Linked List

Liner Abstract Data Type

Topics to be covered

- Linear Data Structures – LIST: List as an ADTLinked List
- Implementation, singly linked lists, circularly linked lists, doubly-linked lists, All operations (Insertion, Deletion, Merge, Traversal, etc.) and their analysis,
- Applications of linked lists - (Polynomial Addition).

Static Vs Dynamic Memory Allocation

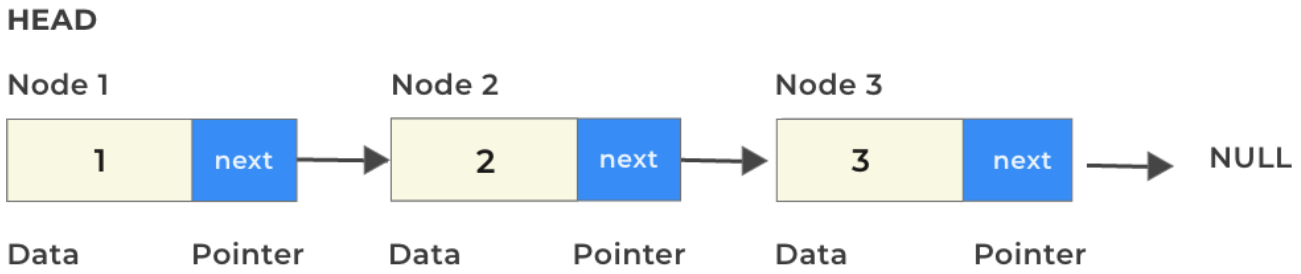
Static Memory Allocation	Dynamic Memory Allocation
Static memory allocation is a memory management technique that involves reserving a fixed amount of memory for a variable at the time of program compilation.	Dynamic memory allocation is a memory management technique that involves reserving memory for variables at runtime. This means that memory is allocated and deallocated as required during the program execution.
ADVANTAGES	**ADVANTAGES**
Faster Access: Since the memory is allocated at compile time, accessing static memory is faster compared to dynamic memory. This is because the memory address is known at the time of compilation.	**Flexible Memory Usage:** Dynamic memory allocation allows the size of the data structure to be changed dynamically during program execution. This makes it more flexible than static memory allocation.
No Overhead: Static memory allocation does not require any runtime overhead for memory allocation and deallocation. This makes it more efficient than dynamic memory allocation.	**Efficient Memory Usage:** Dynamic memory allocation allows memory to be allocated only when it is needed, which makes it more efficient than static memory allocation. This results in less wastage of memory.
Persistent Data: Static variables and arrays retain their data throughout the life of the program. This is useful when data needs to be shared between different functions.	**Global Access:** Dynamic memory can be accessed globally, which means that it can be shared between different functions.

Static Vs Dynamic Memory Allocation

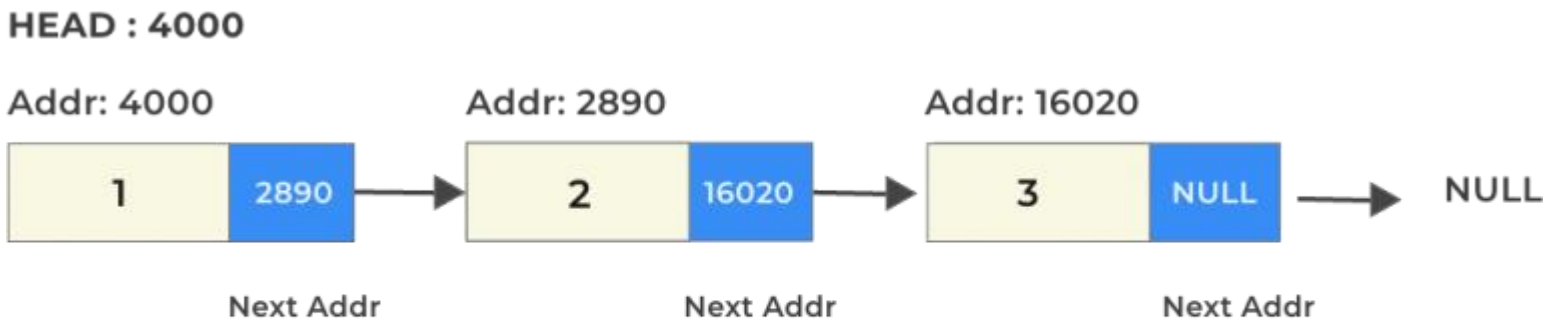
Static Memory Allocation	Dynamic Memory Allocation
DISADVANTAGES	**DIS ADVANTAGES**
Limited Flexibility: Static memory allocation is inflexible because the size of the memory is fixed at compile time. This means that if the size of the data structure needs to be changed, the entire program needs to be recompiled.	**Slower Access:** Accessing dynamic memory is slower compared to static memory because the memory address is not known at compile time. The memory address must be looked up during program execution.
Wastage of Memory: If the size of the data structure is not known in advance, static memory allocation can result in the wastage of memory.	**Memory Leaks:** Dynamic memory allocation can result in memory leaks if memory is not deallocated properly. This can cause the program to crash or slow down.
Limited Scope: Static variables are only accessible within the function where they are defined, or globally if they are defined outside of any function.	**Fragmentation:** Dynamic memory allocation can result in memory fragmentation if the memory is not allocated and deallocated properly. Memory fragmentation occurs when there are small unused gaps between allocated memory blocks. These gaps can prevent larger memory blocks from being allocated, even if there is enough total memory available.

What is Linked List?

- Linked list is linear abstract data type which organize memory(most of the time Dynamically) and perform operations to manipulate the same.
- Structure of Linked List:



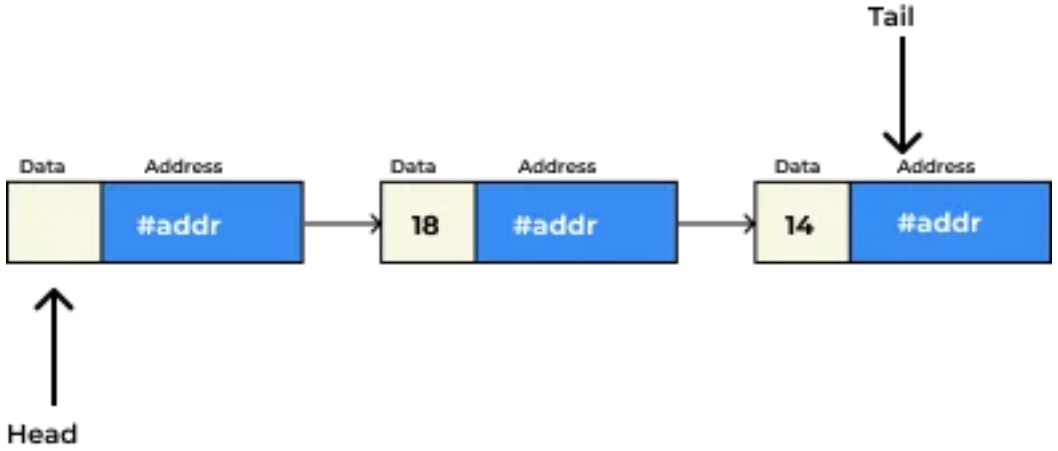
Next Pointer has the address of Next Node



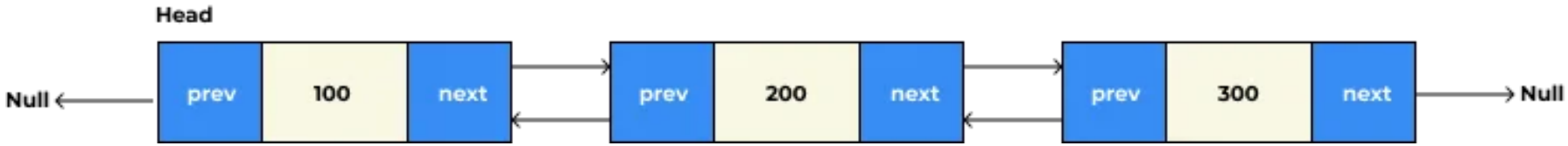
Next Pointer has the address of Next Node

Types of Linked List:

- **Singly Linked Lists:**



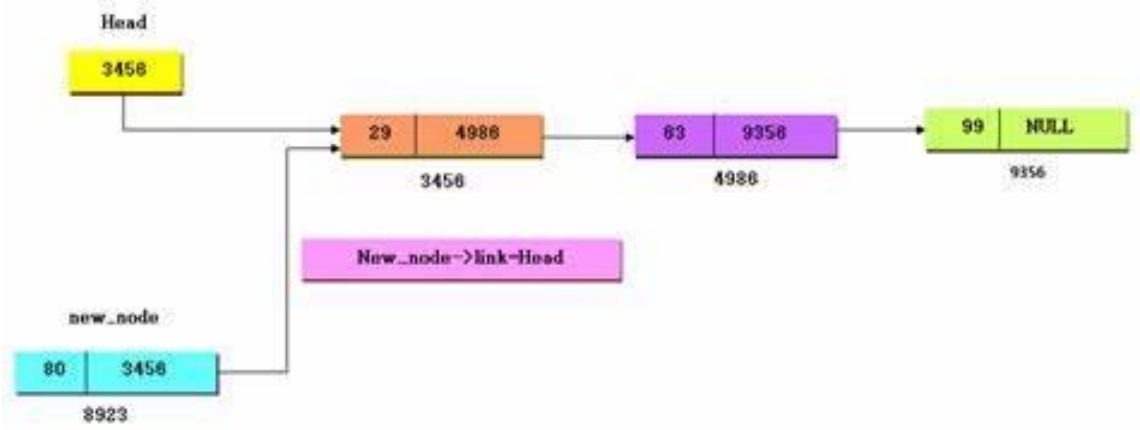
- **Double Linked Lists:**

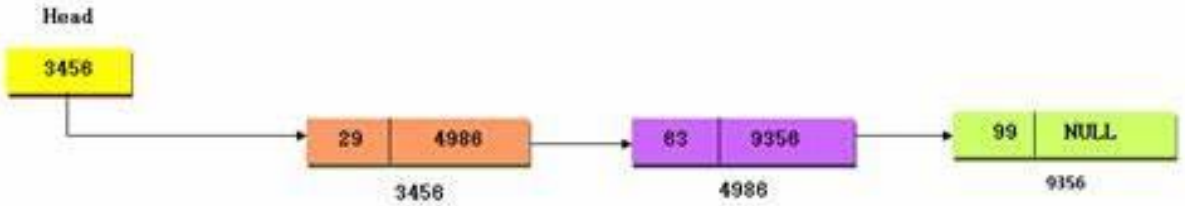


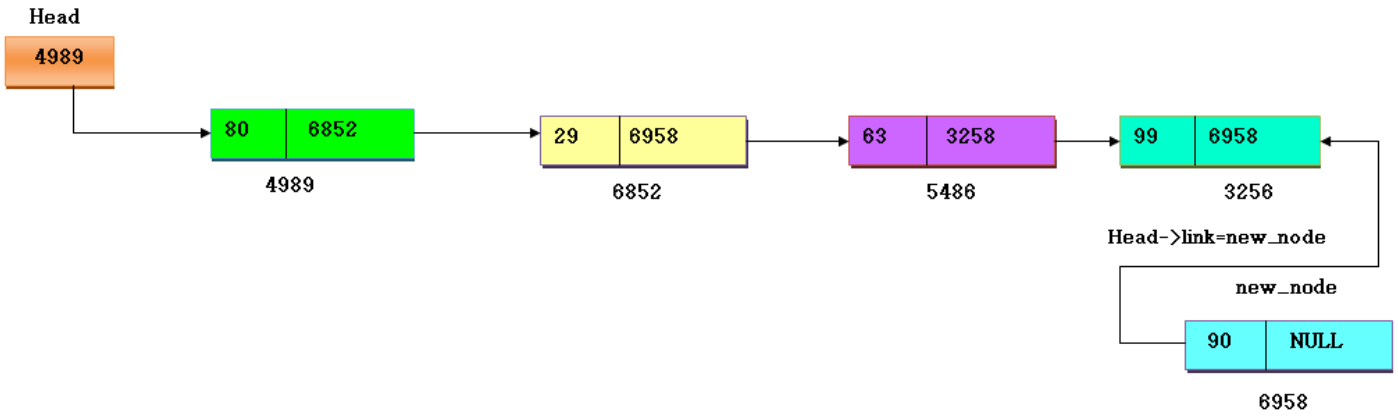
- **Circular Linked Lists:**

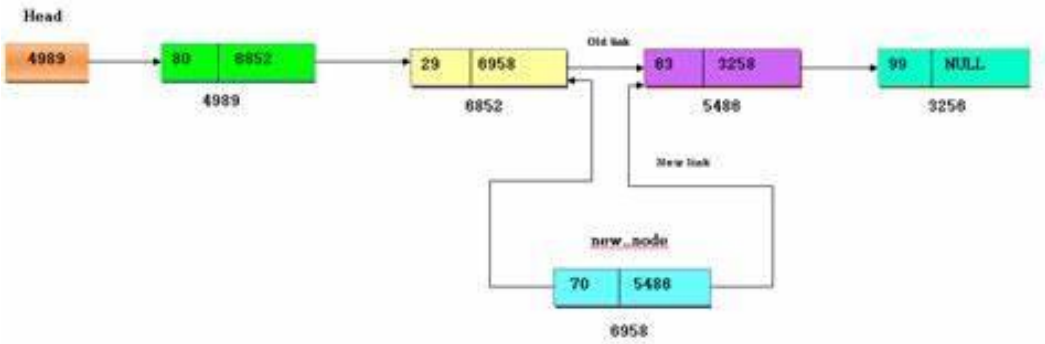
Operations on Linked List:

- Insert Node in List: (at the begin, end and in-between of the list)
- Delete Node in List: (from the begin, end and in between of the list)
- Search Node in list
- Print List





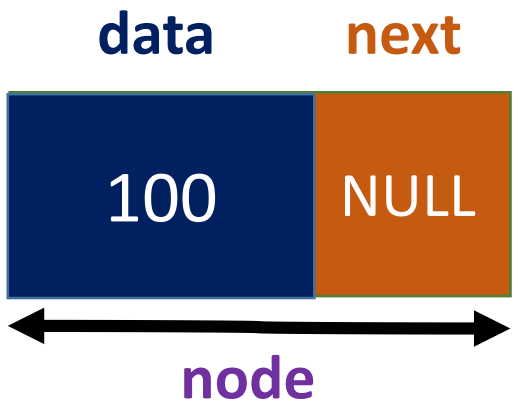




Operation: Create first Node

```
#include<stdio.h>
#include<conio.h>
struct Node* createNode(int);
struct Node
{           int data;
            struct Node* next;
};
struct Node *node,*list=NULL,*last=NULL;
```

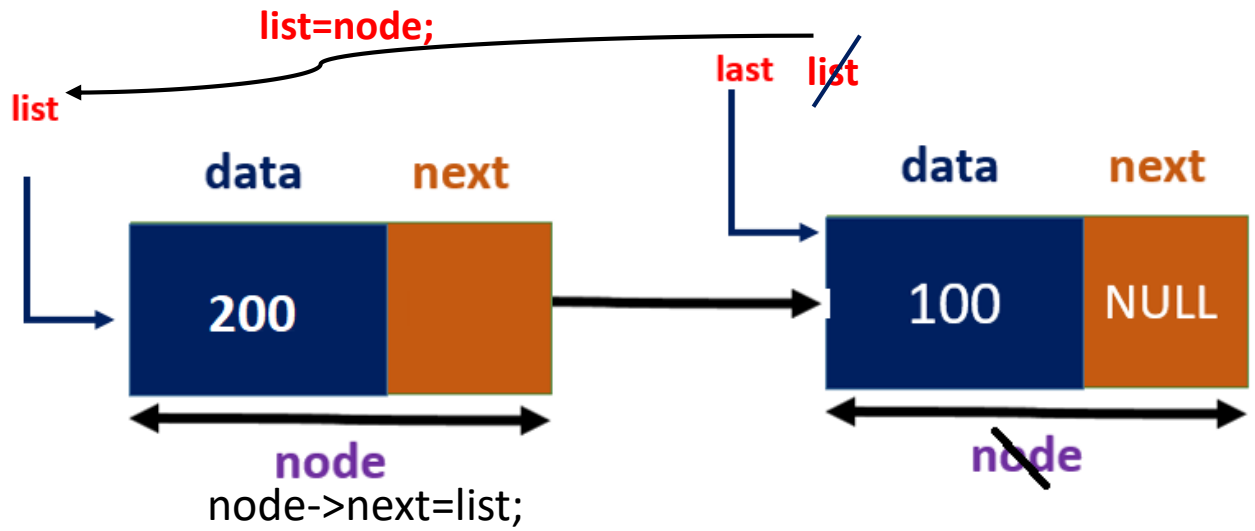
```
struct Node* createNode(int info)
{
            struct Node *node=(struct Node*) malloc(sizeof(struct Node));

            node->data=info;

            node->next=NULL;

            return node;

}
```



Operation: Add Node at the beginning of the List

```
#include<stdio.h>
#include<conio.h>
struct Node* createNode(int);
struct Node
{       int data;
        struct Node* next;
};
struct Node *node,*list=NULL,*last=NULL;
```

```
struct Node* createNode(int info)
{
        struct Node *node=(struct Node*) malloc(sizeof(struct Node));
        node->data=info;
        node->next=NULL;
        return node;
}
```

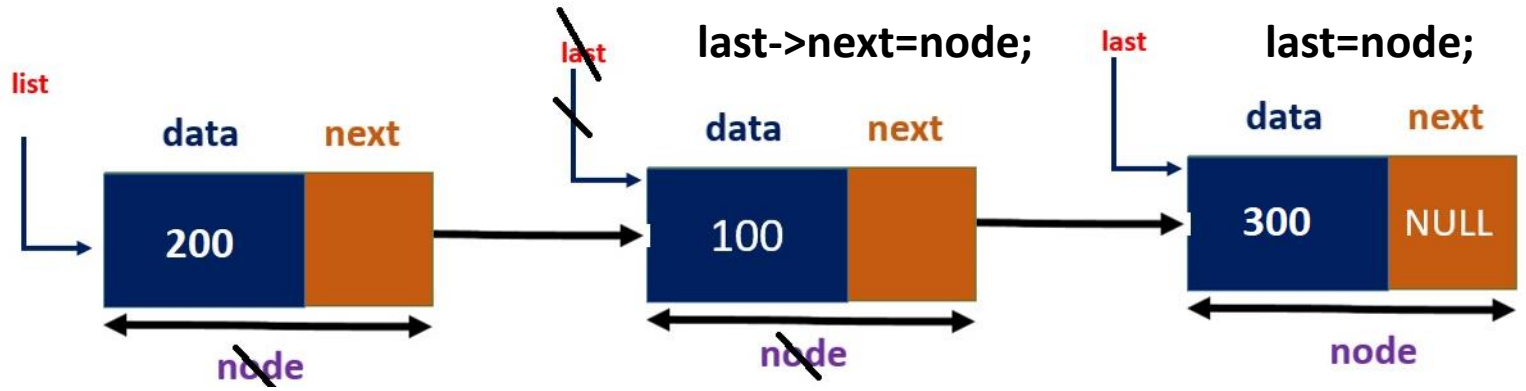



```
void addAtFront(int info)
{
        node=createNode(info);
        if(list==NULL)
         {    list=node;
              last=node;
         }
        else
        {    node->next=list;
             list=node;
        }
}
```

Operation: Add Node at the End of the List

```
#include<stdio.h>
#include<conio.h>
struct Node* createNode(int);
struct Node
{           int data;
            struct Node* next;
};
struct Node *node,*list=NULL,*last=NULL;
```

```
struct Node* createNode(int info)
{
            struct Node *node=(struct Node*) malloc(sizeof(struct Node));
            node->data=info;
            node->next=NULL;
            return node;
}
```

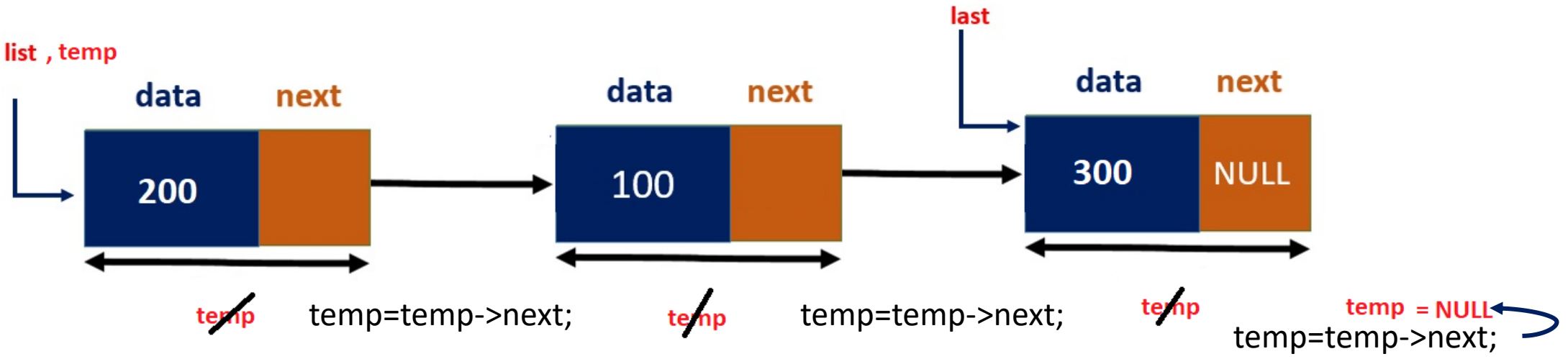



```
void addAtEnd(int info)
{    node=createNode(info);
     node->data=info;
     node->next=NULL;   */
     if(list==NULL)
     {  list=node;
        last=node;
     }
     else
     {   last->next=node;
        last=node;
     }
}
```

Operation: Display List

```
#include<stdio.h>
#include<conio.h>
struct Node* createNode(int);
struct Node
{           int data;
            struct Node* next;
};
struct Node *node,*list=NULL,*last=NULL, temp;
```

```
void displayList()
{
printf("\nTotal Nodes in List: %d\n",count);
            if(list==NULL)
            {
                        printf("\n List is Empty");
            }
            else
            {       temp=list;
                    while(temp!=NULL)
                    {
                        printf("%d-->",temp->data);
                        temp=temp->next;
                    }
}       } // end of display function
```

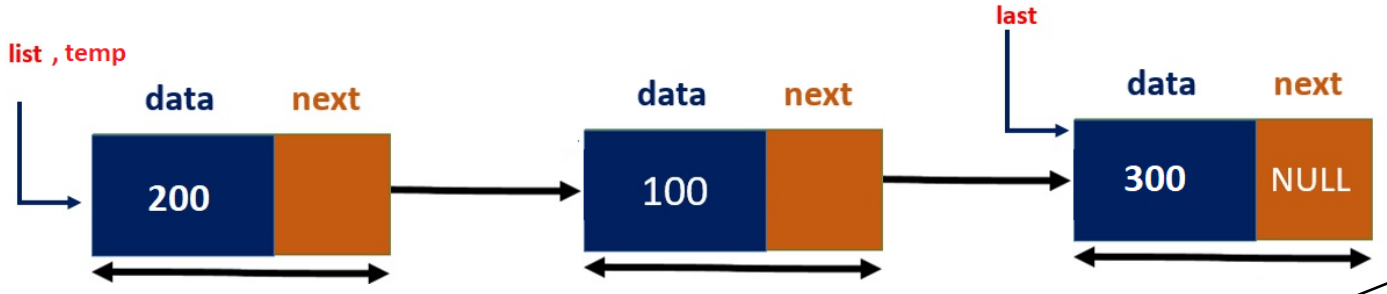


Operation: Delete Node at the Begin of the List

```
struct Node *node,*list=NULL,*last=NULL,
temp;
```

```
void deleteAtBegin()
{
    temp=list;
    if(list==NULL)
    {
            printf("\n List is Empty");
    }
     else
    {
       printf("\n%d Node Deleted",temp->data);
       list=list->next;
       free(temp);
    }
}
```

Present Linked List

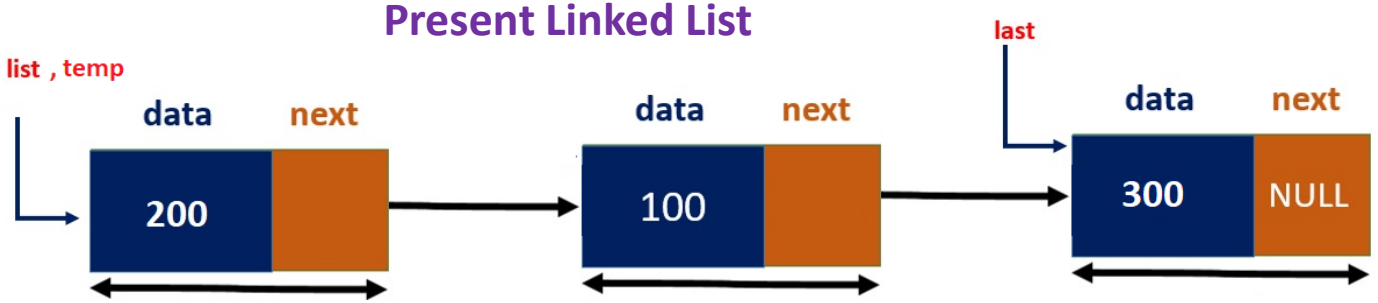


Linked List after Node Deletion →

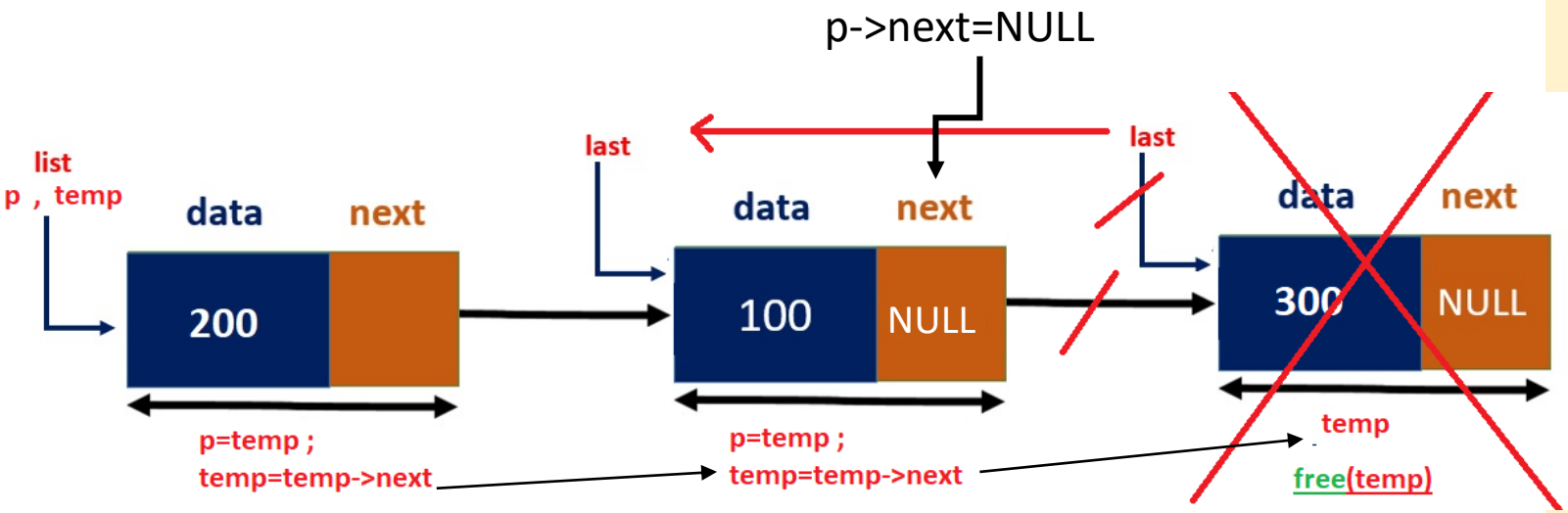
Operation: Delete Node at the End of the List

struct Node *node,*list=NULL,*last=NULL,
temp;

Present Linked List

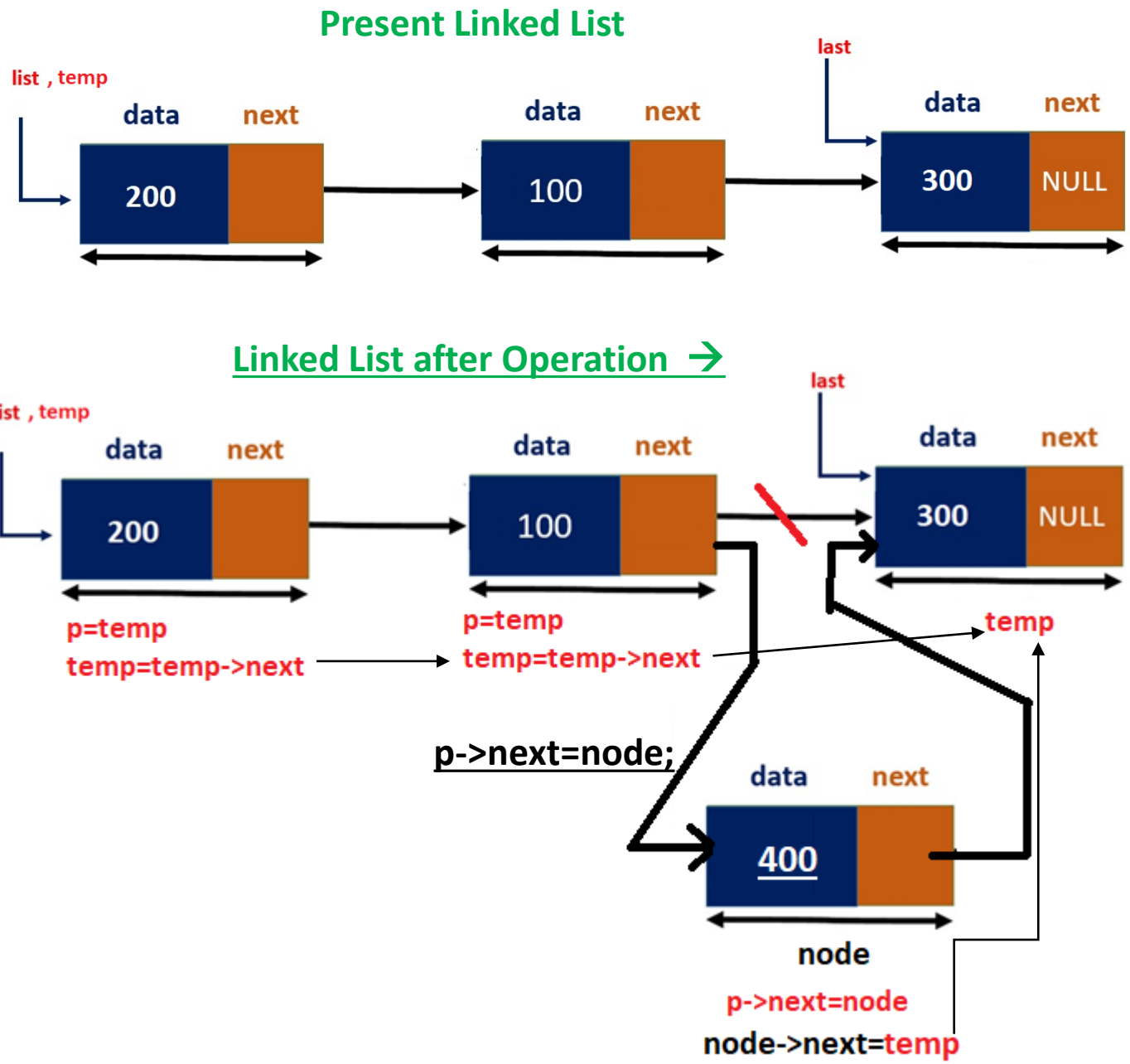


Linked List after Node Deletion →




```
void deleteAtEnd()
{
        if(list==NULL)
        {
                printf("\n List is Empty");

        }
        else if(list->next==NULL)
                list=NULL;
        else
        {       temp=list;
                p=list;
                while(temp->next!=NULL)
                {       p=temp;
                        temp=temp->next;
                }

                node->data=temp->data;
                free(temp);
                p->next=NULL;
                last=p;
        }}
```

Operation: Insert at position in the List




```
void insertAtposition(int info)
{
        int position,i;
        node=createNode(info);
        printf("\nEnter a position: ");
        scanf("%d",&position);  //I/P:position=3
        temp=list;
        if(position==1)
            addAtFront(info);
        else if(position==count+1)
            addAtEnd(info);
        else
        {
                for( i=1;i<position;i++)
                {

                        p=temp;
                        temp=temp->next;
                }
                p->next=node;
                node->next=temp;

        }
}
```

Exercise:

- Perform below Operations:

- **Search Node in the List & Print its location if node found**
- **Delete Node from position in the List**
- **Search & Delete Node from position in the List**
- **Merge 2 Lists**