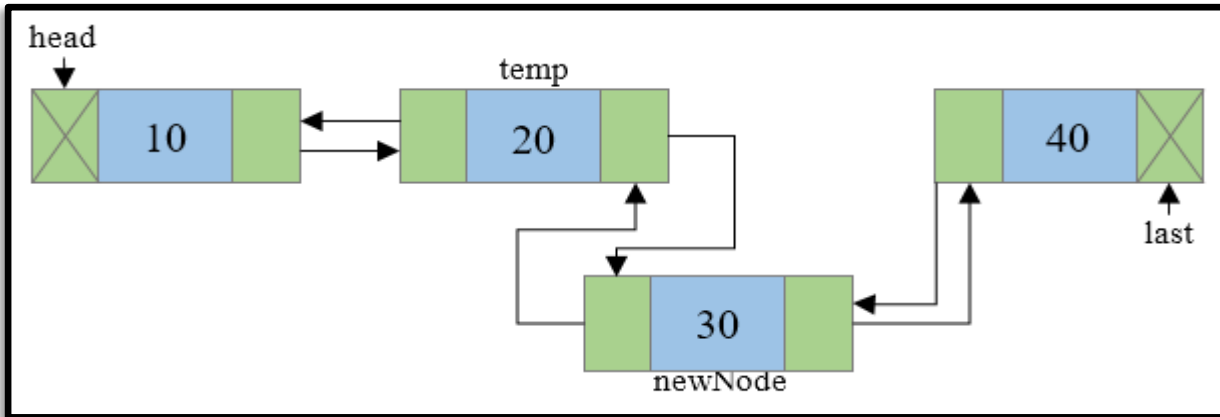


# Doubly Linked List: Insertion at any Position

**Step 5:** Check if `temp.next` is not NULL then, connect the previous address field of node pointed by `temp.next` to `newNode`.

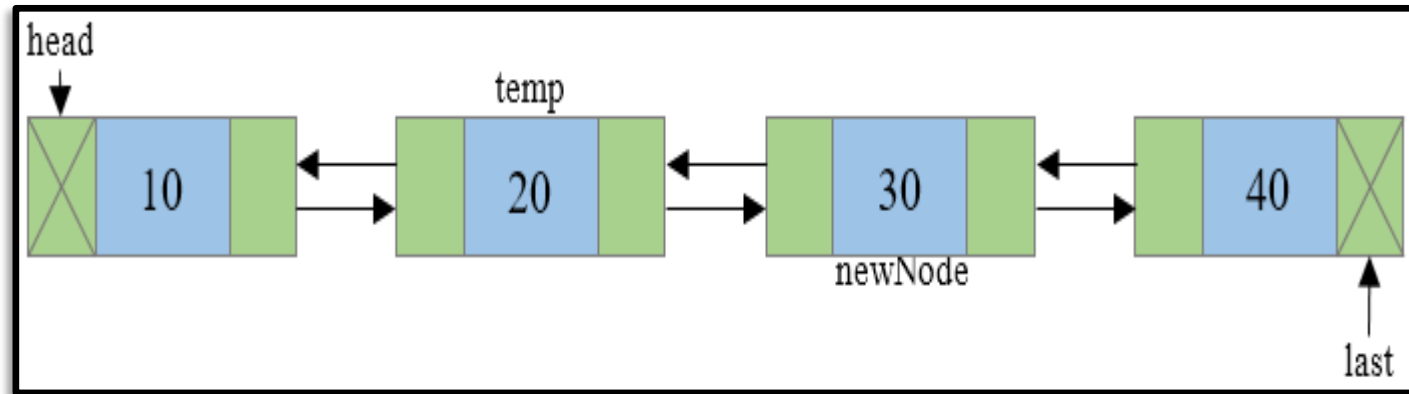


**Step 6:** Connect the next address field of `temp` node to `newNode`.



# Doubly Linked List: Insertion at any Position

**Step 7:** Final doubly linked list looks like



# Doubly Linked List: Insertion at any Position

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

struct node {                                /* Basic structure of Node */
    int data;
    struct node * prev;
    struct node * next;
}*head, *last;

int main()
{
    int n, data;
    head = NULL;
    last = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);                            // function to create Doubly linked list
    displayList();                            // function to display the list

    printf("Enter the position and data to insert new node: ");
    scanf("%d %d", &n, &data);
    insert_position(data, n);                 // function to insert node at any position
    displayList();
    return 0;
}
```

# Doubly Linked List: Insertion at any Position

```
void createList(int n)
{
    int i, data;
    struct node *newNode;
    if(n >= 1){                /* Creates and links the head node */
        head = (struct node *)malloc(sizeof(struct node));
        printf("Enter data of 1 node: ");
        scanf("%d", &data);
        head->data = data;
        head->prev = NULL;
        head->next = NULL;

        last = head;

        for(i=2; i<=n; i++){    /* Creates and links rest of the n-1 nodes */
            newNode = (struct node *)malloc(sizeof(struct node));
            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->prev = last;        //Links new node with the previous node
            newNode->next = NULL;

            last->next = newNode; //Links previous node with the new node
            last = newNode; //Makes new node as last/previous node
        }
        printf("\nDOUBLY LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

# Doubly Linked List: Insertion at any Position

```
void insert_position(int data, int position)
{
    struct node * newNode, *temp;
    if(head == NULL){
        printf("Error, List is empty!\n");
    }
    else{
        temp = head;
        if(temp!=NULL){
            newNode = (struct node *)malloc(sizeof(struct node));

            newNode->data = data;
            newNode->next = temp->next; //Connects new node with n+1th node
            newNode->prev = temp;       //Connects new node with n-1th node

            if(temp->next != NULL)
            {
                temp->next->prev = newNode; /* Connects n+1th node with new node */
            }
            temp->next = newNode;           /* Connects n-1th node with new node */
            printf("NODE INSERTED SUCCESSFULLY AT %d POSITION\n", position);
        }
        else{
            printf("Error, Invalid position\n");
        }
    }
}
```

# Doubly Linked List: Insertion at any Position

```
void displayList()
{
    struct node * temp;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        temp = head;
        printf("DATA IN THE LIST:\n");

        while(temp != NULL)
        {
            printf("DATA of %d node = %d\n", n, temp->data);
            n++;

            /* Moves the current pointer to next node */
            temp = temp->next;
        }
    }
}
```



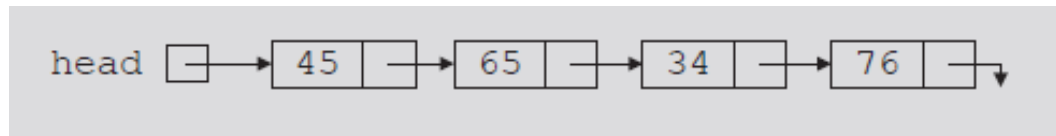
# Self Check

**For doubly linked list write a function to:**

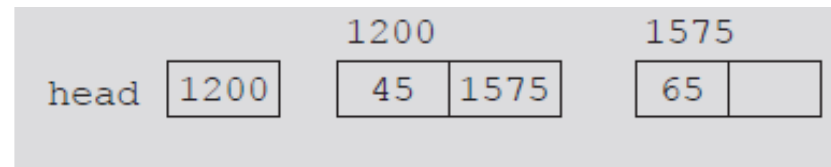
- Insert a node at front of the list and at end of the list.  
`insert_front(data) ;`  
`insert_end(data) ;`
- Sort the DLL in ascending order.
- Count the number of nodes in the given DLL.

# Cross Check...

- Can we create a doubly linked list using one pointer with every node?



Linked list



Linked list and values of the links

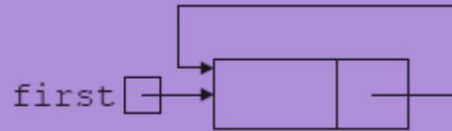
# Cross Check: Solution

- Possible to create a Doubly Linked List using ONLY One Pointer Per Node
- Achieved by XOR-Linking
  - Storing the XOR of the addresses of the previous node and the next node in place of the two pointers
  - XOR Operation Property
    - $(A \text{ XOR } B) \text{ XOR } A = B$
    - $(A \text{ XOR } B) \text{ XOR } B = A$
  - For each node, store **(Address of Previous Node) XOR (Address of Next Node)**
  - To traverse nodes in this list
    1. **Forward Direction**
      - Start with a Pointer to the Current Node and a **Pointer to its Predecessor**
      - **Next Node Address = (Address of Predecessor) XOR (Current Node's XOR Value)**
    2. **Backward Direction**
      - Start with a Pointer to the Current Node and a **Pointer to its Successor**
      - **Previous Node Address = (Address of Successor) XOR (Current Node's XOR Value)**

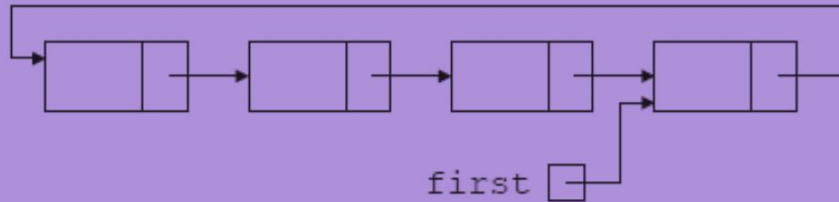
# Circular Linked List

first 

(a) Empty circular list



(b) Circular linked list with one node

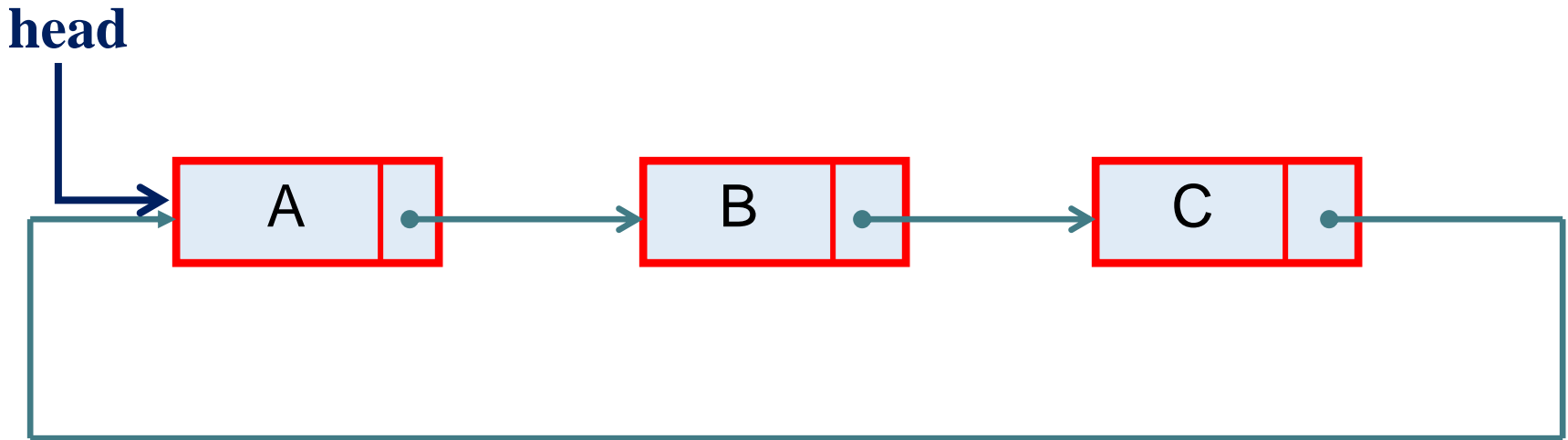


(c) Circular linked list with more than one node

# Circular Linked List

## Circular linked list

- The pointer from the last element in the list points back to the first element.

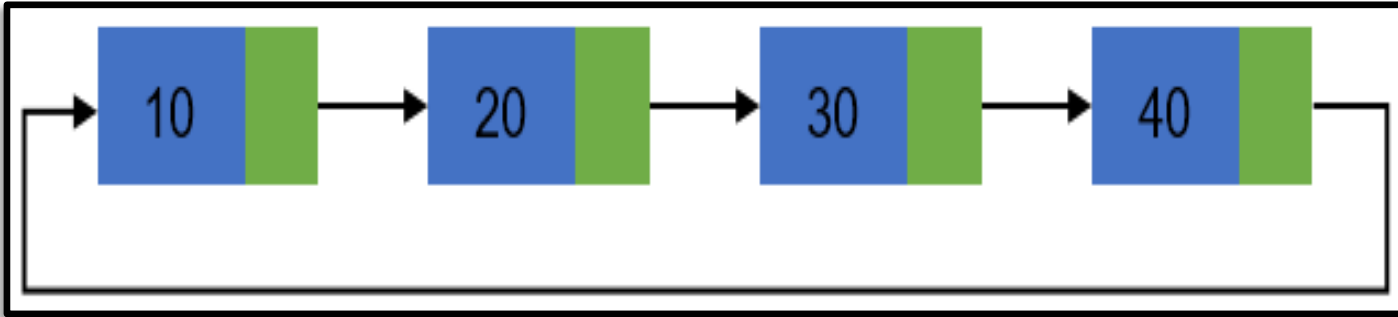


# Circular Linked List

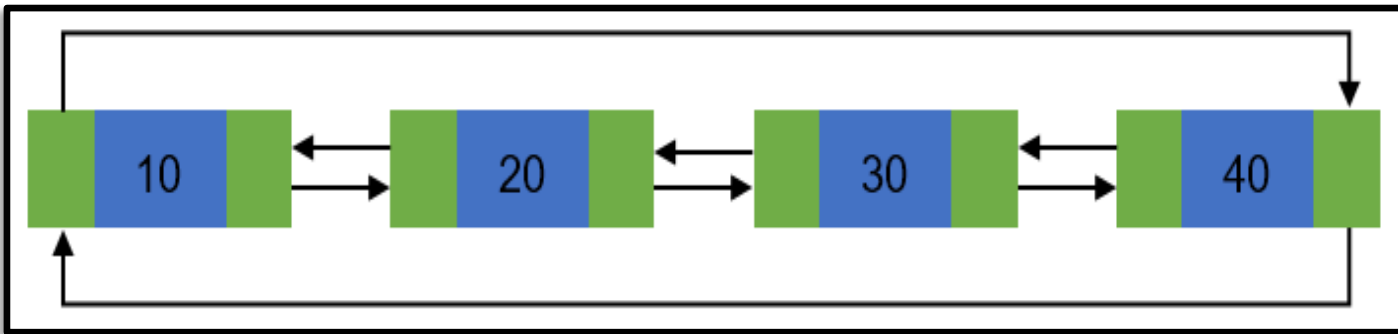
- Basically a linear linked list that may be **single-** or **Doubly-linked**.
- The only difference is that **there is no any NULL** value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a **circle** with **no end to stop** it is called as **circular linked list**.
- There can be no starting or ending node, whole node can be **traversed from any node**.
- In order to traverse the circular linked list, only once we need to traverse entire list until the **starting node is not traversed again**.
- Can be implemented using both **singly linked list** and **doubly linked list**.

# Circular Linked List

**Basic structure of singly circular linked list:**



**Doubly circular linked list:**



# Circular Linked List

## **Advantages of a Circular linked list**

- Entire list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- Despite of being singly circular linked list we can easily traverse to its previous node, which is not possible in singly linked list.

## **Disadvantages of Circular linked list**

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn't supports direct accessing of elements.



# Operations on Circular Linked List

- Creation of list
- Traversal of list
- Insertion of node
  - At the beginning of list
  - At any position in the list
- Deletion of node
  - Deletion of first node
  - Deletion of node from middle of the list
  - Deletion of last node
- Counting total number of nodes
- Reversing of list

# Creation and Traversal of a Circular List

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

/* Basic structure of Node */

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

int main()
{
    int n, data;
    head = NULL;

    printf("Enter the total number of nodes in list: ");
    scanf("%d", &n);
    createList(n);          // function to create circular linked list
    displayList();          // function to display the list

    return 0;
}
```

# Circular Linked List: Creation of List

```
void createList(int n)
{
    int i, data;
    struct node *prevNode, *newNode;
    if(n >= 1){
        head = (struct node *)malloc(sizeof(struct node));

        printf("Enter data of 1 node: ");
        scanf("%d", &data);

        head->data = data;
        head->next = NULL;
        prevNode = head;

        for(i=2; i<=n; i++){
            newNode = (struct node *)malloc(sizeof(struct node));

            printf("Enter data of %d node: ", i);
            scanf("%d", &data);

            newNode->data = data;
            newNode->next = NULL;
            prevNode->next = newNode; //Links the previous node with newly created node
            prevNode = newNode;      //Moves the previous node ahead
        }

        prevNode->next = head; //Links the last node with first node
        printf("\nCIRCULAR LINKED LIST CREATED SUCCESSFULLY\n");
    }
}
```

# Circular Linked List: Traversal of List

```
void displayList()
{
    struct node *current;
    int n = 1;

    if(head == NULL)
    {
        printf("List is empty.\n");
    }
    else
    {
        current = head;
        printf("DATA IN THE LIST:\n");

        do {
            printf("Data %d = %d\n", n, current->data);

            current = current->next;
            n++;
        }while(current != head);
    }
}
```

# Self Check

**For circular linked list write a function to:**

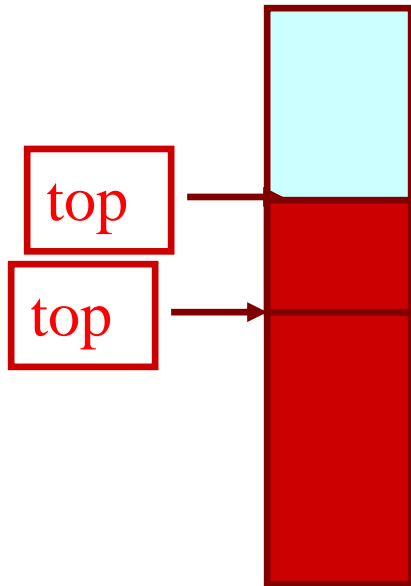
- Insert a node at any position of the list and delete from the beginning of the list.  

```
insert_position(data, position);  
delete_front();
```
- Reverse the given circular linked link.

# Stack Implementations Using Array and Linked List

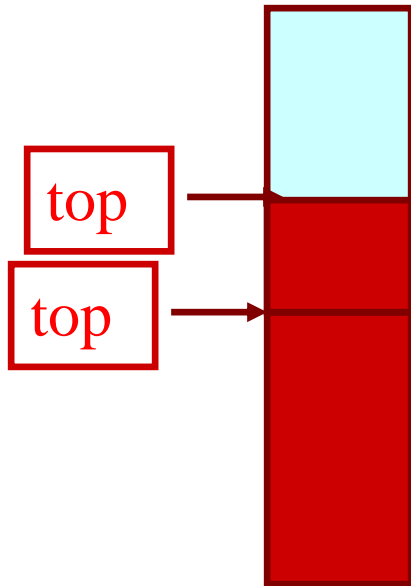
# Stack: Using Array

PUSH



# Stack: Using Array

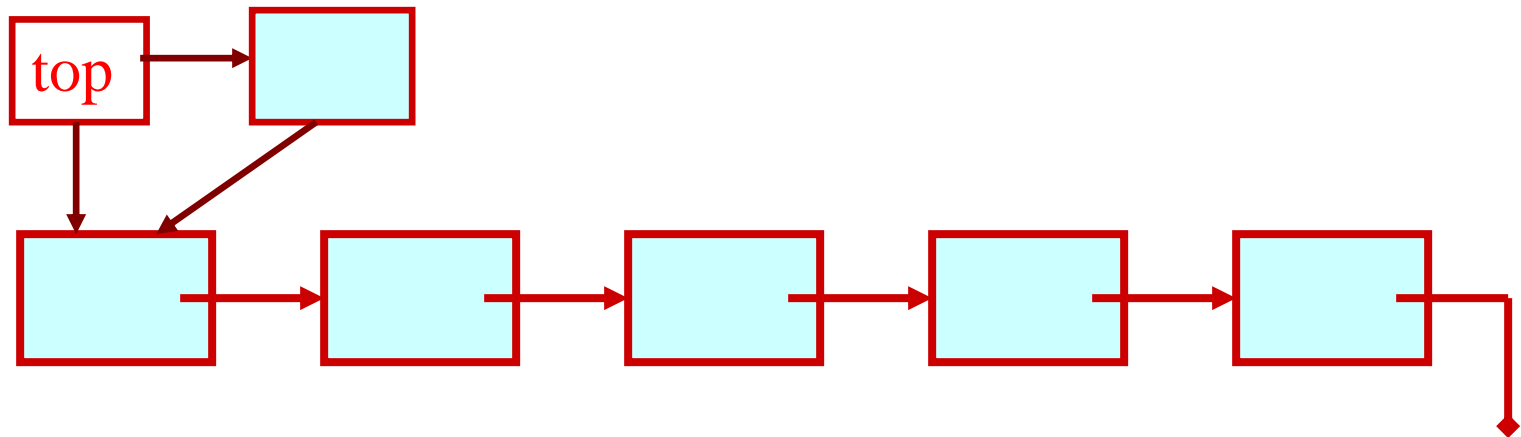
POP





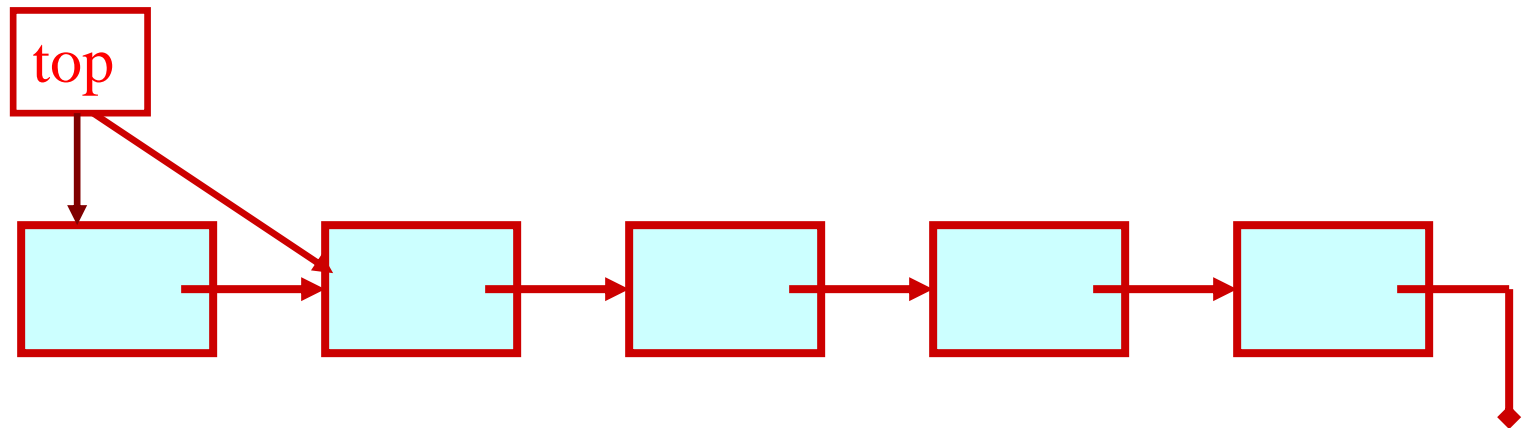
# Stack: Linked List Structure

## PUSH OPERATION



# Stack: Linked List Structure

## POP OPERATION



# Basic Idea

- In the array implementation, we would:
  - Declare an array of fixed size (which determines the maximum size of the stack).
  - Keep a variable which always points to the “top” of the stack.
    - Contains the array index of the “top” element.
- In the linked list implementation, we would:
  - Maintain the stack as a linked list.
  - A pointer variable `top` points to the start of the list.
  - The first element of the linked list is considered as the stack top.

# Declaration

```
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int top;
};
typedef struct lifo
        stack;

stack s;
```

ARRAY

```
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo
        stack;

stack *top;
```

LINKED LIST

# Stack Creation

```
void create (stack *s)
{
    s->top = -1;

    /* s->top points to
       last element
       pushed in;
       initially -1 */
}
```

**ARRAY**

```
void create (stack **top)
{
    *top = NULL;

    /* top points to NULL,
       indicating empty
       stack */
}
```

**LINKED LIST**

# Pushing an element into the stack

```
void push (stack *s, int element)
{
    if (s->top == (MAXSIZE-1))
    {
        printf ("\n Stack overflow");
        exit(-1);
    }
    else
    {
        s->top ++;
        s->st[s->top] = element;
    }
}
```

**ARRAY**

```
void push (stack **top, int element)
{
    stack *new;

    new = (stack *) malloc(sizeof(stack));
    if (new == NULL)
    {
        printf ("\n Stack is full");
        exit(-1);
    }

    new->value = element;
    new->next = *top;
    *top = new;
}
```

## LINKED LIST

# Popping an Element from the Stack

```
int pop (stack *s)
{
    if (s->top == -1)
    {
        printf ("\n Stack underflow");
        exit(-1);
    }
    else
    {
        return (s->st[s->top--]);
    }
}
```

**ARRAY**



```
int pop (stack **top)
{
    int t;
    stack *p;
    if (*top == NULL)
    {
        printf ("\n Stack is empty");
        exit(-1);
    }
    else
    {
        t = (*top)->value;
        p = *top;
        *top = (*top)->next;
        free (p);
        return t;
    }
}
```

## LINKED LIST

# Checking for Stack Empty

```
int isempty (stack *s)
{
    if (s->top == -1)
        return 1;
    else
        return (0);
}
```

ARRAY

```
int isempty (stack *top)
{
    if (top == NULL)
        return (1);
    else
        return (0);
}
```

LINKED LIST

# Checking for Stack Full

```
int isfull (stack *s)
{
    if (s->top ==
        (MAXSIZE-1))
        return 1;
    else
        return (0);
}
```

ARRAY

- Not required for linked list implementation.
- In the `push()` function, we can check the return value of `malloc()`.
  - If -1, then memory cannot be allocated.

LINKED LIST

# Example Main Function :: Array

```
#include <stdio.h>
#define MAXSIZE 100

struct lifo
{
    int st[MAXSIZE];
    int top;
};
typedef struct lifo stack;
```

```
main()
{
    stack A, B;
    create(&A);  create(&B);
    push(&A,10);
    push(&A,20);
```

```
    push(&A,30);
    push(&B,100);  push(&B,5);

    printf ("%d %d", pop(&A),
            pop(&B));

    push (&A, pop(&B));

    if (isempty(&B))
        printf ("\n B is empty");
}
```

# Example Main Function :: Linked List

```
#include <stdio.h>
struct lifo
{
    int value;
    struct lifo *next;
};
typedef struct lifo stack;
```

```
main()
{
    stack *A, *B;
    create(&A); create(&B);
    push(&A, 10);
    push(&A, 20);
```

```
    push(&A, 30);
    push(&B, 100);
    push(&B, 5);

    printf ("%d %d",
            pop(&A), pop(&B));

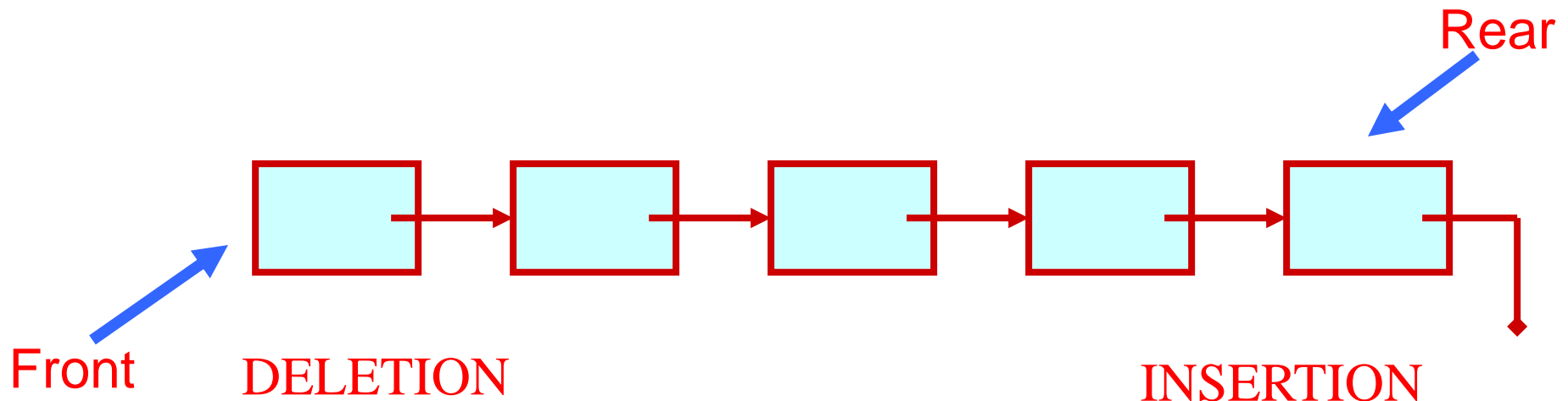
    push (&A, pop(&B));

    if (isempty(B))
        printf ("\n B is
empty");
}
```

# Queue Implementation Using Linked List

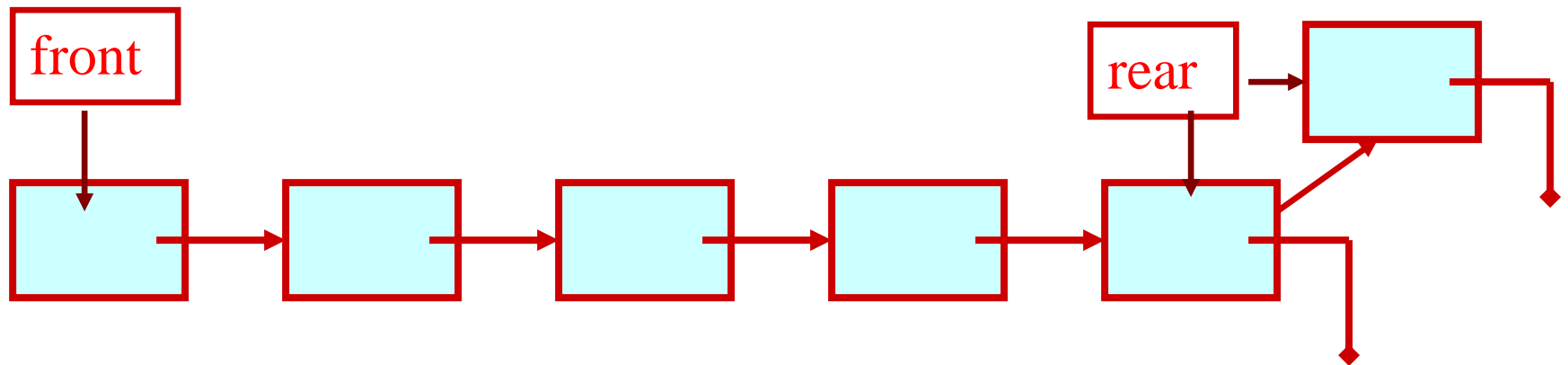
# Basic Idea

- Create a linked list to which items would be added to one end and deleted from the other end.
- Two pointers will be maintained:
  - One pointing to the beginning of the list (point from where elements will be deleted).
  - Another pointing to the end of the list (point where new elements will be inserted).



# Queue: Linked List Structure

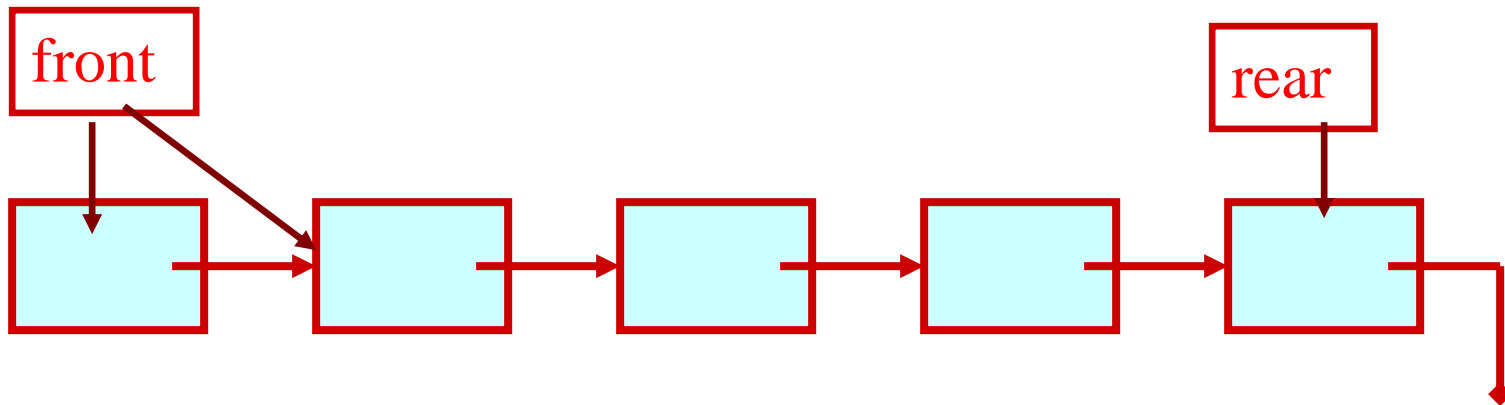
ENQUEUE





# Queue: Linked List Structure

DEQUEUE



# Queue using Linked List

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

struct node{
    char name[30];
    struct node *next;
};

typedef struct node _QNODE;

typedef struct {
    _QNODE *queue_front, *queue_rear;
} QUEUE;
```

```

QNODE *enqueue (QUEUE *q, char x[])
{
    QNODE *temp;
    temp= (QNODE *)
        malloc (sizeof(QNODE));
    if (temp==NULL){
        printf("Bad allocation \n");
        return NULL;
    }
    strcpy(temp->name,x);
    temp->next=NULL;

    if(q->queue_rear==NULL)
    {
        q->queue_rear=temp;
        q->queue_front=
            q->queue_rear;
    }
    else
    {
        q->queue_rear->next=temp;
        q->queue_rear=temp;
    }
    return(q->queue_rear);
}

```

```
char *dequeue(_QUEUE *q,char x[])
{
    QNODE *temp_pnt;
```

```
    if(q->queue_front==NULL){
        q->queue_rear=NULL;
        printf("Queue is empty \n");
        return(NULL);
    }
```

```
    else{
        strcpy(x,q->queue_front->name);
        temp_pnt=q->queue_front;
        q->queue_front=
            q->queue_front->next;
        free(temp_pnt);
        if(q->queue_front==NULL)
            q->queue_rear=NULL;
        return(x);
    }
}
```

```
void init_queue(_QUEUE *q)
{
    q->queue_front= q->queue_rear=NULL;
}
```

```
int isEmpty(_QUEUE *q)
{
    if(q==NULL) return 1;
    else return 0;
}
```

```
main()
{
int i,j;
char command[5],val[30];
QUEUE q;

init_queue(&q);

command[0]='\0';
printf("For entering a name use 'enter <name>\n");
printf("For deleting use 'delete' \n");
printf("To end the session use 'exit' \n");
while(strcmp(command,"exit")){
scanf("%s", command);
```

```
if(!strcmp(command,"enter")) {  
    scanf("%s",val);  
    if((enqueue(&q,val)==NULL))  
        printf("No more pushing please \n");  
    else printf("Name entered %s \n",val);  
}
```

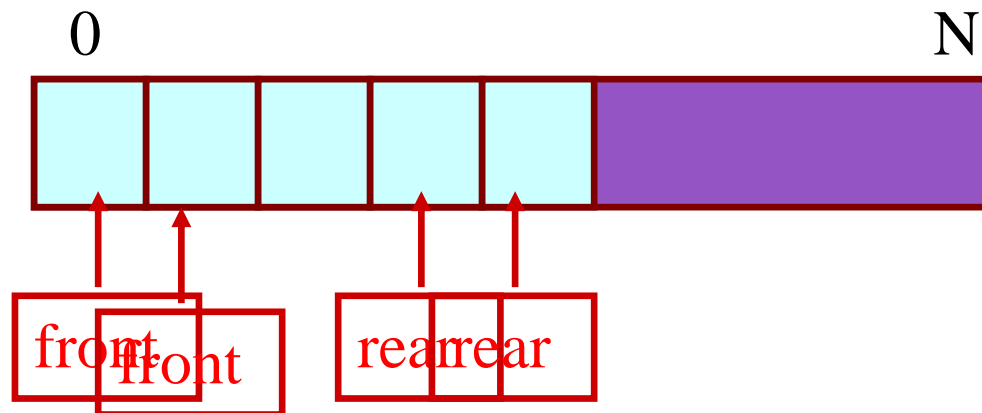
```
    if(!strcmp(command,"delete")) {  
        if(!isEmpty(&q))  
            printf("%s \n",dequeue(&q,val));  
        else printf("Name deleted %s \n",val);  
    }  
} /* while */  
printf("End session \n");  
}
```

# Problem with Array Implementation

ENQUEUE

DEQUEUE

Effective queuing storage area of array gets reduced.



Use of circular array indexing



# Queue: Example with Array Implementation

```
#define MAX_SIZE 100
```

```
typedef struct { char name[30];  
                } _ELEMENT;
```

```
typedef struct {  
    _ELEMENT q_elem[MAX_SIZE];  
    int rear;  
    int front;  
    int full,empty;  
} _QUEUE;
```

## Queue Example: Contd.

```
void init_queue(_QUEUE *q)
{
    q->rear = q->front = 0;
    q->full = 0; q->empty = 1;
}
```

```
int IsFull(_QUEUE *q)
{
    return(q->full);
}
```

```
int IsEmpty(_QUEUE *q)
{
    return(q->empty);
}
```

## Queue Example: Contd.

```
void AddQ(Queue *q, ELEMENT ob)
{
    if(IsFull(q)) {printf("Queue is Full \n"); return;}

    q->rear=(q->rear+1)%(MAX_SIZE);
    q->q_elem[q->rear]=ob;

    if(q->front==q->rear) q->full=1; else q->full=0;
    q->empty=0;

    return;
}
```

# Queue Example: Contd.

```
ELEMENT DeleteQ(Queue *q)
```

```
{
```

```
    _ELEMENT temp;
```

```
    temp.name[0]='\0';
```

```
    if(IsEmpty(q)) {printf("Queue is EMPTY\n");return(temp);}

    q->front=(q->front+1)%(MAX_SIZE);
    temp=q->q_elem[q->front];

    if(q->rear==q->front) q->empty=1; else q->empty=0;
    q->full=0;

    return(temp);
}
```

# Queue Example: Contd.

```
main()
{
int i,j;
char command[5];
ELEMENT ob;
QUEUE A;

init_queue(&A);

command[0]='\0';
printf("For adding a name use 'add [name]'\n");
printf("For deleting use 'delete' \n");
printf("To end the session use 'exit' \n");
```

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

## Queue Example: Contd.

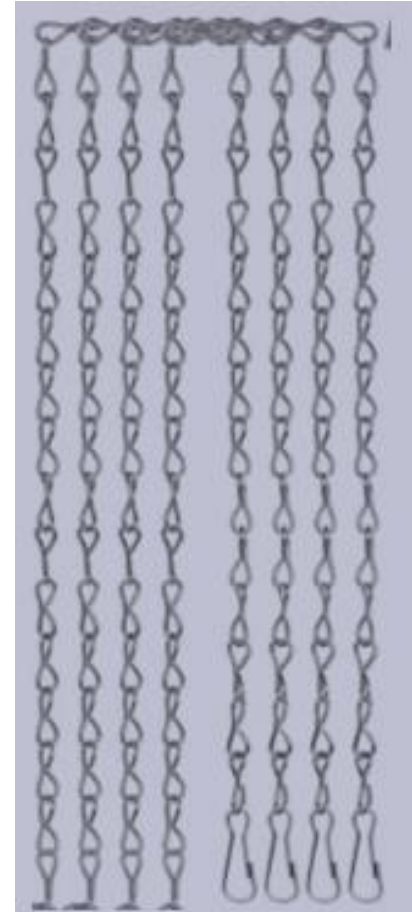
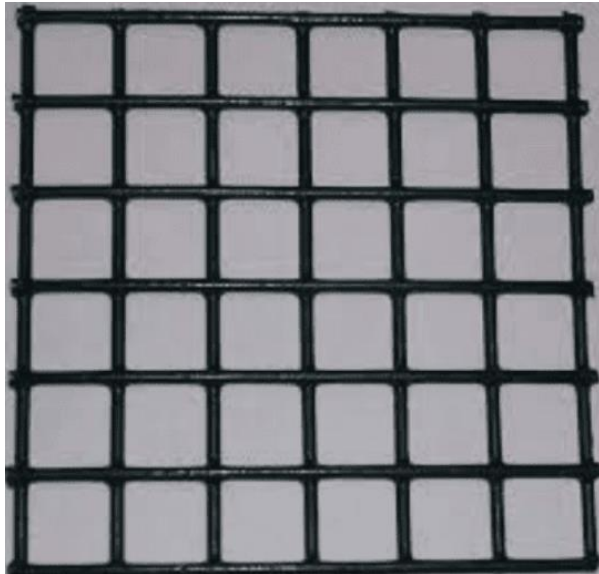
```
while (strcmp(command,"exit")!=0){
    scanf("%s",command);

    if(strcmp(command,"add")==0) {
        scanf("%s",ob.name);
        if (IsFull(&A))
            printf("No more insertion please \n");
        else {
            AddQ(&A,ob);
            printf("Name inserted %s \n",ob.name);
        }
    }
}
```

## Queue Example: Contd.

```
    if (strcmp(command,"delete")==0) {  
        if (IsEmpty(&A))  
            printf("Queue is empty \n");  
        else {  
            ob=DeleteQ(&A);  
            printf("Name deleted %s \n",ob.name);  
        }  
    }  
} /* End of while */  
printf("End session \n");  
}
```

# Are Multidimensional Arrays and Nested Lists the Same?





# Are Multidimensional Arrays and Nested Lists the Same?

- In the first one, the layout is rigid
  - If want to go directly to a particular coordinate, can navigate directly to it.
    - The layout is fixed and regular.
- For the second one
  - Those chain links could be all jumbled up
  - They are just laid out nice for the photo
  - Its layout is very flexible
  - To find a particular coordinate, you are going to have to feel your way down the spine, and then down one of the chains.
- Arrays and nested linked lists are rather similar
- Arrays have rigid regular layout. If you know where one element is, you know where all of them are, and can navigate in  $O(1)$  time.
- Linked lists are extremely flexible, since each link could be anywhere in memory. Need to follow the links sequentially to find a given link.
- Random access in a  $n \times n$ -dimensional array is  $O(1)$
- Random access in an  $n$ -dimensional nested list is

$$O(\sum_{k=1}^n d_k) \quad \text{where } d_1, d_2, \text{ etc. are the dimensions.}$$

If you try to solve problems yourself, then you will learn many things.

Enjoy the study...