Unit 3: Linked Lists

Unit 5

Sandesh Poudel

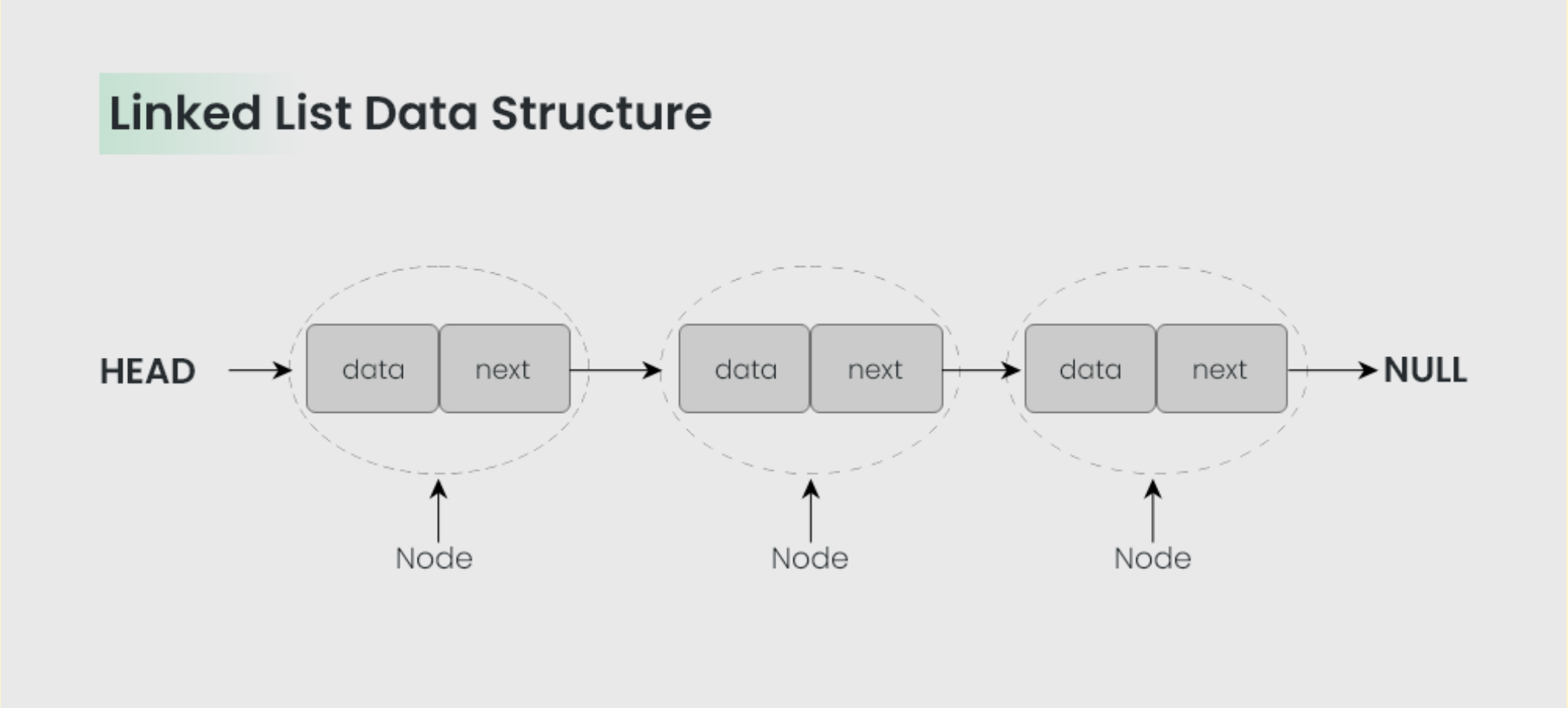
(MCP/MSCIT)

(Birendra Multiple Campus)

Linked list

- A **linked list** is a fundamental data structure in computer science. It consists of nodes where each node contains data and a reference (link) to the next node in the sequence. This allows for dynamic memory allocation and efficient insertion and deletion operations compared to arrays.

- A linked list is a linear data structure consisting of a sequence of elements, called nodes, where each node contains a data field and a reference (or pointer) to the next node in the sequence.
- Unlike arrays, which have a fixed size and contiguous memory allocation, linked lists allow dynamic memory allocation and can easily grow or shrink in size.



- A linked list starts with a **head** node which points to the first node. Every node consists of data which holds the actual data (value) associated with the node and a next pointer which holds the memory address of the next node in the linked list. The last node is called the tail node in the list which points to **null** indicating the end of the list.

Array List vs. Linked List: A Comparison

- Both array lists and linked lists are fundamental data structures used to store and manage collections of data, but they have different strengths and weaknesses depending on your specific needs.
- Here's a comparison to help you decide which one is better for your situation:

Characteristic	Array List	Linked List
Storage	Contiguous memory allocation	Non-contiguous memory allocation (nodes linked by pointers)
Access	Efficient random access by index (O(1))	Inefficient random access (requires traversing to the desired element)
Insertion/ Deletion	Inefficient (requires shifting elements)	Efficient (only modify links)
Memory Overhead	Lower (no extra pointers in nodes)	Higher (each node has pointers)
Cache Friendliness	More cache-friendly (data stored contiguously)	Less cache-friendly (data scattered in memory)
Resizing	Can be resized dynamically, but may involve copying data	Can be resized dynamically without data movement
Common Operations	Random access, retrieving elements by index	Adding/removing elements from beginning or end, iterating through the list
Applications	Storing static data, implementing fixed-size buffers, random access operations	Frequent insertions/deletions, representing graphs, dynamic memory allocation

Linked Lists vs Arrays

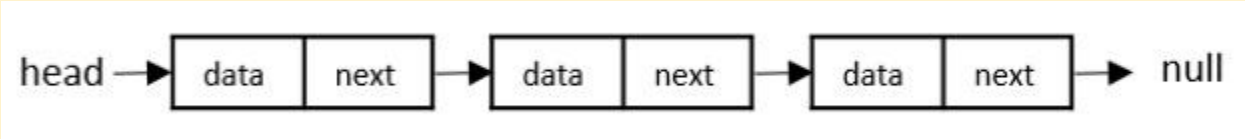
- In case of arrays, the size is given at the time of creation and so arrays are of fixed lenght where as Linked lists are dynamic in size and any number of nodes can be added in the linked lists dynamically.
- An array can accommodate similar types of data types where as linked lists can store various nodes of different data types.

Types of Linked Lists:

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

Types of Linked Lists:

1. **Singly Linked List**



In a singly linked list, each node points to the next node in the sequence, forming a unidirectional chain.
The last node in the list points to NULL.
Example:

[Data1|Next] -> [Data2|Next] -> [Data3|Next] -> ... -> [DataN|NULL]

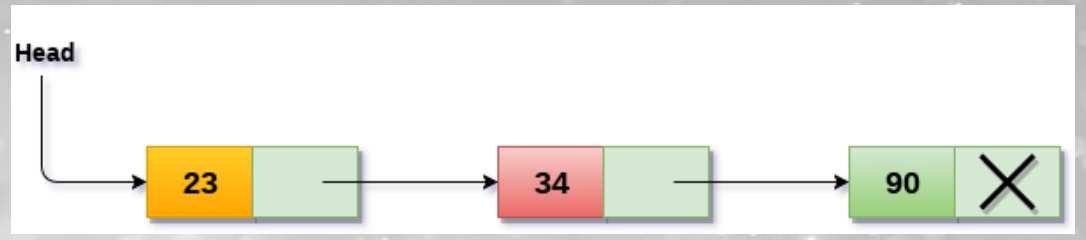
Operations in Singly Linked List:

Inserting – Adds an element to the list.

Display – Displays the complete list.

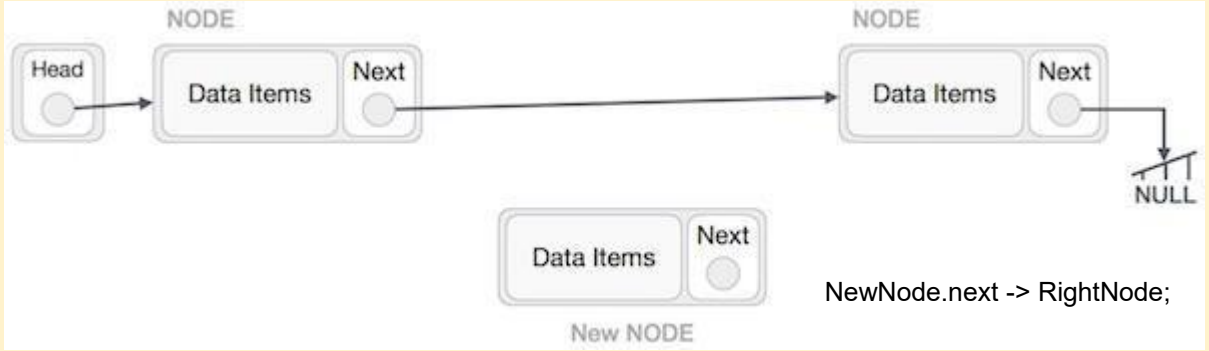
Search – Searches an element using the given key.

Delete – Deletes an element using the given key.



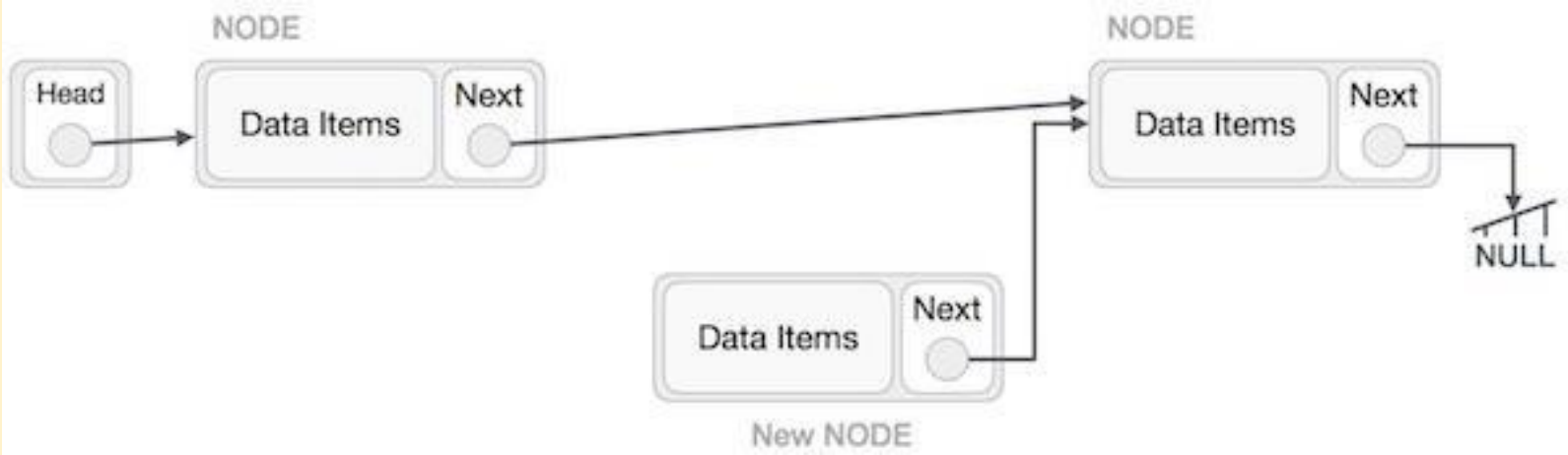
Insertion Operation

- Adding a new node in linked list is a more than one step activity.
- First, create a node using the same structure and find the location where it has to be inserted.

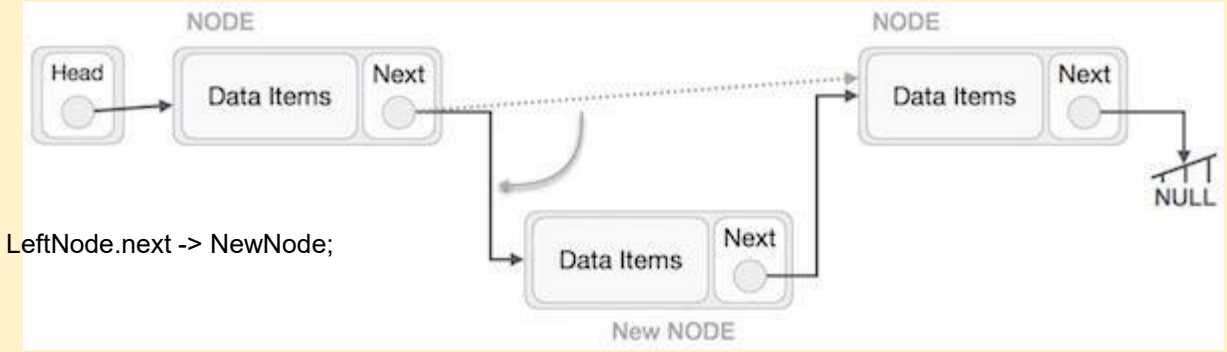


Imagine that we are inserting a node B (NewNode), between A (LeftNode) and C (RightNode). Then point B.next to C

It should look like this –



- Now, the next node at the left should point to the new node.



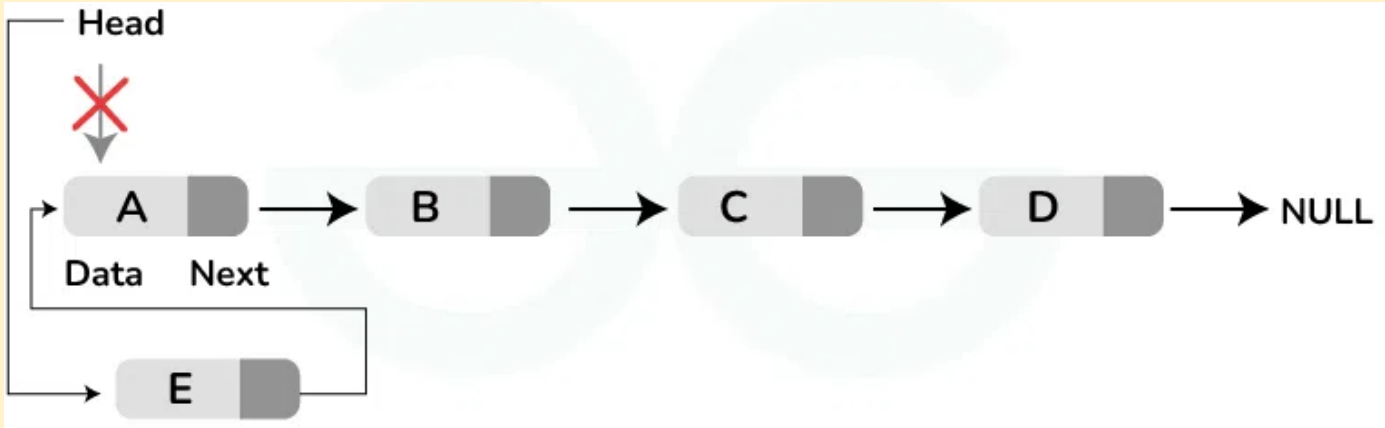
This will put the new node in the middle of the two.

Insertion in linked list can be done in three different ways.

- **Insertion at Beginning**
- **Insertion at Ending**
- **Insertion at a Given Position**

Insertion at Beginning (Algorithm)

1. START
2. Create a new node using malloc function
3. Read and Assign data into the new node.
4. Remove the head from the original first node of Linked List
5. Make the new node as the Head of the Linked List.
6. END



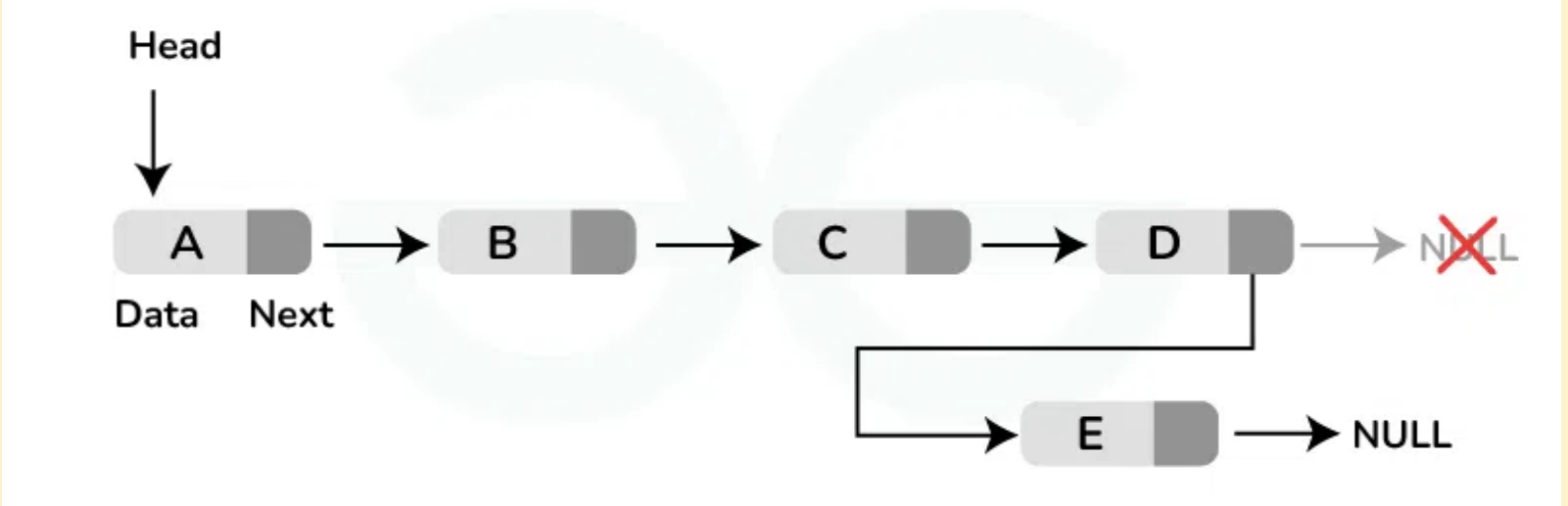
```
// Insert at begin of LinkedList
struct node
{
int data;
node *next;
node(int x)
{
	data=x; // assigning value to data variable
	next=NULL; // initially next is NULL
}
};
node *insertbegin(node *head, int data)
{
node *temp = new node(data);
temp -> next = head;
return temp;
}
```

Full program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
// display the list
void printList(){
   struct node *p = head;
   printf("\n[");
   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}
//insertion at the beginning
void insertatbegin(int data)
   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

// point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   insertatbegin(44);
   insertatbegin(50);
   printf("Linked List: ");

   // print list
   printList();
}
```

Insertion at Ending (Algorithm)

1. START
2. Create a new node and assign the data using malloc function
3. Find the last node
4. Point the last node to new node
5. Set the next of new node to null
5. END



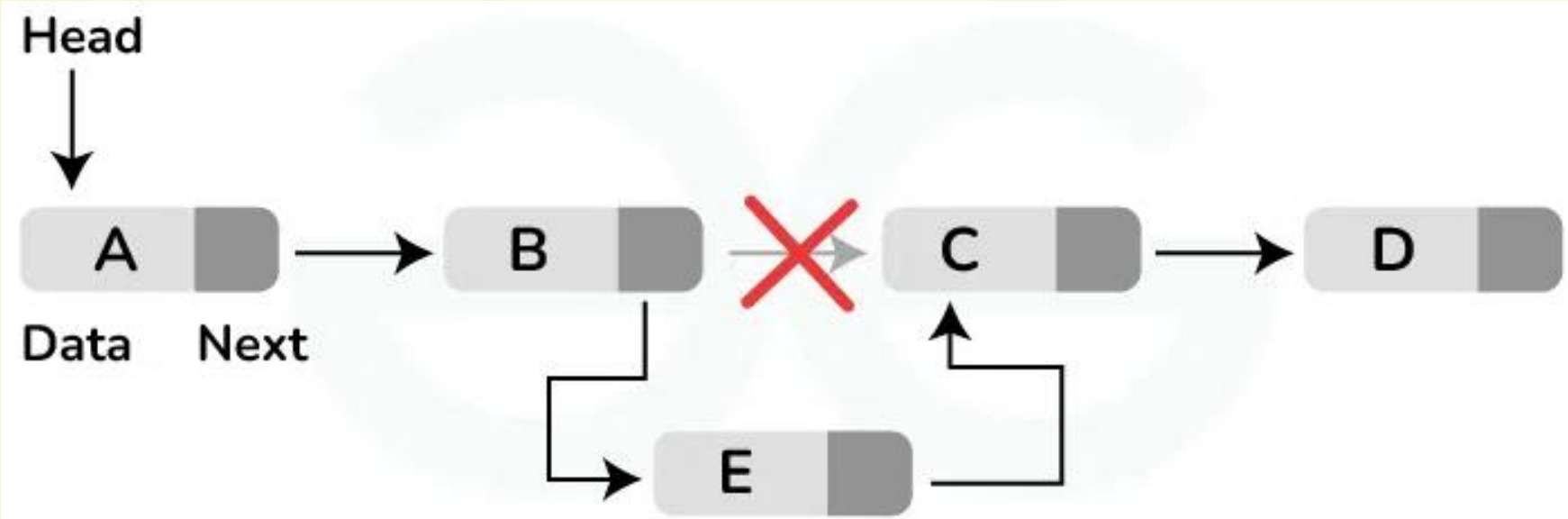
```
// insert at end of LinkedList
node *insertend(node *head, int data)
{
node *temp = new node(data);

if(head == NULL)
{
	return temp;
}

node *curr = head;

while(curr->next != NULL)
{
	curr = curr -> next;
}

curr -> next = temp;
temp -> next = NULL;
return head;
}
```

Full Program

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

struct Node {
    int data;
    struct Node* next;
};

void append(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    struct Node *last = *head_ref;
    new_node->data = new_data;
    new_node->next = NULL;

    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = new_node;
}

void printList(struct Node *node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}

int main() {
    struct Node* head = NULL;
    append(&head, 1);
    append(&head, 2);
    append(&head, 3);
    printList(head); // Output: 1 2 3
    return 0;
}
```

Insertion at a Given Position

1. START
2. Create a new node using malloc function and assign data to it
3. Iterate until the node at position is found
4. Point first to new first node
5. END



```
node *insertpos(node *head, int position, int data)          // Insert at nth position of LinkedList
{
node *temp = new node(data);
if(position == 1)
{
        temp -> next = head;
        return temp;
}
node *curr = head;
for(int i = 1;i <= position-2 && curr != NULL; i++)
{
        curr = curr -> next;
}
if(curr == NULL)
{
        return head;
}
//Insertion and returning head
temp -> next = curr -> next;
curr -> next = temp;
return head;
}
```

Full Program

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

struct Node {
    int data;
    struct Node* next;
};

void insertAfter(struct Node* prev_node, int new_data) {
    if (prev_node == NULL) {
        printf("The given previous node cannot be NULL\n");
        return;
    }

    struct Node* new_node = (struct Node*) malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = prev_node->next;
    prev_node->next = new_node;
}

void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
}
int main() {
    struct Node* head = NULL;
    struct Node* second = NULL;

    head = (struct Node*)malloc(sizeof(struct Node));
    second = (struct Node*)malloc(sizeof(struct Node));

    head->data = 1;
    head->next = second;

    second->data = 2;
    second->next = NULL;

    insertAfter(head, 3);
    printList(head); // Output: 1 3 2
    return 0;
}
```

Single Linked List - Deletion Operation

- Deletion is also a more than one step process. First, locate the target node to be removed, by using searching algorithms.

The left (previous) node of the target node now should point to the next node of the target node

LeftNode.next -> TargetNode.next;

- This will remove the link that was pointing to the target node. Now, using the following code, we will remove what the target node is pointing at.

- TargetNode.next -> NULL;

If we need to use the target (deleted) node, We can keep that in memory otherwise we can simply deallocate memory and wipe off the target node completely.

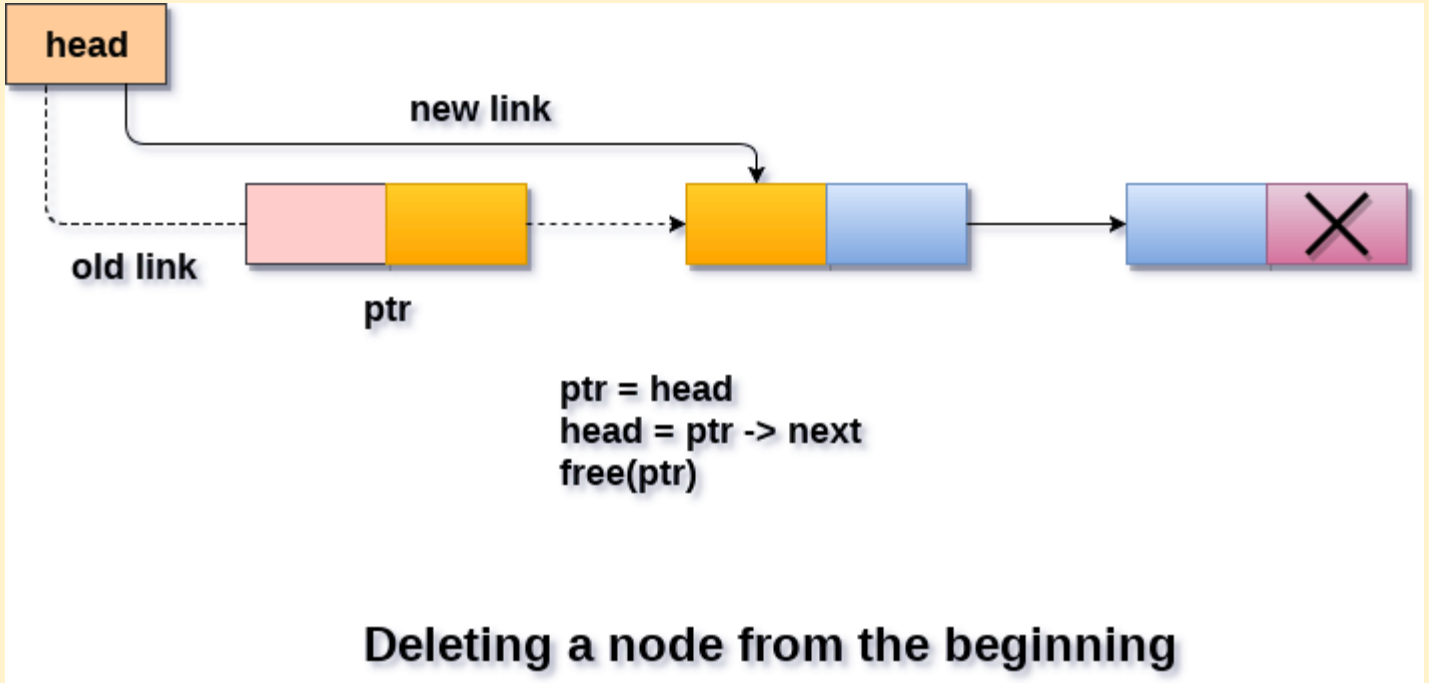
```c
#include <stdio.h>
#include <stdlib.h>

struct Node {
    int data;
    struct Node* next;
};

void insertAtBeginning(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

void insertAtEnd(struct Node** head_ref, int new_data) {
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    new_node->next = NULL;
    if (*head_ref == NULL) {
        *head_ref = new_node;
        return;
    }
    struct Node* last = *head_ref;
    while (last->next != NULL)
        last = last->next;
    last->next = new_node;
}

void insertAtPosition(struct Node** head_ref, int new_data, int position) {
    if (position < 1) {
        printf("Invalid position.\n");
        return;
    }
    if (position == 1 || *head_ref == NULL) {
        insertAtBeginning(head_ref, new_data);
        return;
    }
    struct Node* new_node = (struct Node*)malloc(sizeof(struct Node));
    new_node->data = new_data;
    struct Node* current = *head_ref;
    for (int i = 1; i < position - 1 && current != NULL; i++)
        current = current->next;
    if (current == NULL) {
        printf("Position out of bounds.\n");
        return;
    }
    new_node->next = current->next;
    current->next = new_node;
}

void printList(struct Node* node) {
    while (node != NULL) {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;
    int choice, data, position;

    while (1) {
        printf("\nMenu:\n");
        printf("1. Insert at Beginning\n");
        printf("2. Insert at End\n");
        printf("3. Insert at Position\n");
        printf("4. Print List\n");
        printf("5. Exit\n");
        printf("Enter your choice: ");
        scanf("%d", &choice);

        switch (choice) {
            case 1:
                printf("Enter data to insert at beginning: ");
                scanf("%d", &data);
                insertAtBeginning(&head, data);
                printf("Inserted %d at the beginning.\n", data);
                break;
            case 2:
                printf("Enter data to insert at end: ");
                scanf("%d", &data);
                insertAtEnd(&head, data);
                printf("Inserted %d at the end.\n", data);
                break;
            case 3:
                printf("Enter data to insert: ");
                scanf("%d", &data);
                printf("Enter position: ");
                scanf("%d", &position);
                insertAtPosition(&head, data, position);
                printf("Inserted %d at position %d.\n", data, position);
                break;
            case 4:
                printf("Linked List: ");
                printList(head);
                break;
            case 5:
                printf("Exiting...\n");
                exit(0);
            default:
                printf("Invalid choice. Please enter a valid option.\n");
        }
    }
    return 0;
}
```

Deletion in linked lists is also performed in three different ways.

- **Deletion at Beginning**
- **Deletion at Ending**
- **Deletion at a Given Position**

Deletion at Beginning

1. START
2. Assign the head pointer to the next node in the list
3. Delete the first node using the free method
4. If the Linked List is empty that it is not possible to delete
5. END



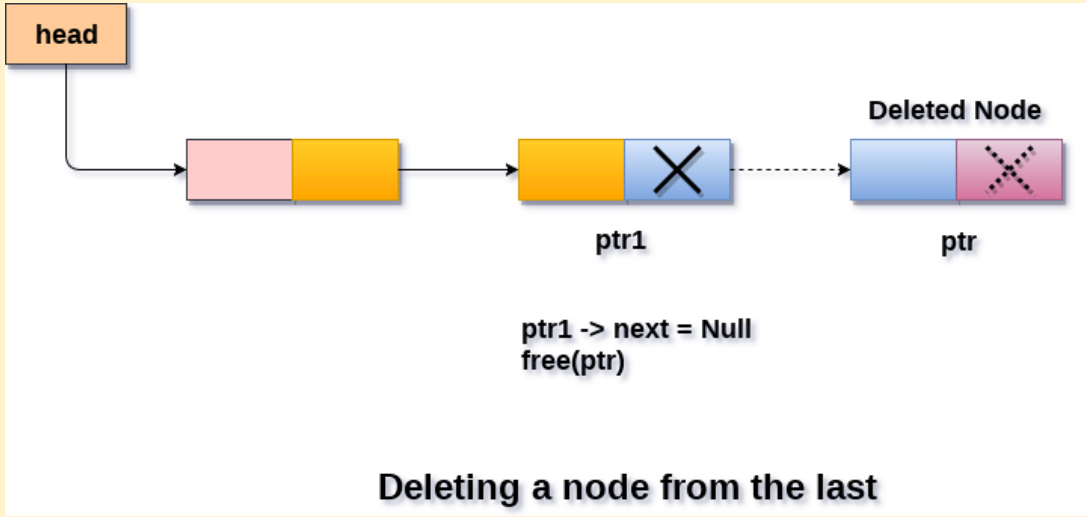
```
void deleteStart (struct Node **head)
{
  struct Node *temp = *head;

  // if there are no nodes in Linked List can't delete
  if (*head == NULL)
    {
      printf ("Linked List Empty, nothing to delete");
      return;
    }

  // move head to next node
  *head = (*head)->next;
  printf ("Deleted: %d\n", temp->data);
  free (temp);
}
```

Full Program

```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
  struct node *p = head;
  printf("\n[");

  //start from the beginning
  while(p != NULL) {
    printf(" %d ",p->data);
    p = p->next;
  }
  printf("]");
}

//insertion at the beginning
void insertatbegin(int data){

  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;

  // point it to old first node
  lk->next = head;

  //point first to new first node
  head = lk;
}
void deleteatbegin(){
  head = head->next;
}
int main(){
  int k=0;
  insertatbegin(12);
  insertatbegin(22);
  insertatbegin(30);
  insertatbegin(40);
  insertatbegin(55);
  printf("Linked List: ");

  // print list
  printList();
  deleteatbegin();
  printf("\nLinked List after deletion: ");
  // print list
  printList();
}
```

Deletion at Ending

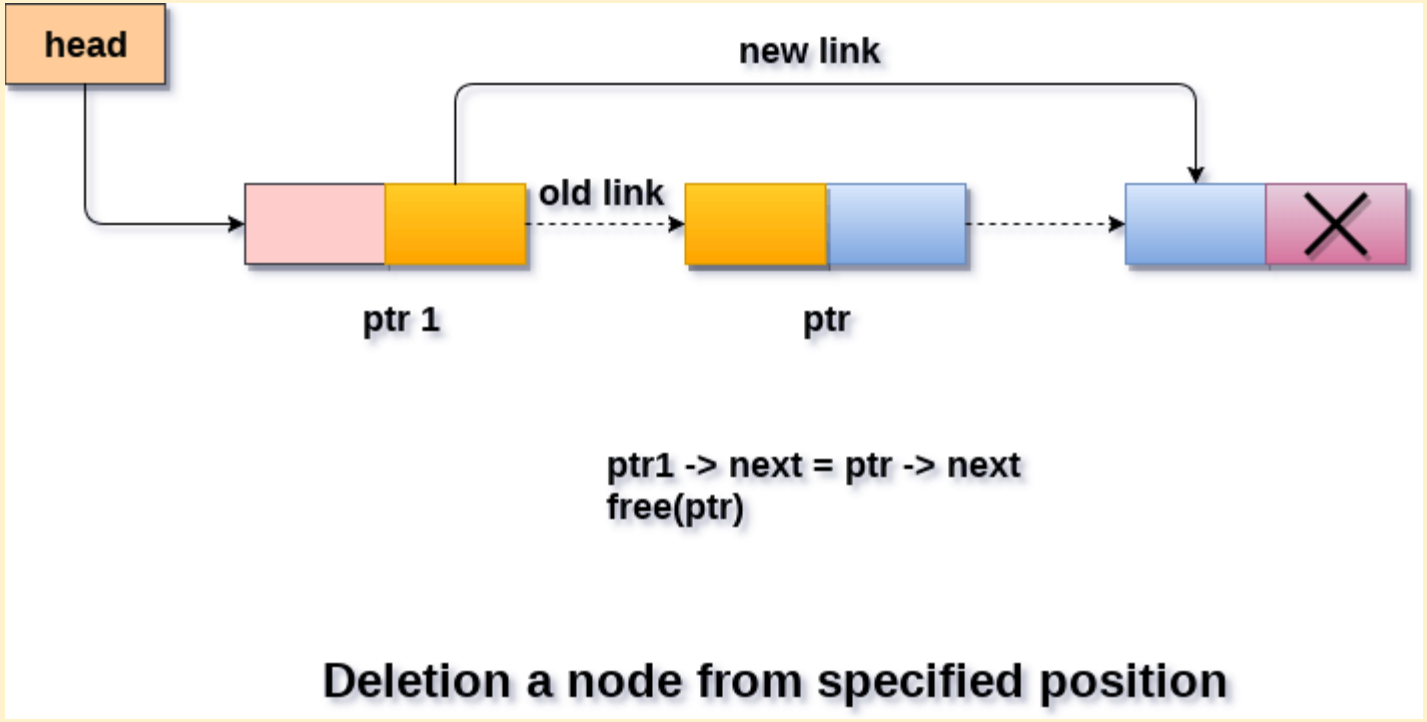
1. START
2. Iterate until you find the second last element in the list.
3. Assign NULL to the second last element in the list.
4. END



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

// display the list
void printList(){
  struct node *p = head;
  printf("\n[");

  //start from the beginning
  while(p != NULL) {
    printf(" %d ",p->data);
    p = p->next;
  }
  printf("]");
}
//insertion at the beginning
void insertatbegin(int data){
  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;

  // point it to old first node
  lk->next = head;

  //point first to new first node
  head = lk;
}
void deleteatend(){
  struct node *linkedlist = head;
  while (linkedlist->next->next != NULL)
    linkedlist = linkedlist->next;
  linkedlist->next = NULL;
}
void main(){
  int k=0;
  insertatbegin(12);
  insertatbegin(22);
  insertatbegin(30);
  insertatbegin(40);
  insertatbegin(55);
  printf("Linked List: ");

  // print list
  printList();
  deleteatend();
  printf("\nLinked List after deletion: ");
// print list
  printList();
}
```

Deletion at a Given Position

1. START
2. Iterate until find the current node at position in the list.
3. Assign the adjacent node of current node in the list
   to its previous node.
4. END



```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
// display the list
void printList(){
  struct node *p = head;
  printf("\n[");
  //start from the beginning
  while(p != NULL) {
    printf(" %d ",p->data);
    p = p->next;
  }
  printf("]");
}
//insertion at the beginning
void insertatbegin(int data){
  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  // point it to old first node
  lk->next = head;
  //point first to new first node
  head = lk;
}
void deletenode(int key){
  struct node *temp = head, *prev;
  if (temp != NULL && temp->data == key) {
    head = temp->next;
    return;
  }
  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
  }
  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;
}
void main(){
  int k=0;
  insertatbegin(12);
  insertatbegin(22);
  insertatbegin(30);
  insertatbegin(40);
  insertatbegin(55);
  printf("Linked List: ");

  // print list
  printList();
  deletenode(30);
  printf("\nLinked List after deletion: ");

  // print list
  printList();
}
```

Single Linked List - Search Operation

Searching for an element in the list using a key element. This operation is done in the same way as array search; comparing every element in the list with the key element given.

- **Algorithm**

1 START

2 If the list is not empty, iteratively check if the list contains the key

3 If the key element is not present in the list, unsuccessful search

4 END

Full program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;
// display the list
void printList(){
  struct node *p = head;
  printf("\n[");
  //start from the beginning
  while(p != NULL) {
    printf(" %d ",p->data);
    p = p->next;
  }
  printf("]");
}
//insertion at the beginning
void insertatbegin(int data){
  //create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
 // point it to old first node
  lk->next = head;
  //point first to new first node
  head = lk;
}
```

```
int searchlist(int key){
  struct node *temp = head;
  while(temp != NULL) {
    if (temp->data == key) {
      return 1;
    }
    temp=temp->next;
  }
  return 0;
}
void main(){
  int k=0;
  insertatbegin(12);
  insertatbegin(22);
  insertatbegin(30);
  insertatbegin(40);
  insertatbegin(55);
  printf("Linked List: ");

  // print list
  printList();
  int ele = 30;
  printf("\nElement to be searched is: %d", ele);
  k = searchlist(30);
  if (k == 1)
    printf("\nElement is found");
  else
    printf("\nElement is not found in the list");
}
```

Full Program with options

```
 #include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void main ()
{
    int choice,item,loc;
    do
    {
        printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
        scanf("%d",&choice);
        switch(choice)
        {
            case 1:
            printf("\nEnter the item\n");
            scanf("%d",&item);
            create(item);
            break;
            case 2:
            search();
            case 3:
            exit(0);
            break;
            default:
            printf("\nPlease enter valid choice\n");
    }while(choice != 3);
}
    void create(int item)
    {
        struct node *ptr = (struct node *)malloc(sizeof(struct node *));
        if(ptr == NULL)
        {
            printf("\nOVERFLOW\n");
        }
        else
        {
            ptr->data = item;
            ptr->next = head;
            head = ptr;
            printf("\nNode inserted\n");
        }

    }
void search()
{
    struct node *ptr;
    int item,i=0,flag;
    ptr = head;
    if(ptr == NULL)
    {
        printf("\nEmpty List\n");
    }
else
    {
        printf("\nEnter item which you want to search?\n");
        scanf("%d",&item);
        while (ptr!=NULL)
        {
            if(ptr->data == item)
            {
                printf("item found at location %d ",i+1);
                flag=0;
            }
            else
            {
                flag=1;
            }
            i++;
            ptr = ptr -> next;
        }
        if(flag==1)
        {
            printf("Item not found\n");
        }
    }

}
```

Single Linked List - Traversal Operation

- The traversal operation walks through all the elements of the list in an order and displays the elements in that order.

Algorithm

1. START
2. While the list is not empty and did not reach the end of the list,

   print the data in each node
3. END

```
ptr = head;
      while (ptr!=NULL)
         {
            ptr = ptr -> next;
         }
```

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
   int data;
   struct node *next;
};
struct node *head = NULL;
struct node *current = NULL;

   // display the list
   void printList(){
   struct node *p = head;
   printf("\n[");

   //start from the beginning
   while(p != NULL) {
      printf(" %d ",p->data);
      p = p->next;
   }
   printf("]");
}
```

```
 //insertion at the beginning
   void insertatbegin(int data){

   //create a link
   struct node *lk = (struct node*) malloc(sizeof(struct node));
   lk->data = data;

   // point it to old first node
   lk->next = head;

   //point first to new first node
   head = lk;
}
void main(){
   int k=0;
   insertatbegin(12);
   insertatbegin(22);
   insertatbegin(30);
   printf("Linked List: ");

   // print list
   printList();
}
```

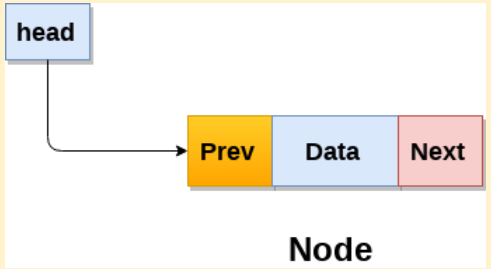
Program to implement Singly Linked List all operation

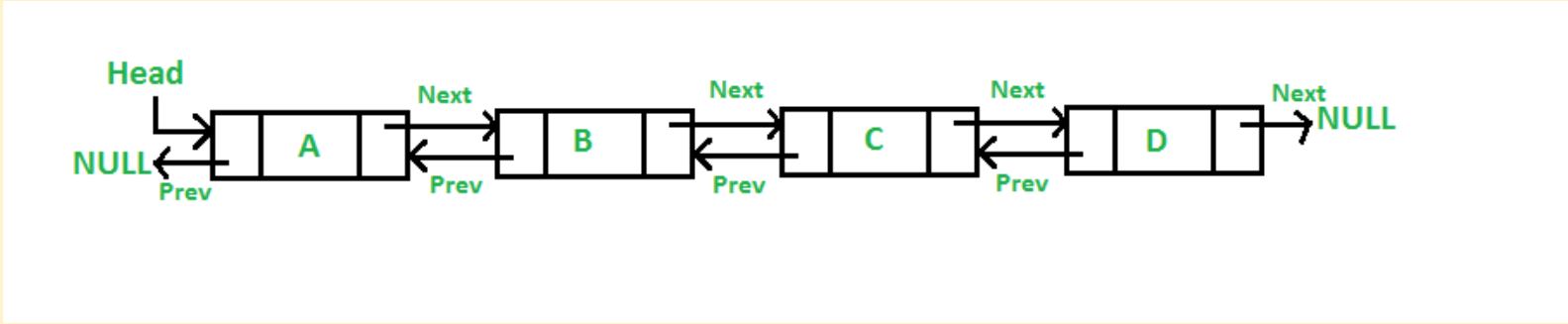
```c
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
struct node {
  int data;
  struct node *next;
};
struct node *head = NULL;
// Display the list
void printList() {
  struct node *p = head;
  printf("\n[");
  // Start from the beginning
  while (p != NULL) {
    printf(" %d ", p->data);
    p = p->next;
  }
  printf("]");
}
```

```c
// Insertion at the beginning
void insertatbegin(int data) {
  // Create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  // Point it to old first node
  lk->next = head;
  // Point first to new first node
  head = lk;
}

void insertatend(int data) {
  // Create a link
  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  lk->next = NULL;
  if (head == NULL) {
    head = lk;
    return;

  }
  struct node *linkedlist = head;
  // Traverse to the end of the list
  while(linkedlist->next != NULL)
    linkedlist = linkedlist->next;
  // Point last node to new node
  linkedlist->next = lk;
}

void insertafternode(struct node *list, int data) {
  if (list == NULL) {
    printf("Error: Node cannot be NULL");
    return;
  }

  struct node *lk = (struct node*) malloc(sizeof(struct node));
  lk->data = data;
  lk->next = list->next;
  list->next = lk;
}

void deleteatbegin() {
  if (head == NULL) {
    printf("Error: List is empty");
    return;
  }

  head = head->next;
}

void deleteatend() {
  if (head == NULL) {
    printf("Error: List is empty");
    return;
  }
```

```
  if (head->next == NULL) {
    free(head);
    head = NULL;
    return;
  }

  struct node *linkedlist = head;
  while (linkedlist->next->next != NULL)
    linkedlist = linkedlist->next;
  free(linkedlist->next);
  linkedlist->next = NULL;
}

void deletenode(int key) {
  struct node *temp = head, *prev = NULL;

  if (temp != NULL && temp->data == key) {
    head = temp->next;
    free(temp);
    return;
  }

  // Find the key to be deleted
  while (temp != NULL && temp->data != key) {
    prev = temp;
    temp = temp->next;
  }

  // If the key is not present
  if (temp == NULL) return;

  // Remove the node
  prev->next = temp->next;
  free(temp);
}

int searchlist(int key) {
  struct node *temp = head;
  while(temp != NULL) {
    if (temp->data == key) {
      return 1;
    }
    temp=temp->next;
  }
  return 0;
}

int main() {
  insertatbegin(12);
  insertatbegin(22);
  insertatend(30);
  insertatend(44);
  insertatbegin(50);
  insertafternode(head->next->next, 33);
  printf("Linked List: ");

  // Print list
  printList();
  deleteatbegin();
  deleteatend();
  deletenode(12);
  printf("\nLinked List after deletion: ");

  // Print list
  printList();
  insertatbegin(4);
  insertatbegin(16);
  printf("\nUpdated Linked List: ");
  printList();
  int k = searchlist(16);
  if (k == 1)
    printf("\nElement is found");
  else
    printf("\nElement is not present in the list");

  return 0;
}
```

Types of Linked Lists:



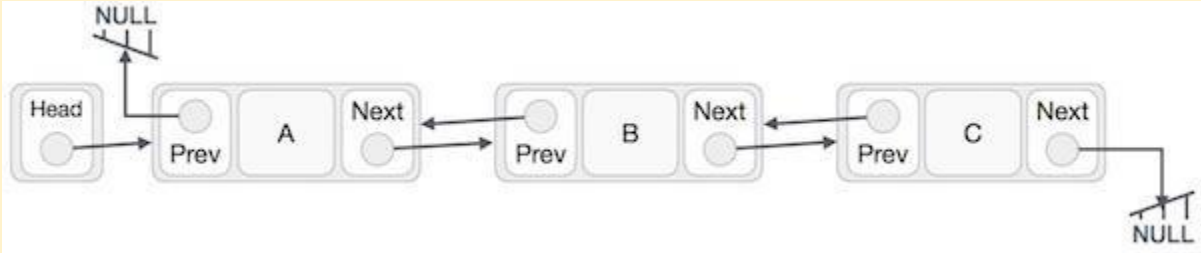


2. Doubly Linked List

In a doubly linked list, each node has two pointers: one pointing to the next node and one pointing to the previous node.
This allows traversal in both forward and backward directions.
Example:

NULL <- [Data1|Prev|Next] <-> [Data2|Prev|Next] <-> [Data3|Prev|Next] <-> ... <-> [DataN|Prev|NULL]

Doubly Linked List Representation



Link − Each link of a linked list can store a data called an element.

Next − Each link of a linked list contains a link to the next link called Next.

Prev − Each link of a linked list contains a link to the previous link called Prev.

Linked List − A Linked List contains the connection link to the first link called First and to the last link called Last.

Doubly Linked List (eg)

```
// Node of a doubly linked list

struct Node {
        int data;

        // Pointer to next node in DLL
        struct Node* next;

        // Pointer to previous node in DLL
        struct Node* prev;
};
```

Doubly Linked List Operations

- **Insertion on a Doubly Linked List**
- **Deletion from a Doubly Linked List**
- **Traversal Operations**
- **Search**

Doubly Linked List Operations

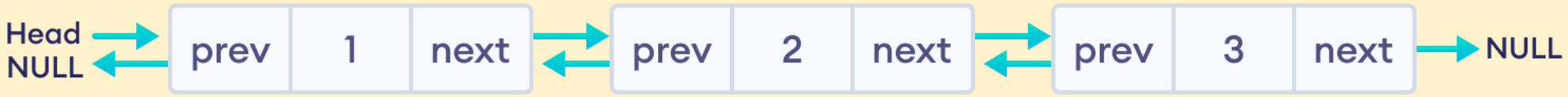
- **Insertion on a Doubly Linked List**

1. Insertion at the beginning
2. Insertion in-between nodes
3. Insertion at the End

Doubly Linked List - Insertion at the Beginning

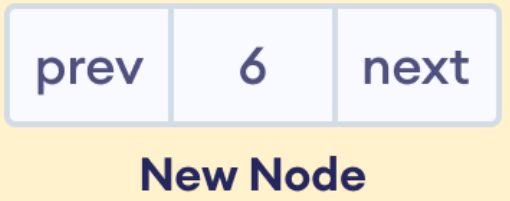
1. START
2. Create a new node with three variables: prev, data, next.
3. Store the new data in the data variable
4. If the list is empty, make the new node as head.
5. Otherwise, link the address of the existing first node to the next variable of the new node, and assign null to the prev variable.
6. Point the head to the new node.
7. END

Doubly linked list-Insertion at the Beginning



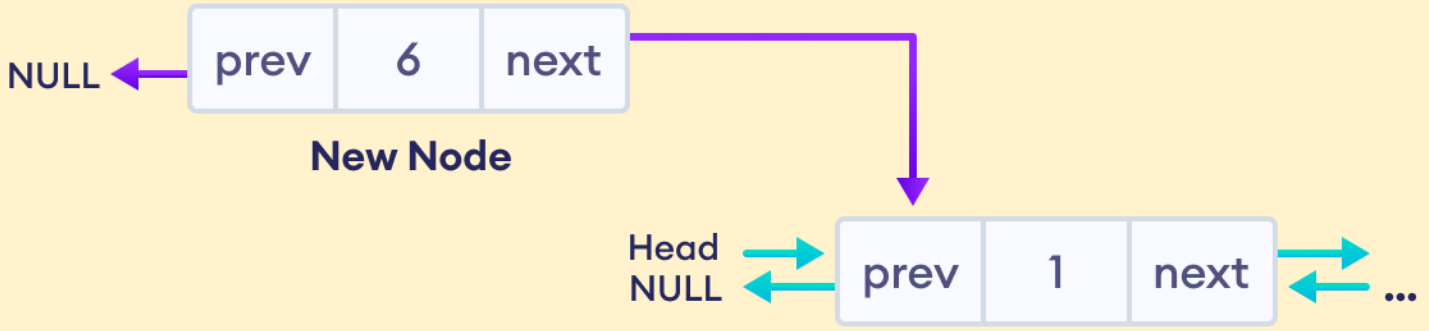
1. Create a new node

- allocate memory for newNode
- assign the data to newNode.



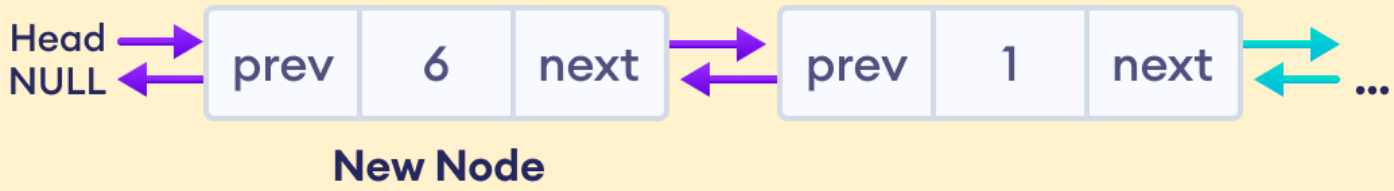
2. Set prev and next pointers of new node

- point next of newNode to the first node of the doubly linked list
- point prev to null
- Reorganize the pointers (changes are denoted by purple arrows)



3. Make new node as head node

- Point prev of the first node to newNode (now the previous head is the second node)
- Point head to newNode
- Reorganize the pointers



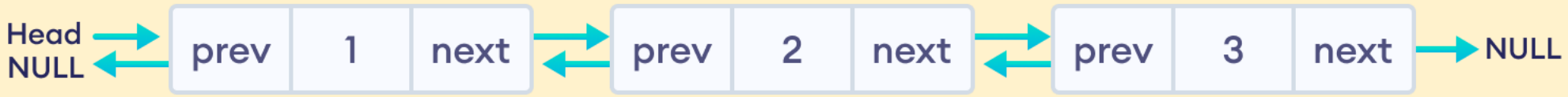
```
// insert node at the front
void insertFront(struct Node** head, int data) {

    // allocate memory for newNode
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // point next of newNode to the first node of the doubly linked list
    newNode->next = (*head);

    // point prev to NULL
    newNode->prev = NULL;

    // point previous of the first node (now first node is the second node) to newNode
    if ((*head) != NULL)
        (*head)->prev = newNode;

    // head points to newNode
    (*head) = newNode;
}
```

Full program

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <stdbool.h>
struct node {
  int data;   int key;
  struct node *next;
  struct node *prev;
};
//this link always point to first Link
struct node *head = NULL;
//this link always point to last Link
struct node *last = NULL;
struct node *current = NULL;
//is list empty
bool isEmpty(){
  return head == NULL;
}
//display the doubly linked list
void printList(){
  struct node *ptr = head;
  while(ptr != NULL) {
    printf("(%d,%d) ",ptr->key,ptr->data);
    ptr = ptr->next;
  }
```

```
//insert link at the first location
void insertFirst(int key, int data){
  //create a link
  struct node *link = (struct node*) malloc(sizeof(struct node));
  link->key = key;
  link->data = data;
  if(isEmpty()) {

    //make it the last link
    last = link;
  } else {

    //update first prev link
    head->prev = link;
  }

  //point it to old first link
  link->next = head;

  //point first to new first link
  head = link;
}
void main(){
  insertFirst(1,10);
  insertFirst(2,20);
  insertFirst(3,30);
  insertFirst(4,1);
  insertFirst(5,40);
  insertFirst(6,56);
  printf("\nDoubly Linked List: ");
  printList();
}
```

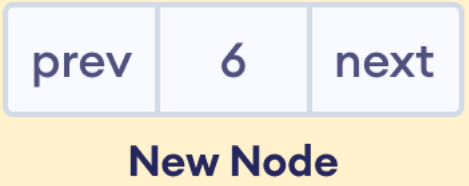
2. Insertion in between two nodes



Add a node with value 6 after node with value 1 in the doubly linked list.

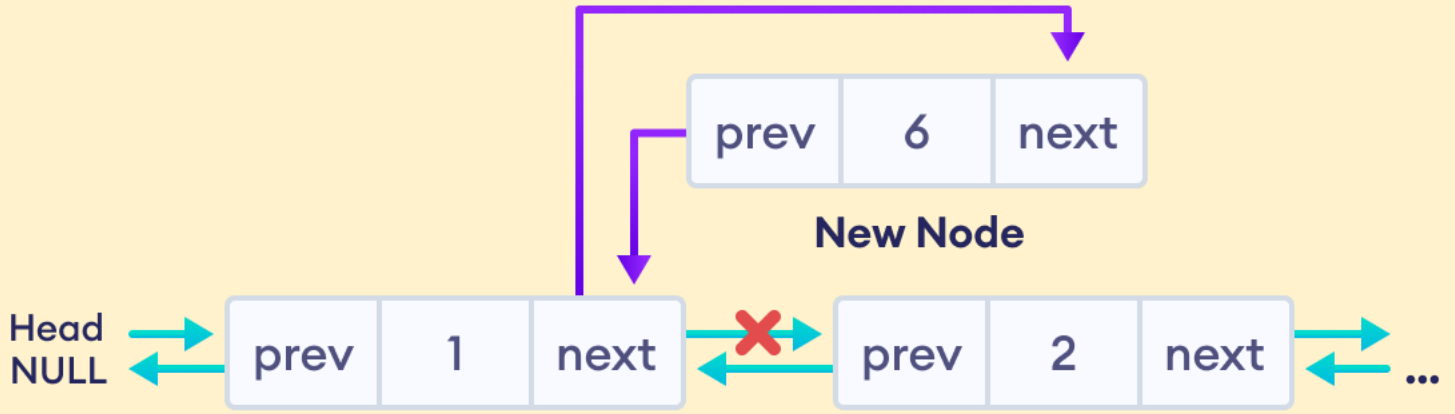
1. Create a new node

- allocate memory for newNode
- assign the data to newNode.



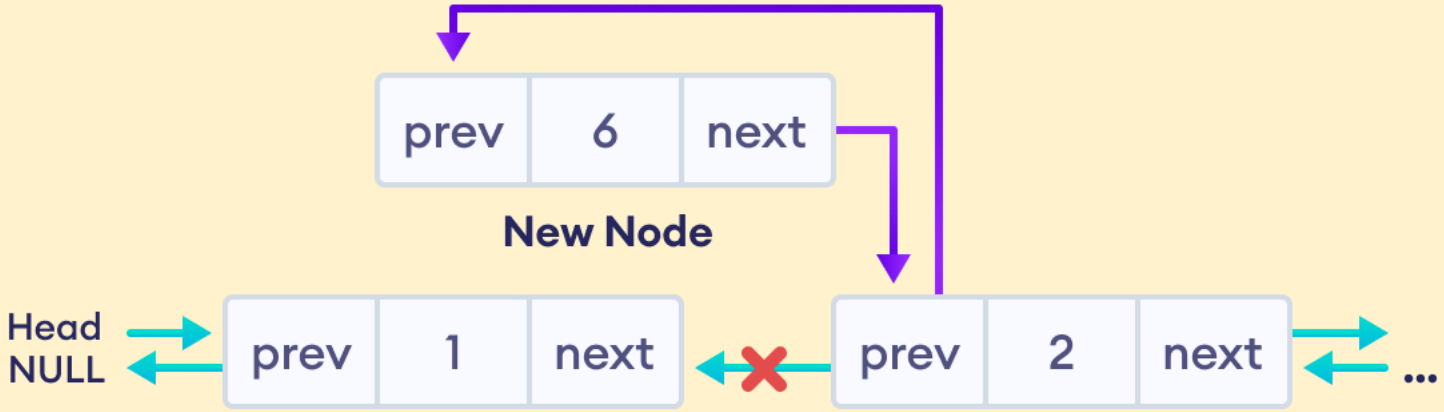
2. Set the next pointer of new node and previous node

- assign the value of next from previous node to the next of newNode
- assign the address of newNode to the next of previous node

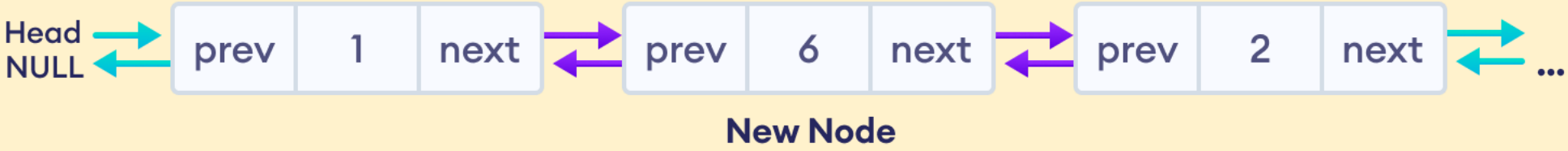


3. Set the prev pointer of new node and the next node

- assign the value of prev of next node to the prev of newNode
- assign the address of newNode to the prev of next node



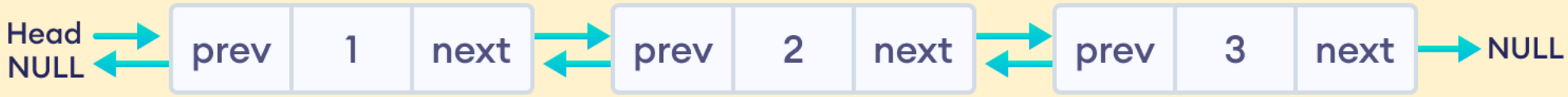
The final doubly linked list is after this insertion is:



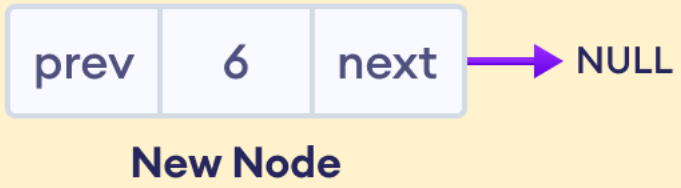
Code for Insertion in between two Nodes

```
// insert a node after a specific node
void insertAfter(struct Node* prev_node, int data) {

    // check if previous node is NULL
    if (prev_node == NULL) {
        cout << "previous node cannot be NULL";
        return;
    }

    // allocate memory for newNode
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // set next of newNode to next of prev node
    newNode->next = prev_node->next;

    // set next of prev node to newNode
    prev_node->next = newNode;

    // set prev of newNode to the previous node
    newNode->prev = prev_node;

    // set prev of newNode's next to newNode
    if (newNode->next != NULL)
        newNode->next->prev = newNode;
}
```

3. Insertion at the End

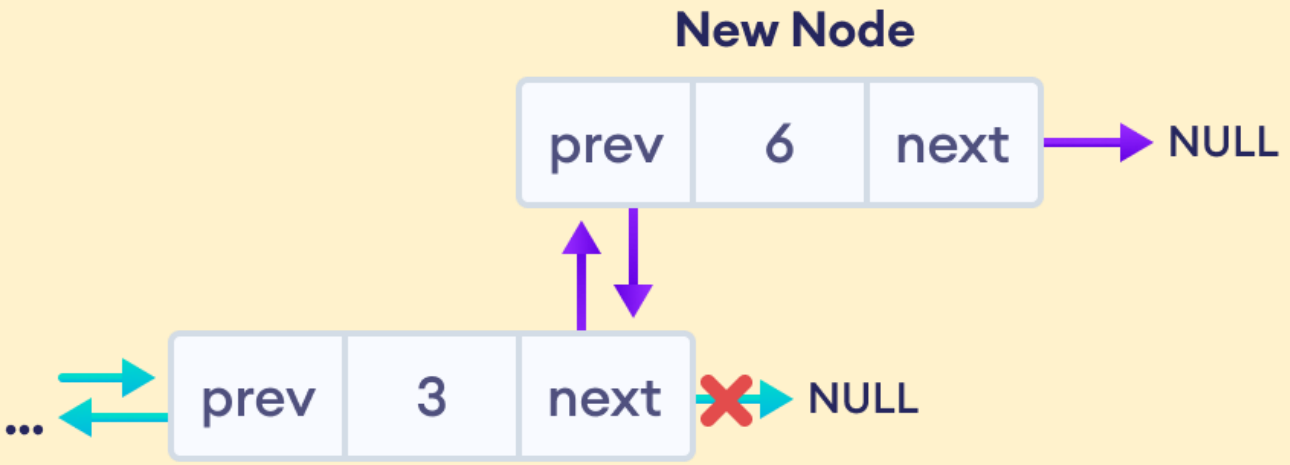


1. Create a new node



2. Set prev and next pointers of new node and the previous node

- If the linked list is empty, make the newNode as the head node. Otherwise, traverse to the end of the doubly linked list and



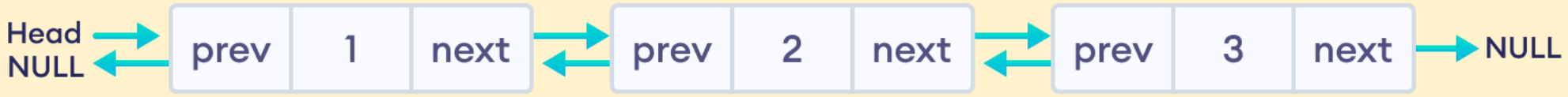
The final doubly linked list looks like this.

Code for Insertion at the End

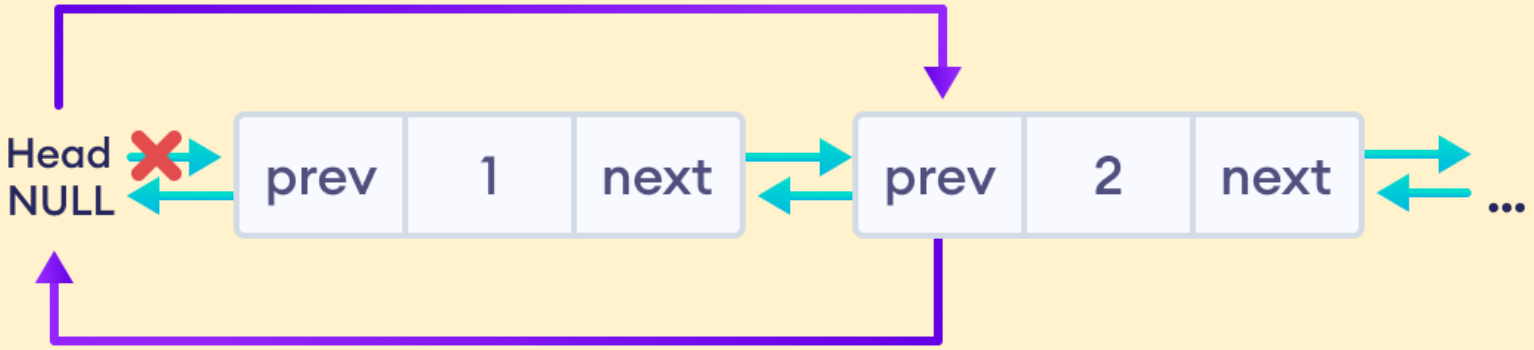
```
// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
    // allocate memory for node
    struct Node* newNode = new Node;

    // assign data to newNode
    newNode->data = data;

    // assign NULL to next of newNode
    newNode->next = NULL;

    // store the head node temporarily (for later use)
    struct Node* temp = *head;

    // if the linked list is empty, make the newNode as head node
    if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }

    // if the linked list is not empty, traverse to the end of the
linked list
    while (temp->next != NULL)
        temp = temp->next;

    // now, the last node of the linked list is temp

    // point the next of the last node (temp) to newNode.
    temp->next = newNode;

    // assign prev of newNode to temp
    newNode->prev = temp;
}
```

Deletion from a Doubly Linked List

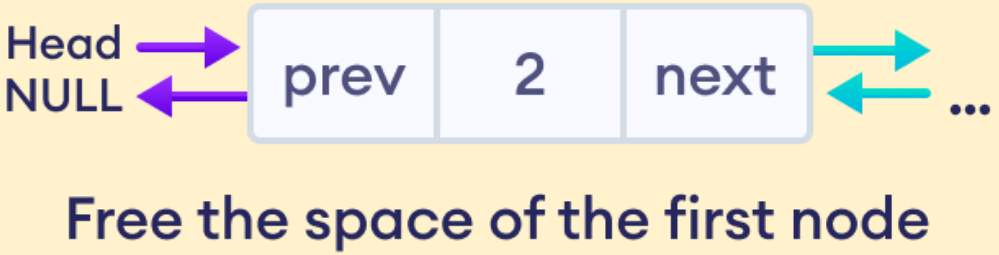
- Similar to insertion, we can also delete a node from **3** different positions of a doubly linked list.
- Suppose we have a double-linked list with elements **1**, **2**, and **3**.



- **1. Delete the First Node of Doubly Linked List**
- If the node to be deleted (i.e. del_node) is at the beginning
- **Reset value node after the del_node (i.e. node two)**



- Finally, free the memory of del_node. And, the linked will look like this

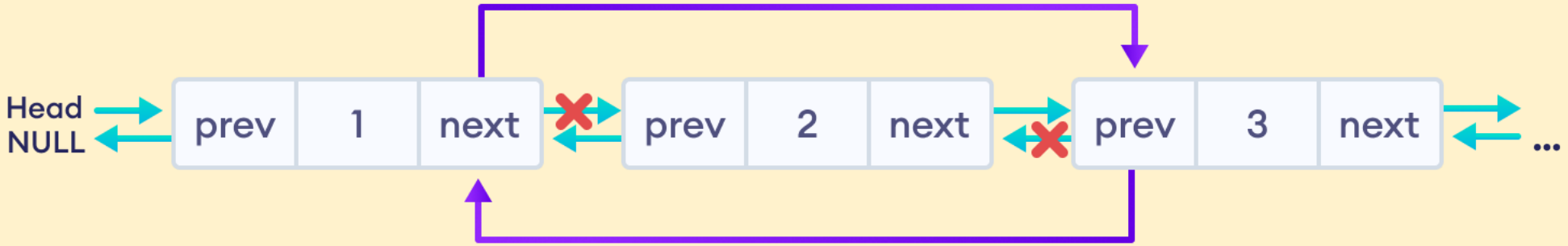


Code for Deletion of the First Node

```
if (*head == del_node)
    *head = del_node->next;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;

free(del);
```

2. Deletion of the Inner Node

- If del_node is an inner node (second node), we must have to reset the value of next and prev of the nodes before and after the del_node.
- **For the node before the del_node (i.e. first node)**
- Assign the value of next of del_node to the next of the first node.
- **For the node after the del_node (i.e. third node)**
- Assign the value of prev of del_node to the prev of the third node.



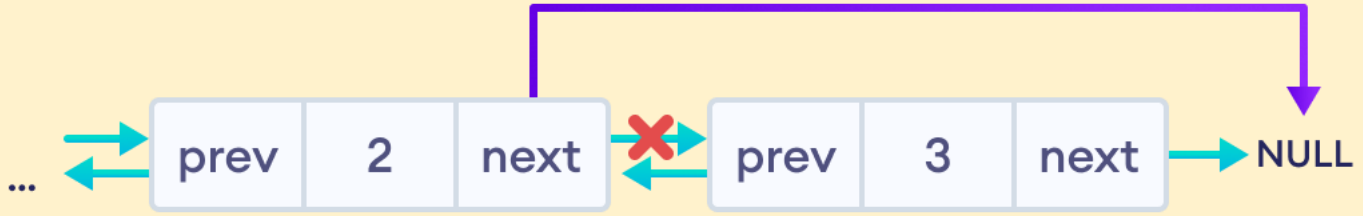
- Finally, we will free the memory of del_node. And, the final doubly linked list looks like this.

Code for Deletion of the Inner Node

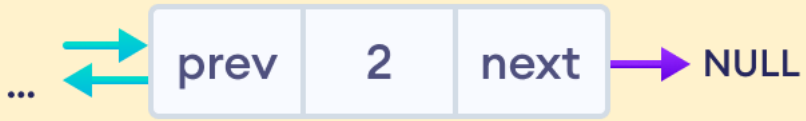
```
if (del_node->next != NULL)
    del_node->next->prev = del_node->prev;

if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

3. Delete the Last Node of Doubly Linked List

- In this case, we are deleting the last node with value **3** of the doubly linked list.
- Here, we can simply delete the del_node and make the next of node before del_node point to NULL.



- The final doubly linked list looks like this.



Code for Deletion of the Last Node

```
if (del_node->prev != NULL)
    del_node->prev->next = del_node->next;
```

- Here, del_node ->next is NULL so del_node->prev->next = NULL.

Searching for a specific node in Doubly Linked List

- Start
- Copy head pointer into a temporary pointer variable ptr.
- declare a local variable I and assign it to 0.
- Traverse the list until the pointer ptr becomes null. Keep shifting pointer to its next and increasing i by +1.
- Compare each element of the list with the item which is to be searched.
- If the item matched with any node value then the location of that value I will be returned from the function else NULL is returned.
- End

```c
#include<stdio.h>
#include<stdlib.h>
void create(int);
void search();
struct node
{
   int data;
   struct node *next;
   struct node *prev;
};
struct node *head;
void main ()
{
   int choice,item,loc;
   do
   {
      printf("\n1.Create\n2.Search\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         search();
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }
   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   if(ptr == NULL)
   {
      printf("\nOVERFLOW");
   }
   else
   {
     if(head==NULL)
   {
      ptr->next = NULL;
      ptr->prev=NULL;
      ptr->data=item;
      head=ptr;
   }
   else
   {
      ptr->data=item;printf("\nPress 0 to insert more ?\n");
      ptr->prev=NULL;
      ptr->next = head;
      head->prev=ptr;
      head=ptr;
   }
    printf("\nNode Inserted\n");
}
}
void search()
{
   struct node *ptr;
   int item,i=0,flag;
   ptr = head;
   if(ptr == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      printf("\nEnter item which you want to search?\n");
      scanf("%d",&item);
      while (ptr!=NULL)
      {
         if(ptr->data == item)
         {
            printf("\nitem found at location %d ",i+1);
            flag=0;
            break;
         }
         else
         {
            flag=1;
         }
         i++;
         ptr = ptr -> next;
      }
      if(flag==1)
      {
         printf("\nItem not found\n");
      }
   }
}
```

Traversing in doubly linked list

- START
- Copy the head pointer in any of the temporary pointer ptr.
- traverse through the list by using while loop.
- Keep shifting value of pointer variable **ptr** until we find the last node.
- The last node contains **null** in its next part.
- END

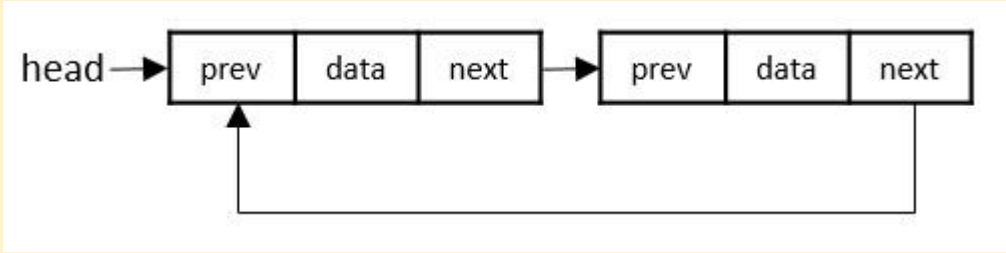
```
while(ptr != NULL)
      {
         printf("%d\n",ptr->data);
         ptr=ptr->next;
      }
```

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
int traverse();
struct node
{
   int data;
   struct node *next;
   struct node *prev;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Traverse\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
         case 2:
         traverse();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease enter valid choice\n");
      }
   }while(choice != 3);
}
void create(int item)
{

  struct node *ptr = (struct node *)malloc(sizeof(struct node));
  if(ptr == NULL)
  {
     printf("\nOVERFLOW\n");
  }
  else
  {
  if(head==NULL)
  {
     ptr->next = NULL;
     ptr->prev=NULL;
     ptr->data=item;
     head=ptr;
  }
  else
  {
     ptr->data=item;printf("\nPress 0 to insert more ?\n");
     ptr->prev=NULL;
     ptr->next = head;
     head->prev=ptr;
     head=ptr;
  }
   printf("\nNode Inserted\n");
}
}
int traverse()
{
   struct node *ptr;
   if(head == NULL)
   {
      printf("\nEmpty List\n");
   }
   else
   {
      ptr = head;
      while(ptr != NULL)
      {
         printf("%d\n",ptr->data);
         ptr=ptr->next;
      }
   }
}
```

Algorithm for Full Doubly Linked List Operation:

- Create a structure for the node containing data, a pointer to the next node, and a pointer to the previous node.
- Initialize head, tail, and temp pointers to NULL.
- Implement functions for:
- Adding a node to the end of the list.
- Adding a node to the beginning of the list.
- Adding a node at a specific position in the list.
- Deleting a node from the beginning of the list.
- Deleting a node from the end of the list.
- Deleting a node from a specific position in the list.
- Displaying the linked list from head to end.

Types of Linked Lists:



3. Circular Linked List

Circular linked lists can exist in both singly linked list and doubly linked list.

Since the last node and the first node of the circular linked list are connected, the traversal in this linked list will go on forever until it is broken.

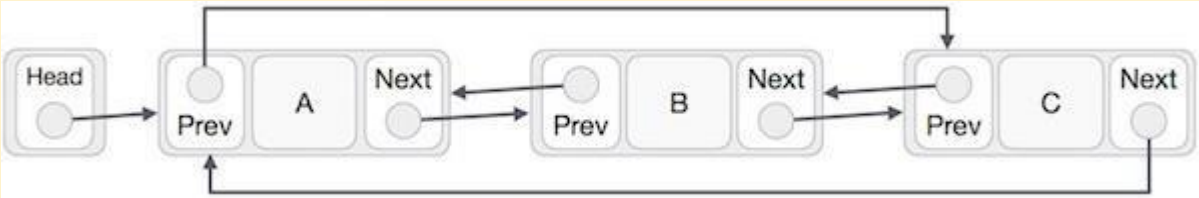
Example:

[Data1|Next] -> [Data2|Next] -> [Data3|Next] -> ... -> [DataN|Next] -> [Data1|Next]

Singly Linked List as Circular

- In singly linked list, the next pointer of the last node points to the first node.

Doubly Linked List as Circular



- The last link's next points to the first link of the list in both cases of singly as well as doubly linked list.
- The first link's previous points to the last of the list in case of doubly linked list.

- In doubly linked list, the next pointer of the last node points to the first node and the previous pointer of the first node points to the last node making the circular in both directions.

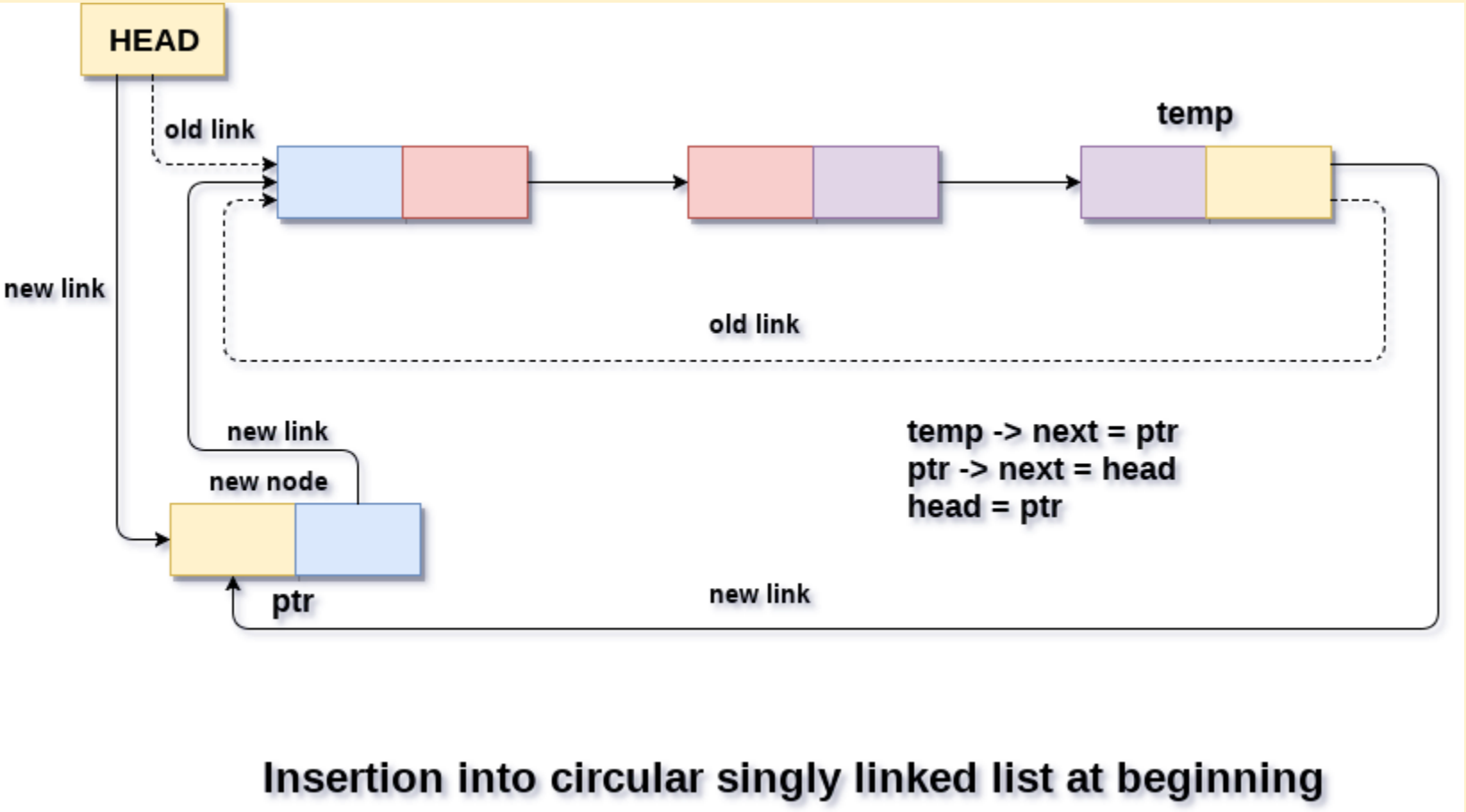
Insert at the beginning:

START

1. Create a new node with the given value.
2. If the list is empty, set the new node as both head and tail.
3. Otherwise, set the new node's next pointer to the current head.
4. Update the head to point to the new node.
5. Return the updated CSLL.
6. END

Example: Original CSLL: 2 -> 3 -> 4 After Insertion at the beginning with value 1: 1 -> 2 -> 3 -> 4 -> 1

Insert at the beginning:



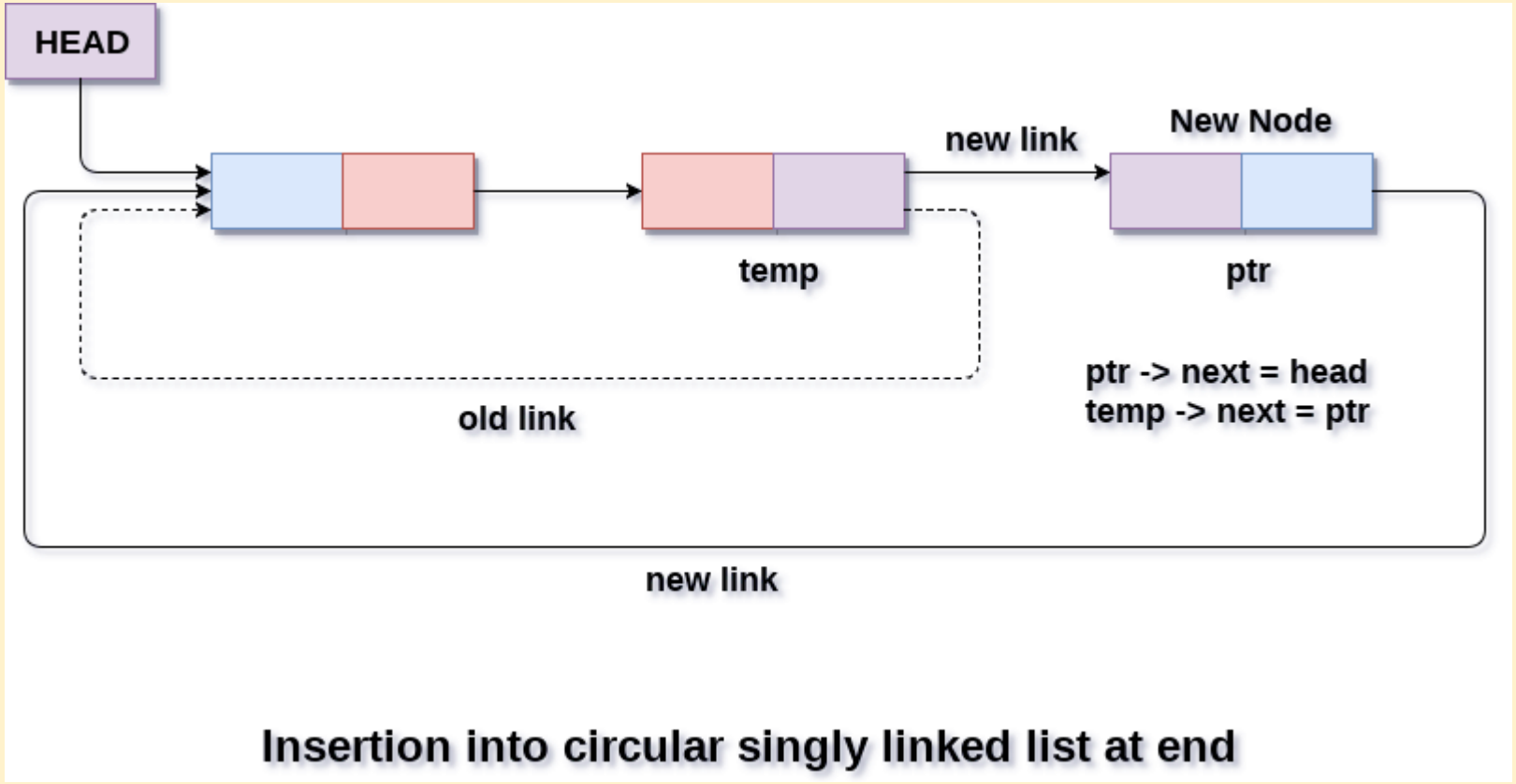
```
#include<stdio.h>
#include<stdlib.h>
void beg_insert(int);
struct node
{
    int data;
    struct node *next;
};
struct node *head;
void main ()
{
    int choice,item;
    do
    {
        printf("\nEnter the item which you want to insert?\n");
        scanf("%d",&item);
        beg_insert(item);
        printf("\nPress 0 to insert more ?\n");
        scanf("%d",&choice);
    }while(choice == 0);
}
void beg_insert(int item)
{

    struct node *ptr = (struct node *)malloc(sizeof(struct node));
    struct node *temp;
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
else
    {
        ptr -> data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
            temp = head;
            while(temp->next != head)
                temp = temp->next;
            ptr->next = head;
            temp -> next = ptr;
            head = ptr;
        }
    printf("\nNode Inserted\n");
    }

}
```

Insert at the end:

1. Create a new node with the given value.
2. If the list is empty, set the new node as both head and tail.
3. Otherwise, set the current tail's next pointer to the new node.
4. Update the tail to point to the new node.
5. Return the updated CSLL.

Example: Original CSLL: 1 -> 2 -> 3 After Insertion at the end with value 4: 1 -> 2 -> 3 -> 4 -> 1

Insertion into circular singly linked list at the end



Insertion into circular singly linked list at end

```
void lastinsert(struct node*ptr, struct node *temp, int item)
{
    ptr = (struct node *)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW\n");
    }
    else
    {
        ptr->data = item;
        if(head == NULL)
        {
            head = ptr;
            ptr -> next = head;
        }
        else
        {
```

```
temp = head;
            while(temp -> next != head)
            {
                temp = temp -> next;
            }
            temp -> next = ptr;
            ptr -> next = head;
        }
    }

}
```

Insert at a specific position:

1. START
2. Create a new node with the given value.
3. If the list is empty or position is 1, perform Insert_Beginning(value).
4. Otherwise, traverse the list until the desired position is reached.
5. Insert the new node at the specified position.
6. Return the updated CSLL.

Example: Original CSLL: 1 -> 3 -> 4 After Insertion of value 2 at position 2: 1 -> 2 -> 3 -> 4 -> 1

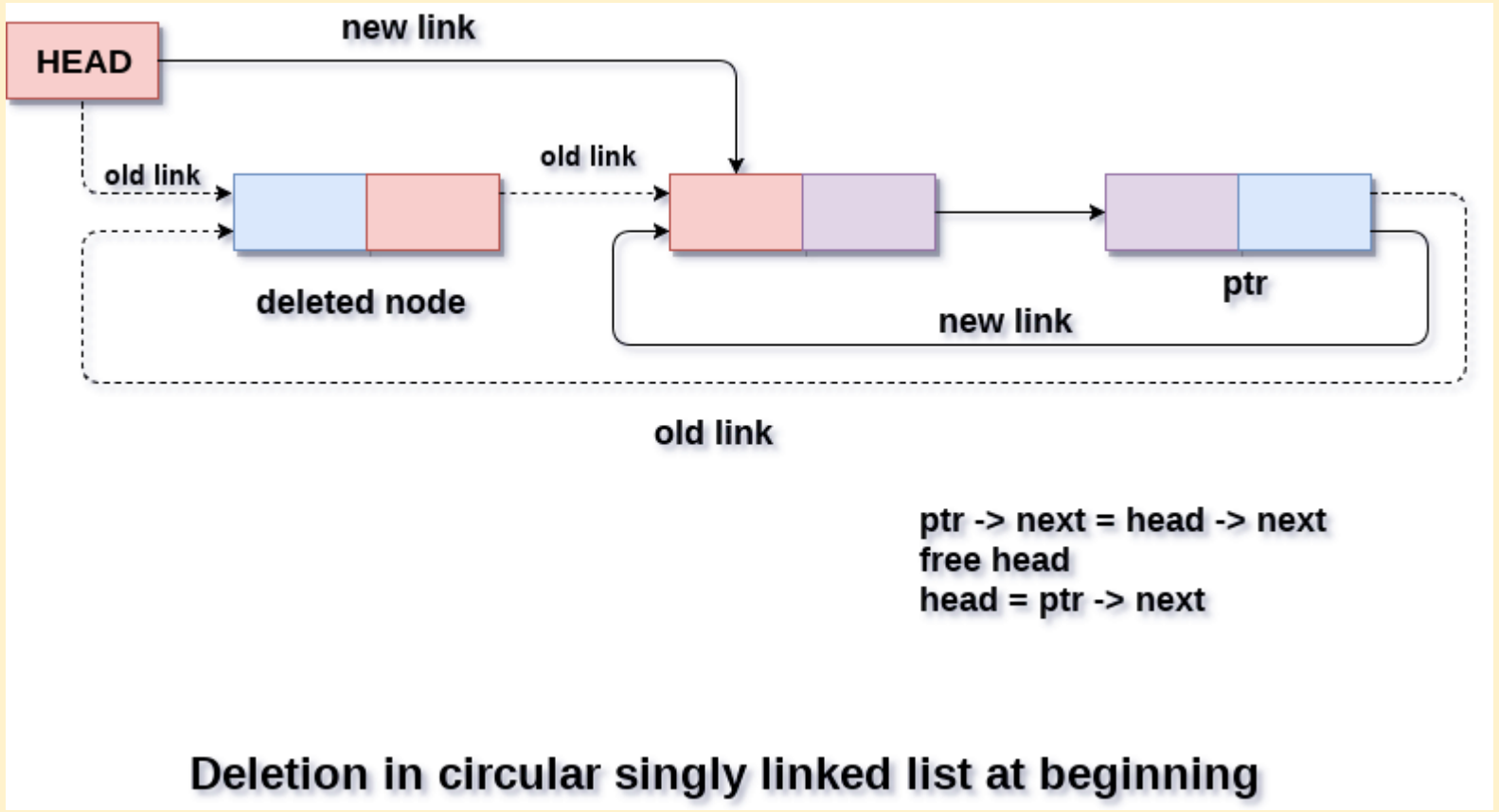
Deletion

- **The first node**
- **The last node**
- **A specific node**

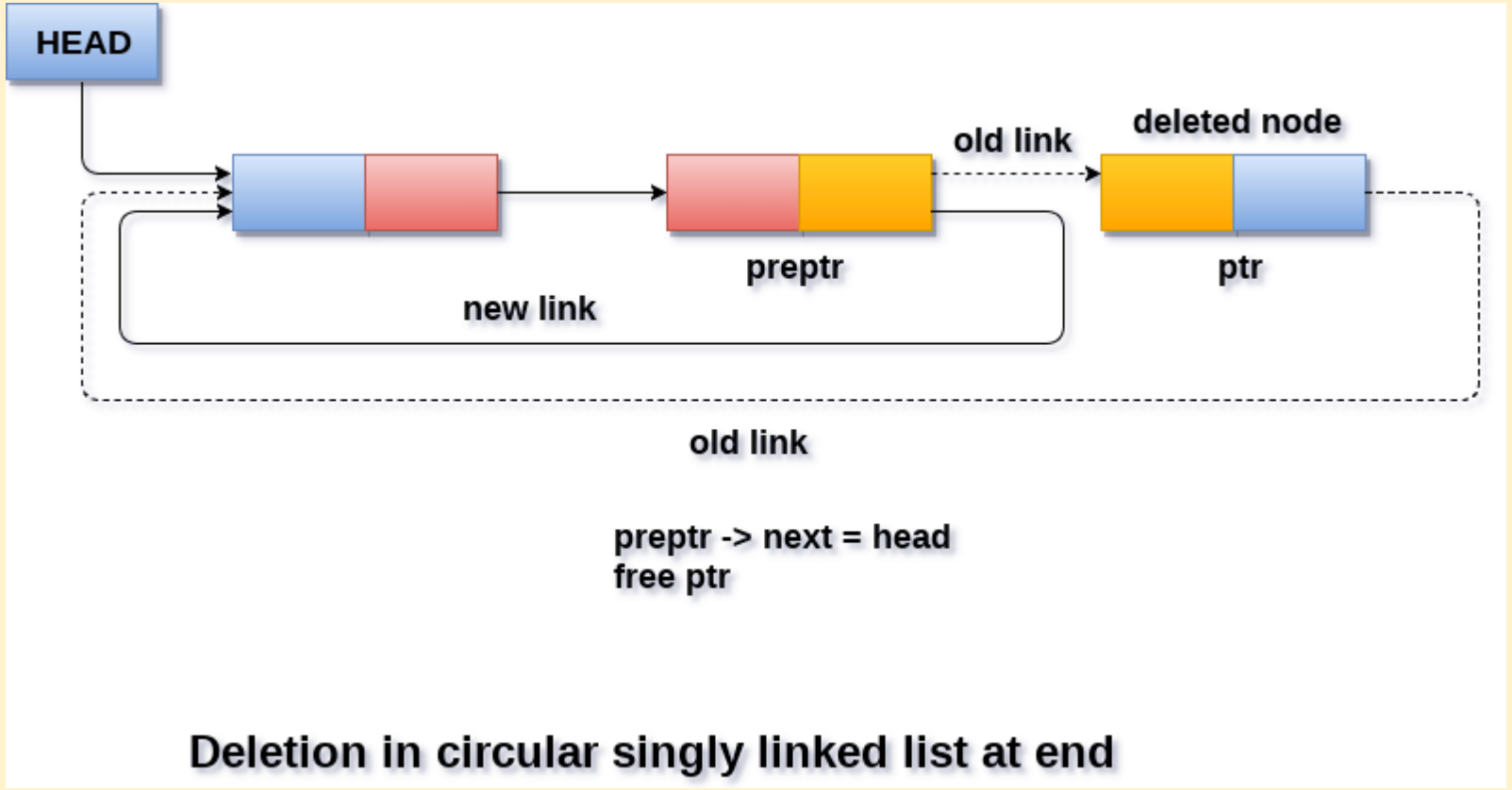
Circular Linked List - Deletion Operation

1. START
2. If the list is empty, then the program is returned.
3. If the list is not empty, we traverse the list using a current pointer that is set to the head pointer and create another pointer previous that points to the last node.
4. Suppose the list has only one node, the node is deleted by setting the head pointer to NULL.
5. If the list has more than one node and the first node is to be deleted, the head is set to the next node and the previous is linked to the new head.
6. If the node to be deleted is the last node, link the preceding node of the last node to head node.
7. If the node is neither first nor last, remove the node by linking its preceding node to its succeeding node.
8. END

Deletion in circular singly linked list at beginning



Deletion in Circular singly linked list at the end



C/W

- Write an Algorithm for Insertion new element i.e 7 after 5 into circular singly linked list. And also write algorithm for deletion of last element of the given CSLL.
- Given Circular Linked list: [2, 5, 3, 9, 6]

Program

```
#include<stdio.h>
#include<stdlib.h>
void create(int);
void last_delete();
struct node
{
   int data;
   struct node *next;
};
struct node *head;
void main ()
{
   int choice,item;
   do
   {
      printf("1.Append List\n2.Delete Node from end\n3.Exit\n4.Enter your choice?");
      scanf("%d",&choice);
      switch(choice)
      {
         case 1:
         printf("\nEnter the item\n");
         scanf("%d",&item);
         create(item);
         break;
 case 2:
         last_delete();
         break;
         case 3:
         exit(0);
         break;
         default:
         printf("\nPlease Enter valid choice\n");
      }

   }while(choice != 3);
}
void create(int item)
{

   struct node *ptr = (struct node *)malloc(sizeof(struct node));
   struct node *temp;
   if(ptr == NULL)
   {
      printf("\nOVERFLOW\n");
   }
   else
   {
      ptr -> data = item;
      if(head == NULL)
      {
         head = ptr;
         ptr -> next = head;
      }
      else
      {
 temp = head;
         while(temp->next != head)
            temp = temp->next;
         ptr->next = head;
         temp -> next = ptr;
         head = ptr;
      }
   printf("\nNode Inserted\n");
   }

}
void last_delete()
{
struct node *ptr, *preptr;
   if(head==NULL)
   {
      printf("\nUNDERFLOW\n");
   }
   else if (head ->next == head)
   {
      head = NULL;
      free(head);
      printf("\nNode Deleted\n");
   }
   else
   {
      ptr = head;
      while(ptr ->next != head)
      {
         preptr=ptr;
         ptr = ptr->next;
      }
      preptr->next = ptr -> next;
      free(ptr);
      printf("\nNode Deleted\n");
   }
}
```

Searching in circular singly linked list

1. START
2. If the list is empty, return NULL.
3. Initialize currentNode as head.
4. Loop until the end of the list (currentNode != tail.next):
   a. If currentNode's data matches the search value, return currentNode.
   b. Move to the next node, currentNode = currentNode.next.
5. If the value is not found, return NULL.
6. End

- Searching for value 3 in CSLL: 1 -> 2 -> 3 -> 4 -> 1
- Result: Node containing value 3 is found.

Traversing in Circular Singly linked list

- Start
- Initialize the temporary pointer variable **temp** to head pointer
- Run the while loop until the next pointer of temp becomes **head**.
- Print each nodes visited.
- End

```
while(ptr -> next != head)
    {

        printf("%d\n", ptr -> data);
        ptr = ptr -> next;
    }
```

Program

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

// Node structure for doubly linked list
struct Node {
    int data;
    struct Node* prev;
    struct Node* next;
};

// Function prototypes
void insertAtBeginning(struct Node** head, int newData);
void insertAtEnd(struct Node** head, int newData);
void deleteNode(struct Node** head, struct Node* delNode);
void displayList(struct Node* head);

// Function to insert a node at the beginning of the list
void insertAtBeginning(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    newNode->data = newData;
    newNode->prev = NULL;
    newNode->next = *head;
    if (*head != NULL) {
        (*head)->prev = newNode;
    }
    *head = newNode;
}

// Function to insert a node at the end of the list
void insertAtEnd(struct Node** head, int newData) {
    struct Node* newNode = (struct Node*)malloc(sizeof(struct Node));
    struct Node* last = *head;
    newNode->data = newData;
    newNode->next = NULL;

 if (*head == NULL) {
        newNode->prev = NULL;
        *head = newNode;
        return;
    }
    while (last->next != NULL) {
        last = last->next;
    }
    last->next = newNode;
    newNode->prev = last;
}
// Function to delete a node from the list
void deleteNode(struct Node** head, struct Node* delNode) {
    if (*head == NULL || delNode == NULL) {
        return;
    }
    if (*head == delNode) {
        *head = delNode->next;
    }

    if (delNode->next != NULL) {
        delNode->next->prev = delNode->prev;
    }

    if (delNode->prev != NULL) {
        delNode->prev->next = delNode->next;
    }
    free(delNode);
}

// Function to display the contents of the list
void displayList(struct Node* head) {
    struct Node* temp = head;
    printf("Doubly Linked List: ");
    while (temp != NULL) {
        printf("%d ", temp->data);
        temp = temp->next;
    }
    printf("\n");
}

int main() {
    struct Node* head = NULL;

    insertAtBeginning(&head, 10);
    insertAtEnd(&head, 20);
    insertAtBeginning(&head, 5);
    insertAtEnd(&head, 30);

    displayList(head);

    deleteNode(&head, head->next);  // Delete the second node
    displayList(head);

    return 0;
}
```

Applications of Linked Lists:

- **Polynomial Representation:** Linked lists are used to represent and manipulate polynomials efficiently, with each node representing a term in the polynomial.
- **Stacks and Queues:** Linked lists are commonly employed to implement stacks and queues due to their ease of insertion and deletion operations.
- **Arithmetic Operations:** Linked lists facilitate arithmetic operations on by storing digits in each node.
- **Graph Implementation:** Linked lists are crucial for implementing graphs, where adjacent vertices are stored in the nodes, aiding in graph algorithms like breadth-first search.

Real-World Applications

- **Music Players:** Linked lists are utilized in music players to create playlists, enabling users to navigate between songs easily.
- **Image Viewers:** Image viewers use linked lists for seamless navigation between images, with each image stored as a node.
- **Web Browsers:** Linked lists store browsing history in web browsers, allowing users to navigate between visited pages using forward and backward buttons.
- **Contact Management:** In mobile phones, linked lists organize contacts alphabetically, ensuring new contacts are inserted correctly.

Practical Work

- Write a program to implement singly, doubly, and circular linked list operations.
- For Visual Understanding: https://visualgo.net/en/list

END OF UNIT 3