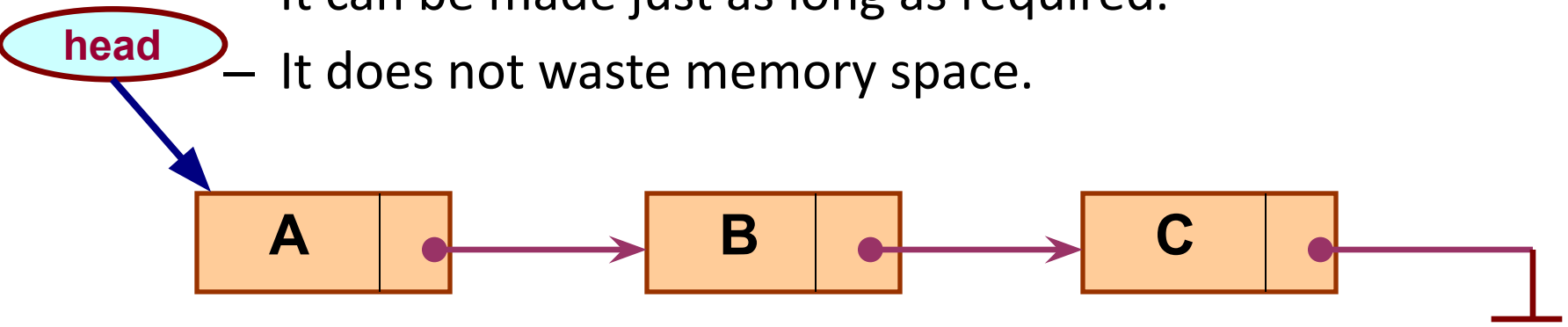
Linked List

Introduction

- A linked list is a data structure which can change during execution.
  - Successive elements 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.



- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

- Linked lists provide flexibility in allowing the items to be rearranged efficiently.
  - Insert an element.
  - Delete an element.

Types of Linked list

- 3 types
  - Singly Linked List(SLL)
  - Doubly Linked List(DLL)
  - Circular Linked List(CLL)

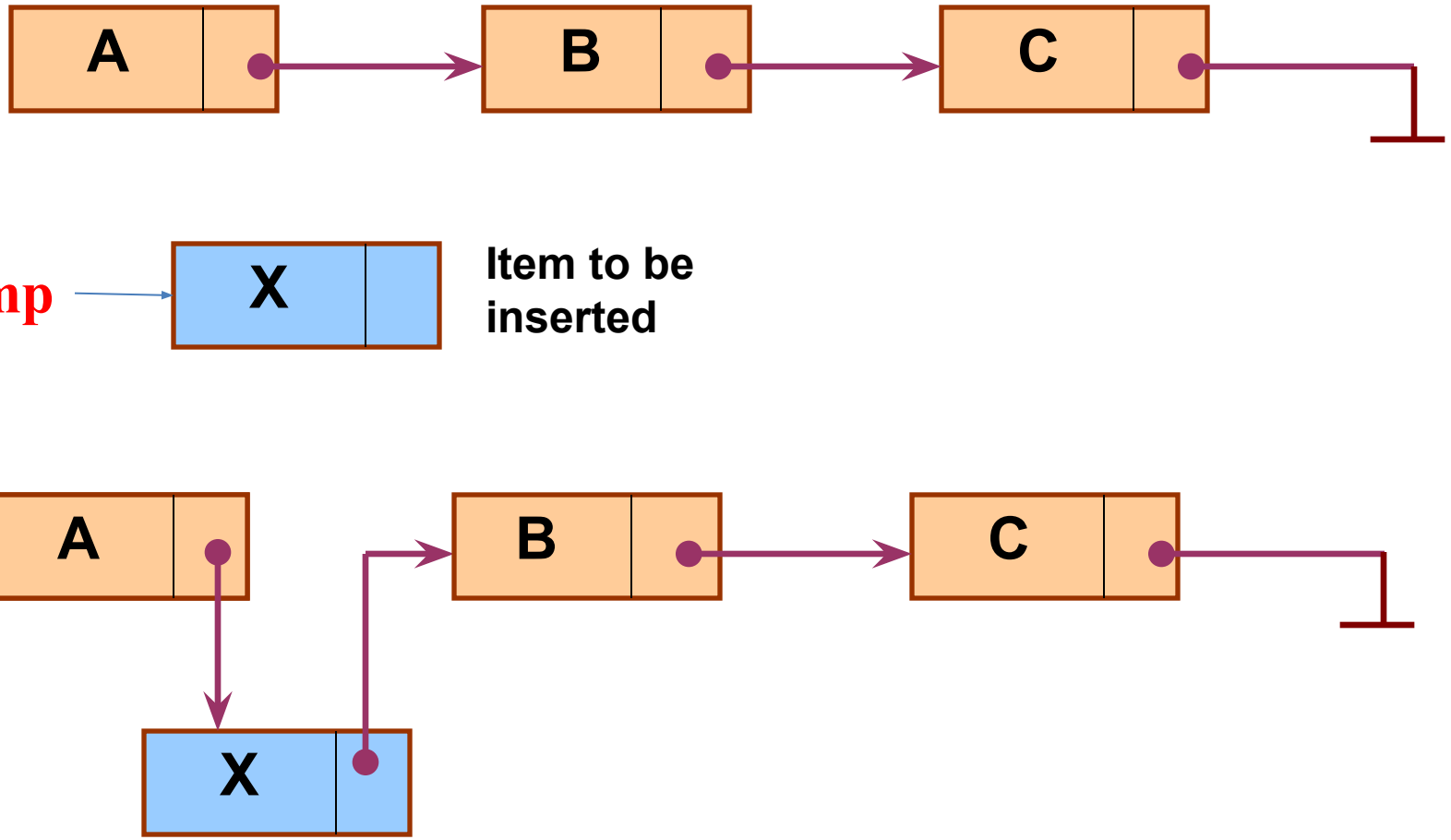
Basic Operations

- **Insertion** – Adds an element into the list
- **Deletion** – Deletes an element from the list
- **Display** – Displays the complete list
- **Search** – Searches an element using the given key

Node representation:

```
Struct Node                struct Node *node1=(struct Node          {
*)malloc(sizeof(struct Node));
    Int data;
    Node *next;
    };

    typedef struct node{    Node *node1=(Node
    {                           *)malloc(sizeof(Node));
        int data;
        struct node* next;
     }Node;
```

Illustration: Insertion



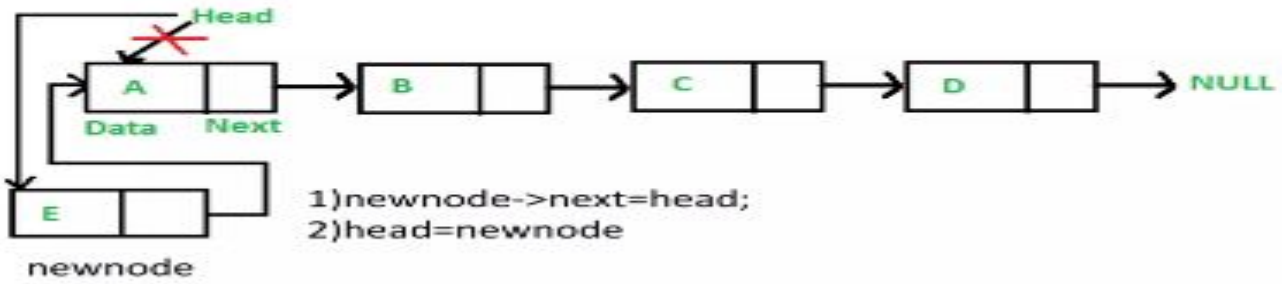
Pseudo-code for insertion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void insert(node *curr)
{
node * tmp;

tmp=(node *) malloc(sizeof(node));
tmp->next=curr->next;
curr->next=tmp;
}
```

Insertion in a SLL

- 3cases in this
  - Insertion at the beginning
  - Insertion at the end
  - Insertion after a particular node

Insert at Beginning of the list



Inserting At Beginning of the list

steps to insert a new node at beginning of the single linked list

Step 1 - Create a **newnode** with given value.

Step 2 - Check whether list is **Empty** (**head** == **NULL**)

Step 3 - If it is **Empty** then, set **newnode→next** = **NULL** and **head** = **new Node**.

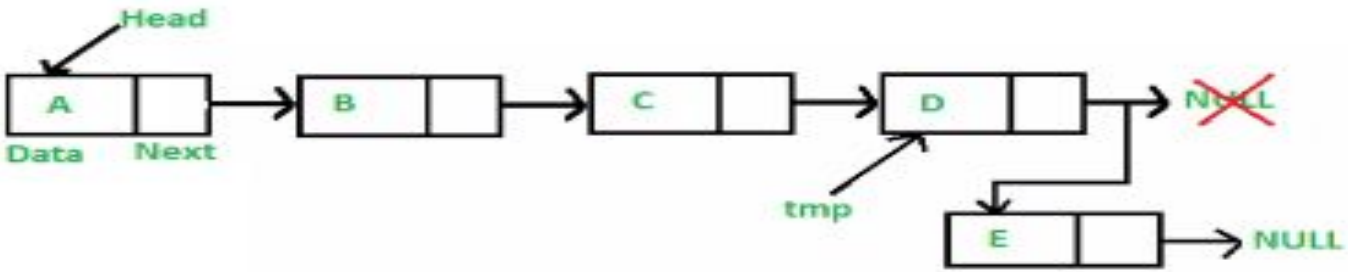
Step 4 - If it is **Not Empty** then, set **newnode→next** = **head** and **head** = **newnode**

```
void insertAtBeginning(int value)
{
struct Node *newnode;
newnode = (struct
Node*)malloc(sizeof(struct Node));
newnode->data = value;
 if(head == NULL)
 {
 newnode->next = NULL;
 head = newnode;
 }
else
 {
newnode->next = head;
head = newnode;
```

Inserting At Beginning of the list

```
void insertAtBeginning(int value)
{
struct Node *newnode;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
newnode->next = head;
head = newnode;
}
```

Insert at End of the list



1) Traverse –Now temp points to last node
2) temp ->next=newnode;

Insert at End of the list

steps to insert a new node at end of the single linked list

Step 1 - Create a **newnode** with given value and **newnode → next** as **NULL**.

Step 2 - Check whether list is **Empty** (**head == NULL**).

Step 3 - If it is **Empty** then, set **head** = **newnode**.

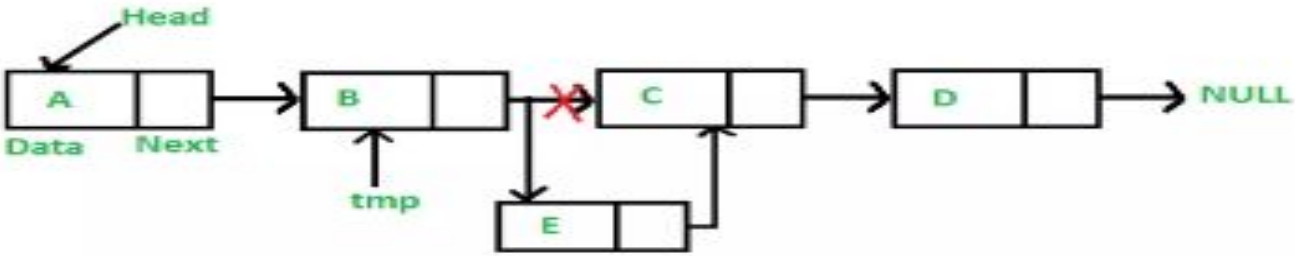
Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.

Step 5 - Keep moving the **temp** to its next node until it to the last node in the list (until **temp → next** is equal to **NULL**).

Step 6 - Set **temp → next** = **newnode**.

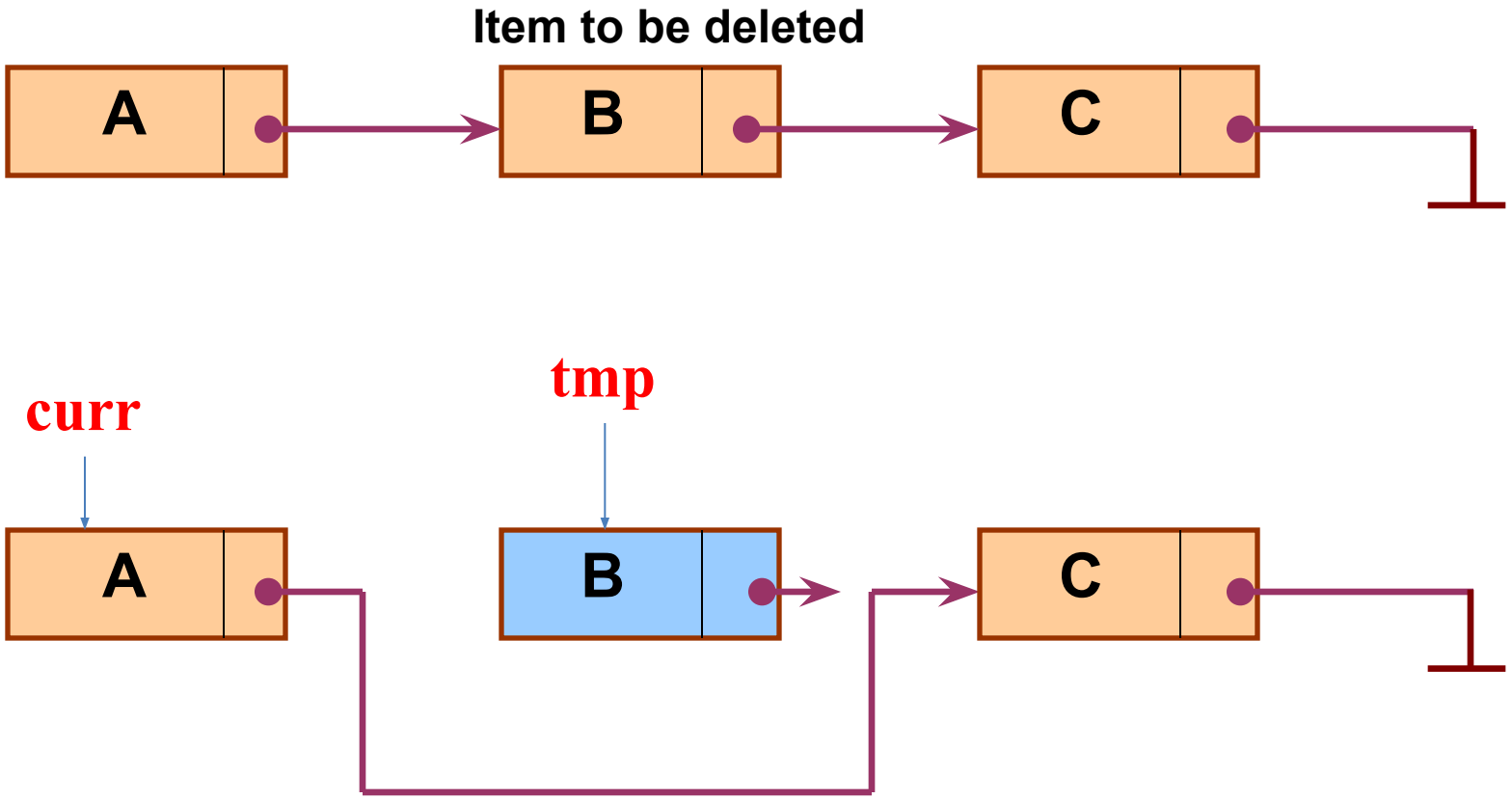
```
void insertAtEnd(int value)
{
struct Node *newnode;
newnode = (struct
Node*)malloc(sizeof(struct Node));
newnode->data = value;
newnode->next = NULL;
if(head == NULL)
        head = newnode;
else
{
        struct Node *temp = head;
        while(temp->next != NULL)
                temp = temp->next;
        temp->next = newnode;
}
}
```

Insert at a given position



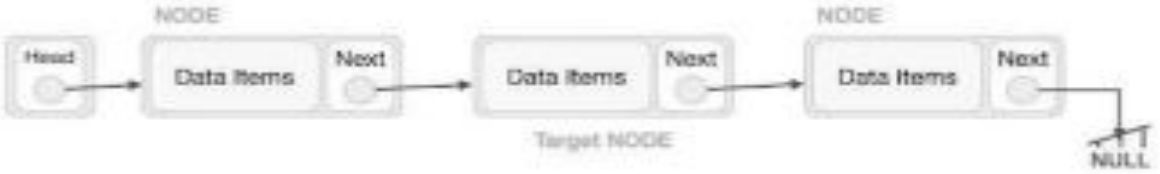
```
void insertatmiddle(int value, int pos)
{
struct Node *newNode,*temp;
Int i,pos;
newnode = (struct Node*)malloc(sizeof(struct Node));
newnode->data = value;
temp=head;
for(i=1;i<pos-1;i++)
{
     temp=temp->next;
}
If(temp==head)
{
    newnode->next=head;
    head=newnode;
}
else
{
   newnode->next=temp->next;
   temp->next=newnode;
}
```

Illustration: Deletion



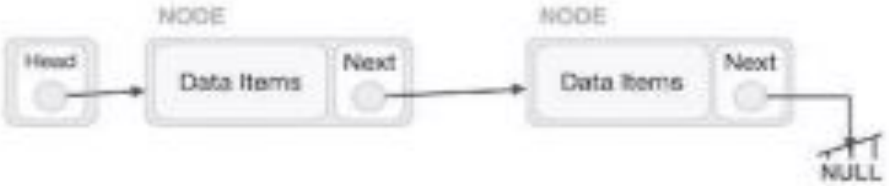
Deletion Operation

- locate the target node to be removed



- left (previous) node of the target node now should point to the next node of the target node
- LeftNode.next –> TargetNode.next;

- remove the target node is pointing at
  - TargetNode.next –> NULL;



Pseudo-code for deletion

```
typedef struct nd {
   struct item data;
   struct nd * next;
   } node;

void delete(node *curr)
{
node * tmp;
 tmp=curr->next;
curr->next=tmp->next;
free(tmp);
}
```

Delete at begin

```
Void deleteatbegin()
{
   struct node *temp;
   temp=head;
   printf("%d is deleted", temp->data);
   head=temp->next;
   free(temp);
}
```

Delete at last

```
void deleteatlast()
 {
  struct node *last, *secondlast;
  last=head;
  secondlast=head;
  while(last->next!=NULL)
  {
          secondlast= last;
          last=last->next;
  }
 If(last==head)
  {
  head=NULL;
  }
  else
  {
   secondlast->next=NULL;
   free(last);
    }
```

Displaying the linked list

```
Void display()
{
    struct node *temp;
   temp=head;
   while(temp!=NULL)
   {
        printf("%d", temp->data);
        temp=temp->next;
   }
}
```

Counting the no. of nodes in a LL

```
void count()
{
 int c=0;
 struct node *temp;
 temp=head;
 while(temp!=NULL)
    {
          c++;
          temp=temp->next;
    }
  printf("Number of nodes in the LL is %d", c);
}
```

Searching in a LL

```
void search()
{
   int key;
   printf("enter the element to search");
   scanf("%d",&key);
    temp = head;
   // Iterate till last element until key is not found
   while (temp != NULL && temp->data != key)
   {
            temp = temp->next;
   }
   if(temp->data==key)
   printf("element  found");
   else
   printf("element not found");
}
```

```
bool searchnode(struct node *head, int key)
{
	struct node *current;
	current = head;
	while(current != NULL)
	{
	 if(current->data == key)
	return true;
	current=current->next;
	}
	return false;
}
```

```
Void destroy(node *head)
{
node *temp;
      while(head!=null)
      {
            temp=head;
            head=head->next;
            free(temp);
      }
}
```

Delete first by key

```
void deleteFirstByKey()
{
   int key;
   struct node *tempprev;
   /* Check if head node contains key */
     printf("enter the element to delete");
     scanf("%d",&key);
   while (head != NULL && head->data == key)
   {
      // Get reference of head node
      temp = head;
      // Adjust head node link
      head = head->next;
      // Delete prev since it contains reference to head node
      free(temp);
   }
```

Delete first by key(contd...)

```
temp  = head;
   /* For each node in the list */
  while (temp != NULL)
  {
     // Current node contains key
     if (temp->data == key)
     {
        // Adjust links for previous node
        if (tempprev != NULL)
           tempprev->next = temp->next;
        // Delete current node
        free(temp);
     }
     tempprev = temp;
     temp = temp->next;
  }
}
```

In essence ...

- For insertion:
  - A record is created holding the new item.
  - The next pointer of the new record is set to link it to the item which is to follow it in the list.
  - The next pointer of the item which is to precede it must be modified to point to the new item.
- For deletion:
  - The next pointer of the item immediately preceding the one to be deleted is altered, and made to point to the item following the deleted item.

Array versus Linked Lists

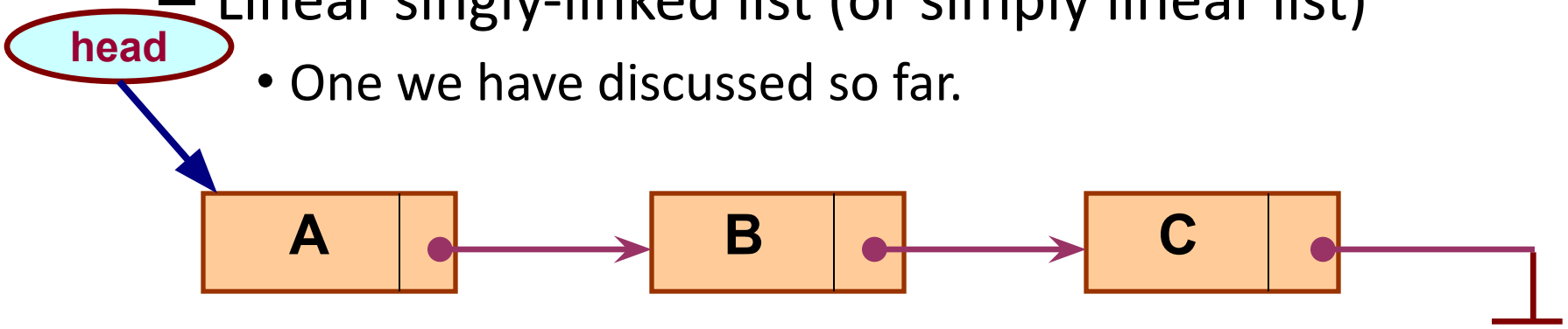
- Arrays are suitable for:
  - Inserting/deleting an element at the end.
  - Randomly accessing any element.
  - Searching the list for a particular value.
- Linked lists are suitable for:
  - Inserting an element.
  - Deleting an element.
  - Applications where sequential access is required.
  - In situations where the number of elements cannot be predicted beforehand.

Linked List vs Array

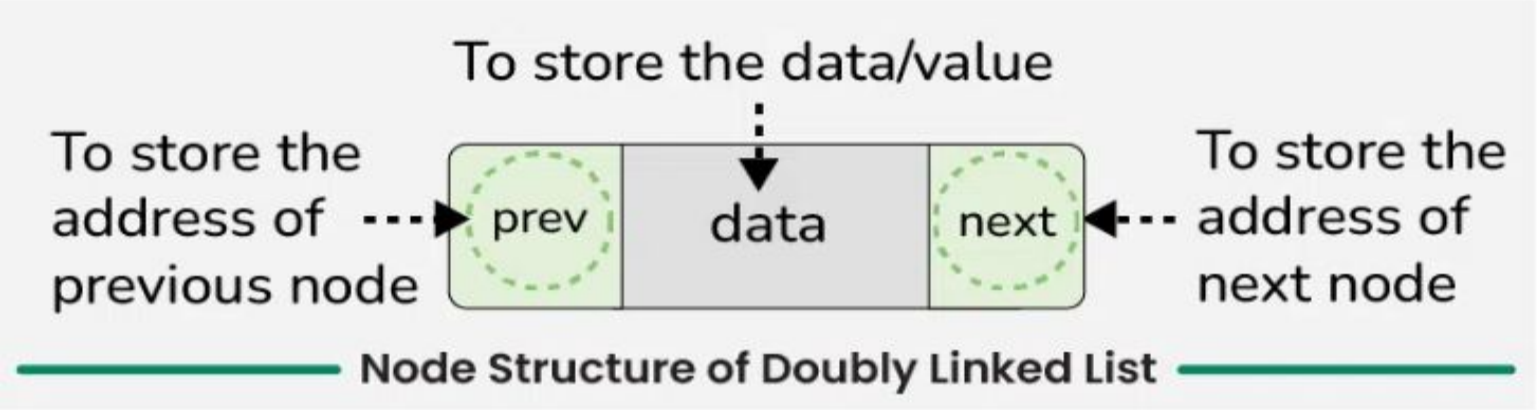
Array	Linked List
data structure that contains a collection of similar type of data elements	non-primitive data structure contains a collection of unordered linked elements known as nodes
Accessing an element in an array is fast	Accessing an element in an array is bit slower.
Operations in arrays consume a lot of time	operations in Linked lists is fast
Arrays are of fixed size.	Linked lists are dynamic and flexible and can expand and contract its size.
In array, memory is assigned during compile time	Linked list it is allocated during execution or runtime.
Elements are stored consecutively in arrays	Elements are stored randomly in Linked lists.
requirement of memory is less due to actual data being stored within the index in the array	more memory in Linked Lists due to storage of additional next and previous referencing elements.
memory utilization is inefficient in the array	memory utilization is efficient in the linked list.

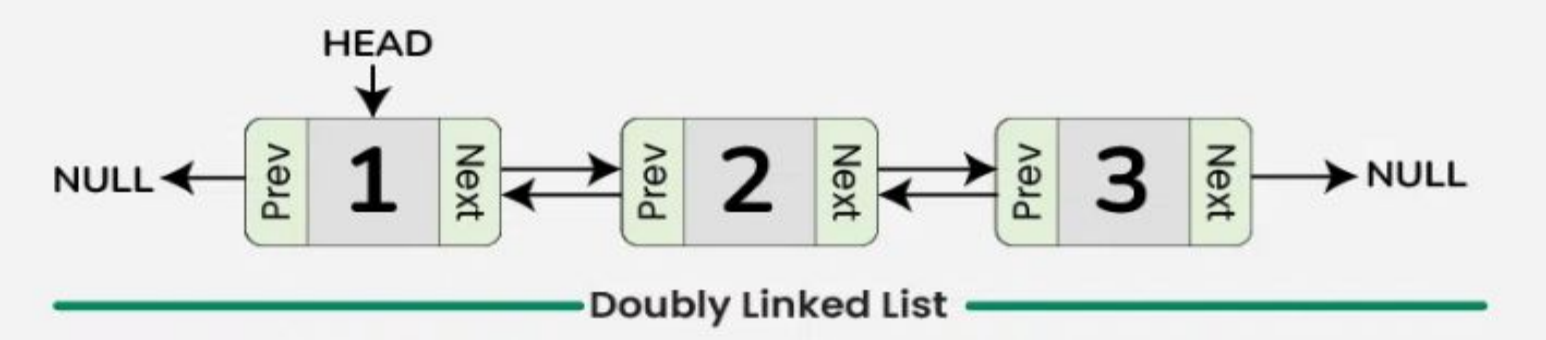
Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



Doubly linked list





Representation of Doubly Linked List in Data Structure

In a data structure, a doubly linked list is represented using nodes that have three fields:

1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)

```
struct Node {

    // To store the Value or data.
    int data;

    // Pointer to point the Previous Element
    Node* prev;

    // Pointer to point the Next Element
    Node* next;
};
```

```c
// Function to create a new node
struct Node *createNode(int new_data) {
    struct Node *new_node = (struct Node *)
    malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    new_node->prev = NULL;
    return new_node;
}
```

Operations on Doubly Linked List

- **Traversal in Doubly Linked List**
- **Searching in Doubly Linked List**
- **Finding Length of Doubly Linked List**
- **Insertion in Doubly Linked List:**
    - Insertion at the beginning of Doubly Linked List
    - Insertion at the end of the Doubly Linked List
    - Insertion at a specific position in Doubly Linked List
- **Deletion in Doubly Linked List:**
    - Deletion of a node at the beginning of Doubly Linked List
    - Deletion of a node at the end of Doubly Linked List
    - Deletion of a node at a specific position in Doubly Linked List

a. Forward Traversal:

- Initialize a pointer to the head of the linked list.
- While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the next node.

b. Backward Traversal:

- Initialize a pointer to the tail of the linked list.
- While the pointer is not null:
    - Visit the data at the current node.
    - Move the pointer to the previous node.

```c
// Function to traverse the doubly linked list
// in forward direction
void forwardTraversal(struct Node* head) {

    // Start traversal from the head of the list
    struct Node* curr = head;

    // Continue until the current node is not
    // null (end of list)
    while (curr != NULL) {

        // Output data of the current node
        printf("%d ", curr->data);

        // Move to the next node
        curr = curr->next;
    }

    // Print newline after traversal
    printf("\n");
}
```

```c
// Function to traverse the doubly linked list
// in backward direction
void backwardTraversal(struct Node* tail) {

    // Start traversal from the tail of the list
    struct Node* curr = tail;

    // Continue until the current node is not
    // null (end of list)
    while (curr != NULL) {

        // Output data of the current node
        printf("%d ", curr->data);

        // Move to the previous node
        curr = curr->prev;
    }

    // Print newline after traversal
    printf("\n");
}
```

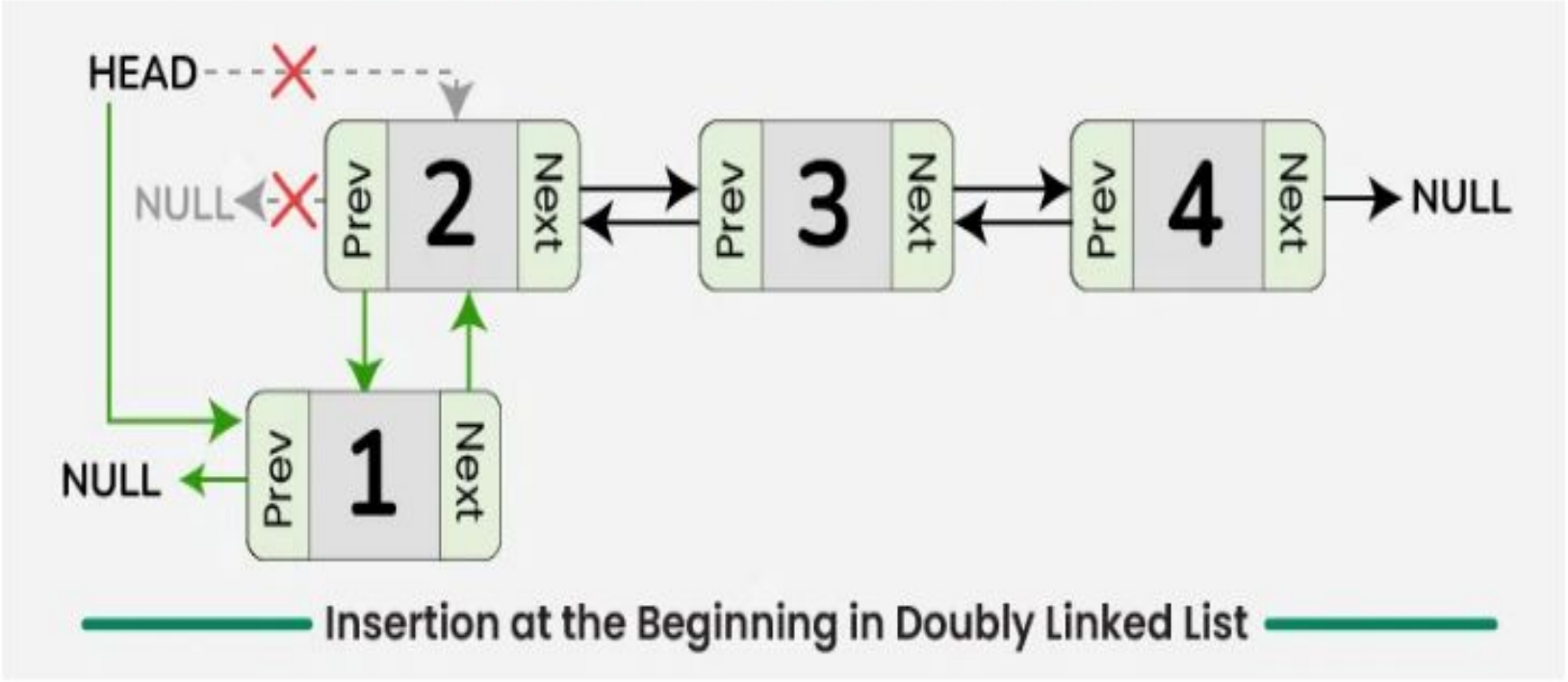
Finding Length of Doubly Linked List

To find the length of doubly list, we can use the following steps:

- Start at the head of the list.
- Traverse through the list, counting each node visited.
- Return the total count of nodes as the length of the list.

```c
// Function to find the length of a doubly linked list
int findLength(struct Node* head) {
    int count = 0;
    for (struct Node* cur = head; cur != NULL; cur = cur->next)
        count++;
    return count;
}
```

Insertion at the Beginning in Doubly Linked List

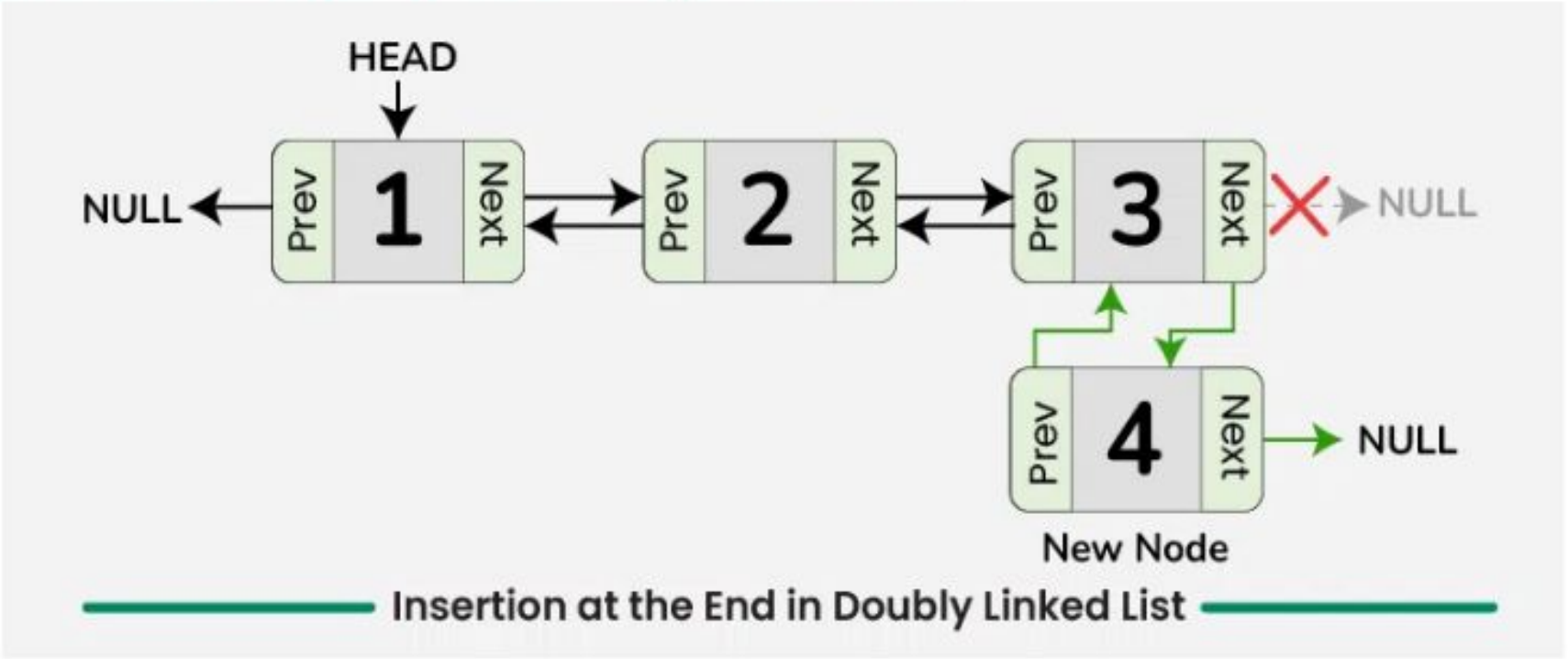


To insert a new node at the beginning of the doubly list, we can use the following steps:

- Create a new node, say **new_node** with the given data and set its previous pointer to null, **new_node->prev = NULL**.
- Set the next pointer of new_node to current head, **new_node->next = head.**
- If the linked list is not empty, update the previous pointer of the current head to new_node, **head->prev = new_node**.
- Return new_node as the head of the updated linked list.

```c
// Insert a node at the beginning
struct Node* insertBegin(struct Node* head, int data) {

    // Create a new node
    struct Node* new_node = createNode(data);

    // Make next of it as head
    new_node->next = head;

    // Set previous of head as new node
    if (head != NULL) {
        head->prev = new_node;
    }

    // Return new node as new head
    return new_node;
}
```

Insertion at the End of Doubly Linked List



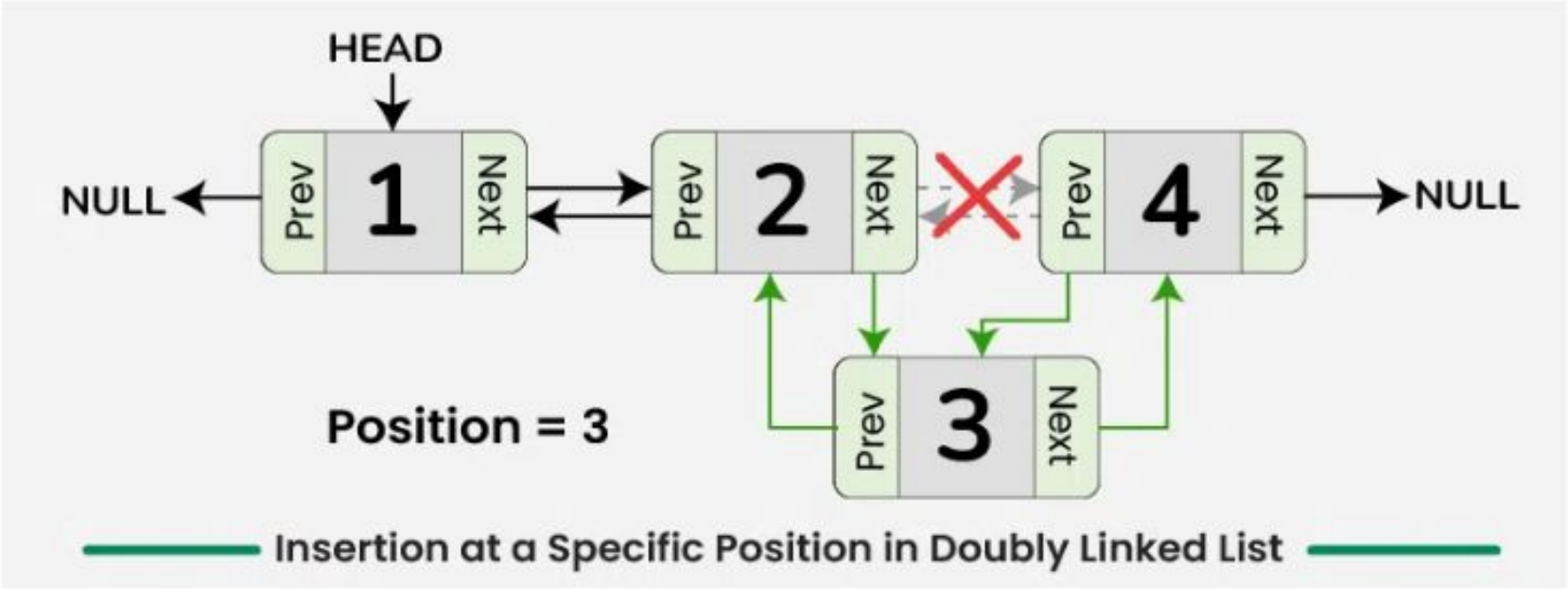
To insert a new node at the end of the doubly linked list, we can use the following steps:

- Allocate memory for a new node and assign the provided value to its data field.
- Initialize the next pointer of the new node to nullptr.
- If the list is empty:
    - Set the previous pointer of the new node to nullptr.
    - Update the head pointer to point to the new node.
- If the list is not empty:
    - Traverse the list starting from the head to reach the last node.
    - Set the next pointer of the last node to point to the new node.
    - Set the previous pointer of the new node to point to the last node.

```c
// Function to insert a new node at the end of the
//doubly linked list
struct Node* insertEnd(struct Node *head, int new_data) {
    struct Node *new_node = createNode(new_data);

    // If the linked list is empty, set the
    //new node as the head
    if (head == NULL) {
        head = new_node;
    } else {
        struct Node *curr = head;
        while (curr->next != NULL) {
            curr = curr->next;
        }

        // Set the next of last node to new node
        curr->next = new_node;
        // Set prev of new node to last node
        new_node->prev = curr;
    }

    return head;
}
```

Insertion at a Specific Position in Doubly Linked List

To insert a node at a specific Position in doubly linked list, we can use the following steps:



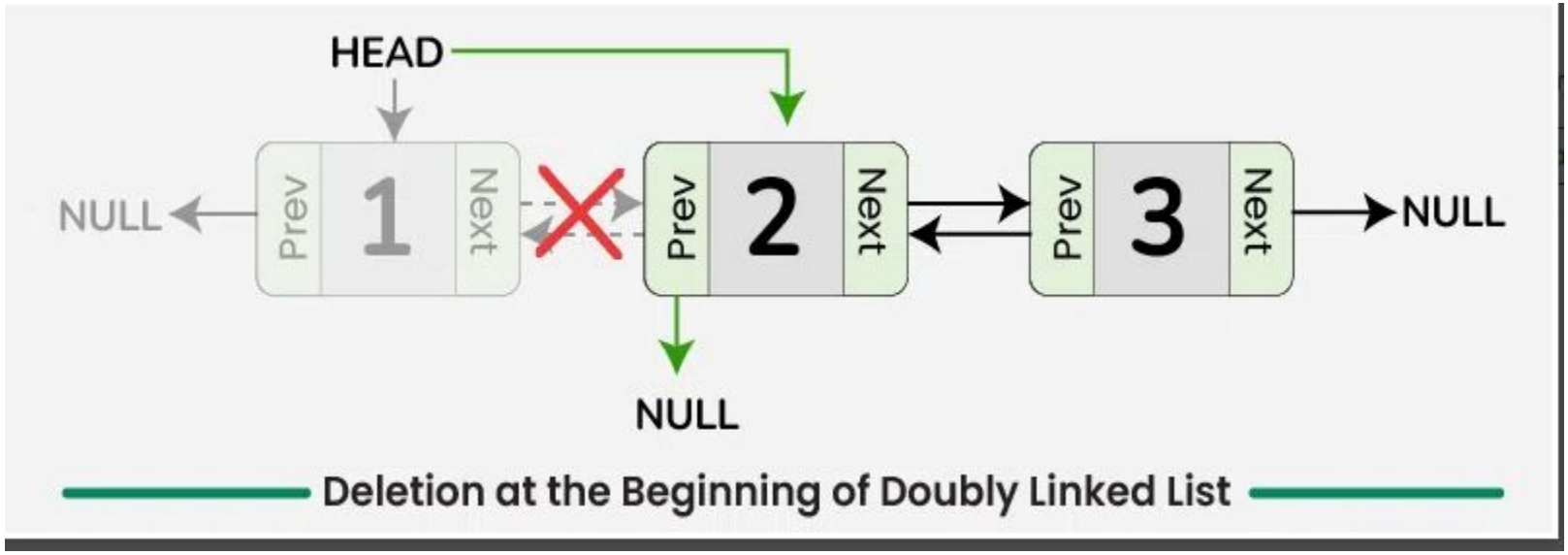
To insert a new node at a specific position,

- If position = 1, create a new node and make it the head of the linked list and return it.
- Otherwise, traverse the list to reach the node at position – 1, say **curr**.
- If the position is valid, create a new node with given data, say **new_node**.
- Update the next pointer of new node to the next of current node and prev pointer of new node to current node, **new_node->next = curr->next** and **new_node->prev = curr.**
- Similarly, update next pointer of current node to the new node, **curr->next = new_node**.
- If the new node is not the last node, update prev pointer of new node's next to the new node, **new_node->next->prev = new_node.**

```c
// Function to insert a new node at a given position
struct Node * insertAtPosition(struct Node * head, int pos, int new_data) {
    // Create a new node
    struct Node * new_node = createNode(new_data);

    // Insertion at the beginning
    if (pos == 1) {
        new_node -> next = head;

        // If the linked list is not empty, set the
      //prev of head to new node
        if (head != NULL) {
            head -> prev = new_node;
        }

        // Set the new node as the head of linked list
        head = new_node;
        return head;
    }

    struct Node * curr = head;

    // Traverse the list to find the node before the insertion point
    for (int i = 1; i < pos - 1 && curr != NULL; ++i) {
        curr = curr -> next;
    }
```

```
    // If the position is out of bounds
    if (curr == NULL) {
        printf("Position is out of bounds.\n");
        free(new_node);
        return head;
    }

    // Set the prev of new node to curr
    new_node -> prev = curr;

    // Set the next of new node to next of curr
    new_node -> next = curr -> next;

    // Update the next of current node to new node
    curr -> next = new_node;

    // If the new node is not the last node, update
    //the prev of next node to new node
    if (new_node -> next != NULL) {
        new_node -> next -> prev = new_node;
    }

    // Return the head of the doubly linked list
    return head;
}
```

Deletion at the Beginning of Doubly Linked List

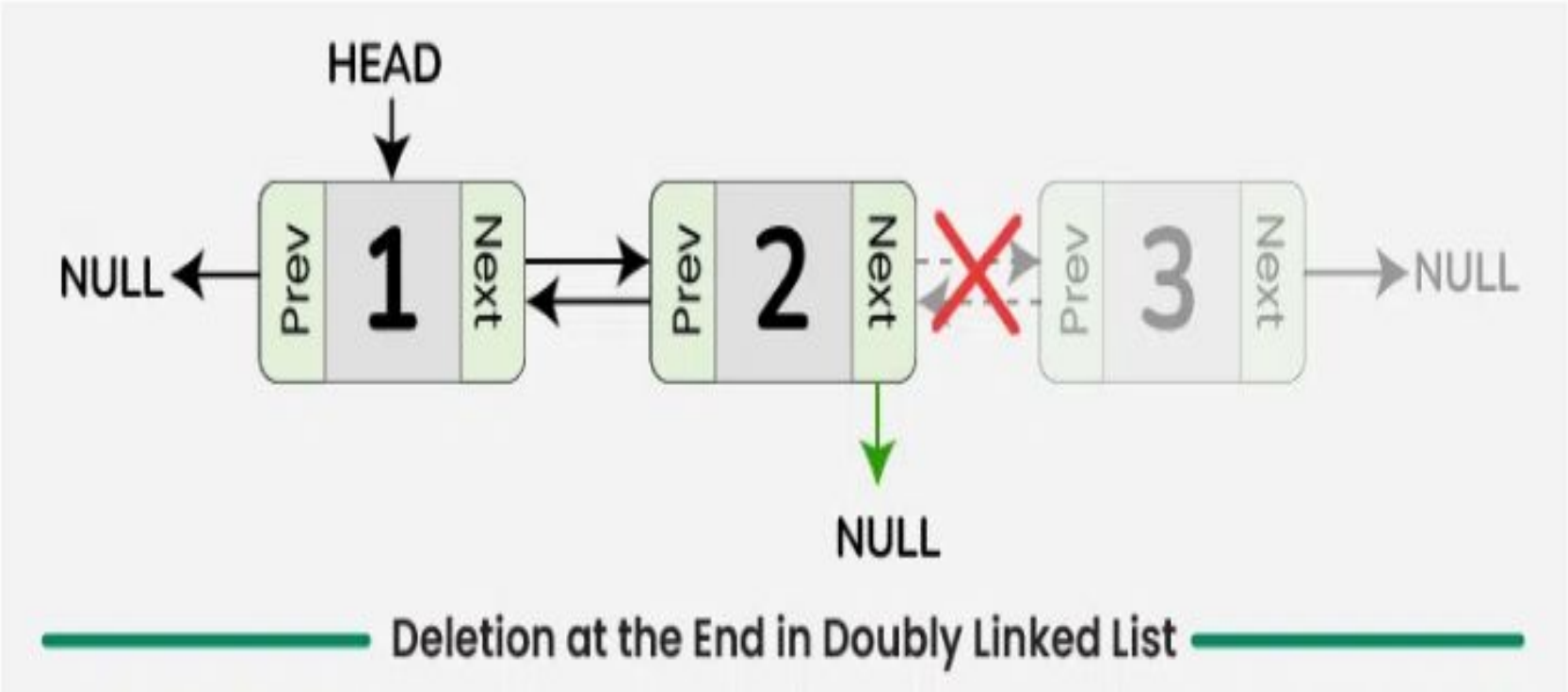


To delete a node at the beginning in doubly linked list, we can use the following steps:

- Check if the list is empty, there is nothing to delete. Return.
- Store the head pointer in a variable, say **temp**.
- Update the head of linked list to the node next to the current head, **head = head->next**.
- If the new head is not NULL, update the previous pointer of new head to NULL, **head->prev = NULL**.

```c
// Function to delete the first node (head) of the list
// and return the second node as the new head
struct Node *delHead(struct Node *head) {
    // If empty, return NULL
    if (head == NULL)
        return NULL;

    // Store in temp for deletion later
    struct Node *temp = head;

    // Move head to the next node
    head = head->next;

    // Set prev of the new head
    if (head != NULL)
        head->prev = NULL;

    // Free memory and return new head
    free(temp);
    return head;
}
```

Deletion at the End of Doubly Linked List



To delete a node at the end in doubly linked list, we can use the following steps:

- Check if the doubly linked list is empty. If it is empty, then there is nothing to delete.
- If the list is not empty, then move to the last node of the doubly linked list, say **curr**.
- Update the second-to-last node's next pointer to NULL, **curr->prev->next = NULL**.
- Free the memory allocated for the node that was deleted.

```c
// Function to delete the last node of the
//doubly linked list
struct Node* delLast(struct Node *head) {

    // Corner cases
    if (head == NULL)
        return NULL;
    if (head->next == NULL) {
        free(head);
        return NULL;
    }

    // Traverse to the last node
    struct Node *curr = head;
    while (curr->next != NULL)
        curr = curr->next;

    // Update the previous node's next pointer
    curr->prev->next = NULL;

    // Delete the last node
    free(curr);

    // Return the updated head
    return head;
}
```

Deletion at a Specific Position in Doubly Linked List

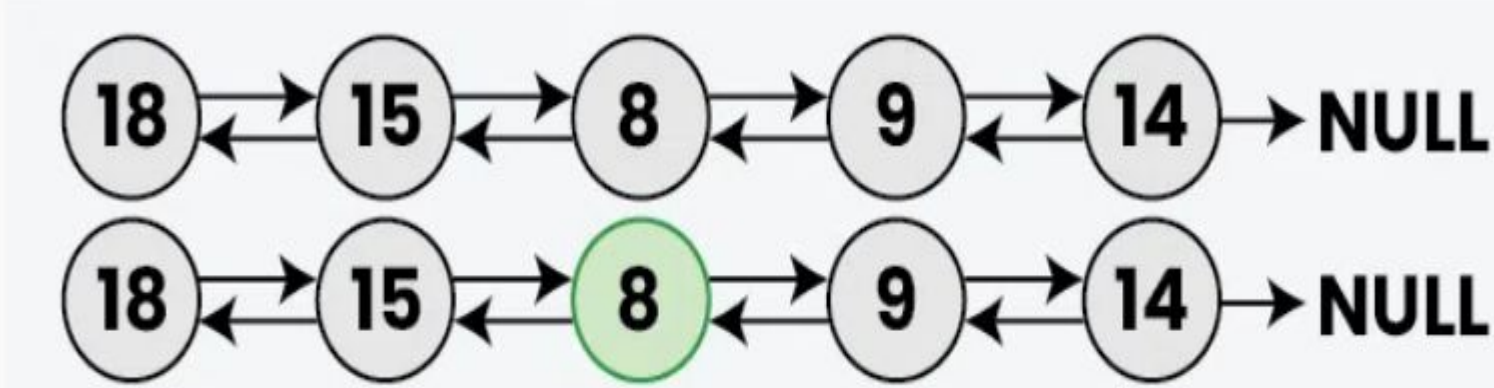
To delete a node at a specific position in doubly linked list, we can use the following steps:

- Traverse to the node at the specified position, say **curr**.
- If the position is valid, adjust the pointers to skip the node to be deleted.
    - If curr is not the head of the linked list, update the next pointer of the node before curr to point to the node after curr, **curr->prev->next = curr-next**.
    - If curr is not the last node of the linked list, update the previous pointer of the node after curr to the node before curr, **curr->next->prev = curr->prev**.
- Free the memory allocated for the deleted node.

```
// Function to delete a node at a specific
//position in the doubly linked list
struct Node * delPos(struct Node * head, int pos) {

    // If the list is empty
    if (head == NULL)
        return head;

    struct Node * curr = head;

    // Traverse to the node at the given position
    for (int i = 1; curr && i < pos; ++i) {
        curr = curr -> next;
    }

    // If the position is out of range
    if (curr == NULL)
        return head;

    // Update the previous node's next pointer
    if (curr -> prev)
        curr -> prev -> next = curr -> next;

    // Update the next node's prev pointer
    if (curr -> next)
        curr -> next -> prev = curr -> prev;

    // If the node to be deleted is the head node
    if (head == curr)
        head = curr -> next;

    // Deallocate memory for the deleted node
    free(curr);
    return head;
}
```

Search an element in a Doubly Linked List

Input: *Linked List = 18 <-> 15 <-> 8 <-> 9 <-> 14, x = 8*
Output: *3*



Explanation: *x= 8 is present at the 3rd node of the doubly linked list.*

Follow the steps below to solve the problem:

- Initialize a variable **pos** to store the position of the node containing data value **x** in the doubly linked list.
- Initialize a pointer **curr** to store the **head node** of the doubly linked list.
- Iterate over the linked list and for every node, check if **data value** of that node is equal to **x** or not.
    - If found return **pos.**
- return **-1** as the node with value x is not present.

```c
// Function to find the position of an integer
// in the doubly linked list
int search(struct Node* head, int x) {
    struct Node* curr = head;
    int pos = 0;

    // Traverse the doubly linked list
    while (curr != NULL && curr->data != x) {
        pos++;
        curr = curr->next;
    }

    // If the integer is not present in the list
    if (curr == NULL || curr->data != x)
        return -1;

    // If the integer is present in the list
    return (pos + 1);
}
```

Display Function:

```c
// Function to print the doubly linked list
void printList(struct Node * head) {
    struct Node * curr = head;
    while (curr != NULL) {
        printf("%d ", curr -> data);
        curr = curr -> next;
    }
    printf("\n");
}
```

Sorting the list:

```
// Function to sort the doubly linked list using
// insertion sort
struct Node* insertionSort(struct Node* head) {

    if (head == NULL) return head;
    struct Node* sorted = NULL;
    struct Node* curr = head;

    // Traverse the list to sort each element
    while (curr != NULL) {

        // Store the next node to process
        struct Node* next = curr->next;

        // Insert `curr` into the sorted part
        if (sorted == NULL || sorted->data >= curr->data) {
            curr->next = sorted;

            // If sorted is not empty, set its `prev`
            if (sorted != NULL) sorted->prev = curr;

            // Update sorted to the new head
            sorted = curr;
            sorted->prev = NULL;

        }
```

```
        else {

            // Pointer to traverse the sorted part
            struct Node* current_sorted = sorted;

            // Find the correct position to insert
            while (current_sorted->next != NULL &&
                   current_sorted->next->data < curr->data) {
                current_sorted = current_sorted->next;
            }

            // Insert `curr` after `current_sorted`
            curr->next = current_sorted->next;

            // Set `prev` if `curr` is not inserted
            // at the end
            if (current_sorted->next != NULL)
                current_sorted->next->prev = curr;

            // Set `next` of `current_sorted` to `curr`
            current_sorted->next = curr;
            curr->prev = current_sorted;
        }

        curr = next;
    }

    return sorted;
}
```

Advantages of Doubly Linked List

- **Efficient traversal in both directions:** Doubly linked lists allow for efficient traversal of the list in both directions, making it suitable for applications where frequent insertions and deletions are required.
- **Easy insertion and deletion of nodes:** The presence of pointers to both the previous and next nodes makes it easy to insert or delete nodes from the list, without having to traverse the entire list.
- **Can be used to implement a stack or queue:** Doubly linked lists can be used to implement both stacks and queues, which are common data structures used in programming.

Disadvantages of Doubly Linked List

- **More complex than singly linked lists:** Doubly linked lists are more complex than singly linked lists, as they require additional pointers for each node.
- **More memory overhead:** Doubly linked lists require more memory overhead than singly linked lists, as each node stores two pointers instead of one.

Applications of Doubly Linked List

- Implementation of undo and redo functionality in text editors.
- Cache implementation where quick insertion and deletion of elements are required.
- Browser history management to navigate back and forth between visited pages.
- Music player applications to manage playlists and navigate through songs efficiently.
- Implementing data structures like Deque (double-ended queue) for efficient insertion and deletion at both ends.

– Circular linked list

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