Chapter 3
Linked List

Definition of a List

- A **list** is a linear data structure that stores elements in a sequential manner.
- It can contain **homogeneous** (same type) or **heterogeneous** (different types) elements, depending on the programming language.
- Lists allow **dynamic resizing**, meaning elements can be added or removed without fixed memory allocation.

Types of Lists in DSA

1. **Array**:
    - Fixed size.
    - Contiguous memory allocation.
    - Example: arr = [1, 2, 3]
2. **Linked List**:
    - Nodes connected via pointers.
    - Types:
        - **Singly Linked List**
        - **Doubly Linked List**
        - **Circular Linked List**
3. **Doubly Linked List**:
    - Nodes have pointers to the previous and next node.
4. **Circular Linked List**:
    - Last node points to the first node.

3. Common List Operations

Here are the most common list operations with examples in the context of DSA:

3.1 Accessing Elements

- Retrieve an element using an index or pointer.
- Example (in arrays): arr[2] gives the 3rd element.

3.2 Insertion

- Adding an element to a list at a specific position.
- Example:
  - **Array**: Shift elements to insert at index i.
  - **Linked List**: Update pointers to add a new node.

3.3 Deletion

- Removing an element from a list.
- Example:
  - **Array**: Shift elements after the removed element.
  - **Linked List**: Update pointers to bypass the node.

3.4 Traversal

- Visiting every element in the list.
- Example: Using a for loop for arrays or iterating over nodes in a linked list.

3.5 Searching

- Finding the position of a specific element in the list.
- Techniques:
  - **Linear Search** (O(n))
  - **Binary Search** (O(log n)) for sorted arrays.

3.6 Sorting

- Arranging elements in a specific order (ascending/descending).
- Algorithms:
  - **Bubble Sort**
  - **Merge Sort**
  - **Quick Sort**

3.7 Merging

- Combining two lists into one.

3.8 Splitting

- Dividing a list into multiple smaller lists.

3.9 Reversal

- Reversing the order of elements in the list.

List Abstract Data Type (ADT)

Definition

- A **List ADT (Abstract Data Type)** represents a collection of elements that are stored in a sequential manner.
- It provides various operations such as insertion, deletion, traversal, and search without specifying how these operations are implemented (array-based or linked list-based).

Operations in List ADT

1. **Create a List**: Initialize an empty list.
2. **Insert**: Add an element at a specific position.
3. **Delete**: Remove an element at a specific position.
4. **Search**: Find the position of a specific element.
5. **Traverse**: Iterate through the list to access elements.
6. **Update**: Modify an element at a specific position.

Types of List ADT Implementations

1. **Array-based implementation**: Uses a fixed-size or dynamic array.
2. **Linked List implementation**: Uses nodes with pointers to manage elements dynamically.

Linked list

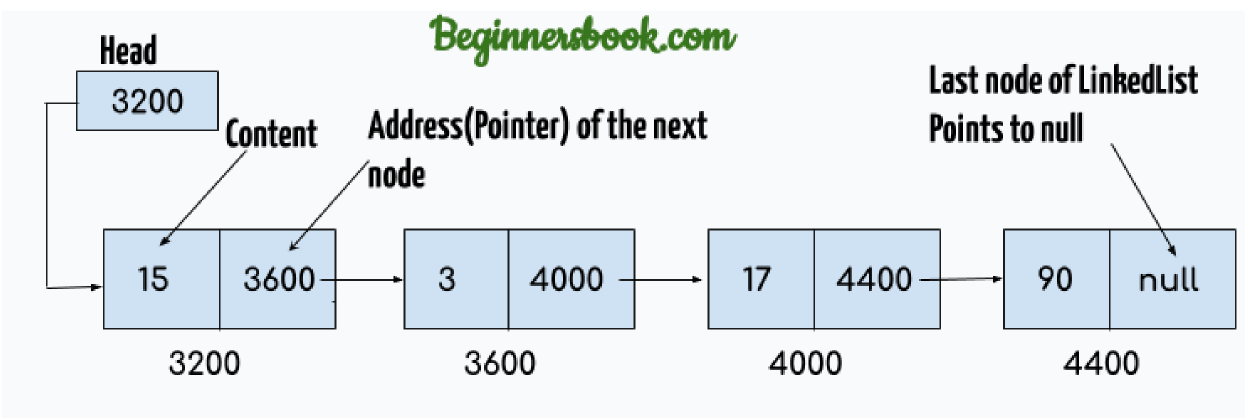
A linked list is a linear data structure where each element, known as a node, is connected to the next one using pointers. Unlike array, elements of linked list are stored in random memory locations.

What is a Linked List?

A linked list is a sequence of nodes where each node contains two parts:

- **Data**: The value stored in the node.
- **Pointer**: A reference to the next node in the sequence.
  (There can be multiple pointers for different kind of linked list.)

Unlike arrays, linked lists do not store elements in contiguous memory locations. Instead, each node points to the next, forming a chain-like structure and to access any element (node), we need to first sequentially traverse all the nodes before it.
It is a recursive data structure in which any smaller part of it is also a linked list in itself.



```c
struct node {
  int data;
  struct node *next;
};
```

Linked List Operations: Traverse, Insert and Delete

There are various linked list operations that allow us to perform different actions on linked lists. For example, the insertion operation adds a new element to the linked list.

Here's a list of basic linked list operations that we will cover in this article.

- Traversal - access each element of the linked list
- Insertion - adds a new element to the linked list
- Deletion - removes the existing elements
- Search - find a node in the linked list
- Sort - sort the nodes of the linked list

Representation of Linked List in C

In C, linked lists are represented as the pointer to the first node in the list. For that reason, the first node is generally called **head** of the linked list. Each node of the linked list is represented by a structure that contains a data field and a pointer of the same type as itself. Such structure is called self-referential structures.

Things to Remember about Linked List

- `head` points to the first node of the linked list
- `next` pointer of the last node is `NULL`, so if the next current node is `NULL`, we have reached the end of the linked list.

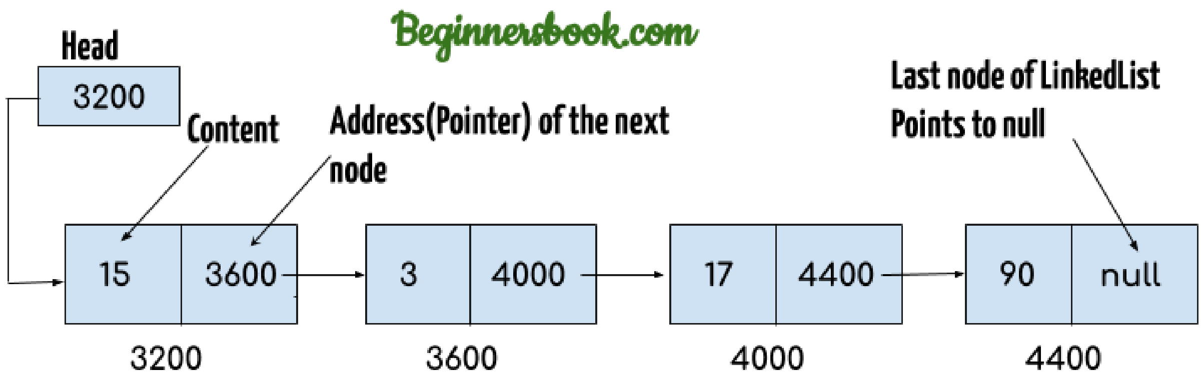
Types of Linked List in C

Linked list can be classified on the basis of the type of structure they form as a whole and the direction of access. Based on this classification, there are five types of linked lists:

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

Singly Linked List in C

A linked list or singly linked list is a linear data structure that is made up of a group of nodes in which each node has two parts: the data, and the pointer to the next node. The last node's (also known as tail) pointers point to NULL to indicate the end of the linked list.



Representation of Singly Linked List in C

A linked list is represented as a pointer to the first node where each node contains:

- **Data:** Here the actual information is stored.
- **Next:** Pointer that links to the next node.

```c
struct node {
  int data;
  struct node *next;
};
```

Insertion in Linked List:

- Insert at the beginning
- Insert at end
- Insert after a given location

Create and Display Singly Linked List

Algorithm: createLinkedList

Input: None
Output: Pointer to the head of the linked list

1. **Initialize**:
   - Set head = NULL. Declare newnode and temp.
2. **Repeat Until User Stops**:
   - Allocate memory for newnode using malloc.
   - If allocation fails, print error and return head.
   - Input data into newnode->data and set newnode->next = NULL.
   - If head == NULL, set head = temp = newnode. Otherwise, link temp->next = newnode and update temp = newnode.
   - Ask the user: "Continue? (0 to stop, 1 to continue): " and read choice.
3. **Return**:
   - Return `head`.

struct node {

int data;

struct node *next;

```
};

// Function to create the linked list
struct node* createLinkedList() {
    struct node *head, *newnode, *temp;
    head = NULL;
        // Allocate memory for the new node
        newnode = (struct node *)malloc(sizeof(struct node));
        if (newnode == NULL) {
            printf("Memory allocation failed.\n");
            return head;
        }

        // Input data for the new node
        printf("Enter data: ");
        scanf("%d", &newnode->data);
        newnode->next = NULL;

        // Add the new node to the list
        if (head == NULL) {
            head = temp = newnode; // First node
```

```
        } else {
            temp->next = newnode; // Add at the end
            temp = newnode;       // Update temp
        }

        // Ask the user if they want to continue
        printf("Do you want to continue (0 to stop, 1 to continue)? ");
        scanf("%d", &choice);

    return head;
}
```

 # C program to create and display linked list

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

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

// Function to create the linked list
struct node* createLinkedList() {
```

```
struct node *head, *newnode, *temp;
int choice; // Declare the variable to store user input
head = NULL;

do {
    // Allocate memory for the new node
    newnode = (struct node *)malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }

    // Input data for the new node
    printf("Enter data: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;

    // Add the new node to the list
    if (head == NULL) {
        head = temp = newnode; // First node
    } else {
        temp->next = newnode; // Add at the end
        temp = newnode;       // Update temp
    }
```

```c
        // Ask the user if they want to continue
        printf("Do you want to continue (0 to stop, 1 to continue)? ");
        scanf("%d", &choice);

    } while (choice == 1); // Repeat until the user decides to stop

    return head;
}

// Function to display the linked list
void displayLinkedList(struct node* head) {
    struct node* temp = head;
    printf("The linked list is: ");
    while (temp != NULL) {
        printf("%d -> ", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}

int main() {
    // Create the linked list
    struct node* head = createLinkedList();

    // Display the linked list
```

displayLinkedList(head);

return 0;

}

---

1. **Insertion at the beginning**

Algorithm to Insert a Node at the Beginning of a Singly Linked List

1. **Step 1**: Define a structure for a node of the linked list.
   - Each node contains:
     - data to store the value.
     - next (a pointer) to link to the next node.
2. **Step 2**: Initialize the linked list.
   - Start with an empty list where head is set to NULL.
3. **Step 3**: Allocate memory for a new node.
   - Use the malloc() function to allocate memory dynamically for the new node.
4. **Step 4**: Check if memory allocation is successful.
   - If malloc fails (returns NULL), display an error message and return head (the list remains empty).
5. **Step 5**: Input data into the new node.
   - Prompt the user to enter data.
   - Store the entered data in the data field of the new node.
6. **Step 6**: Update the links.
   - Set the next pointer of the new node to NULL (or another node if inserting in an existing list).
   - Set head to point to the new node if the list was empty.
7. **Step 7**: Return the pointer to the head of the list.

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

-
- // Function to insert an element at the beginning of the linked list
- struct node* insertAtBeginning(struct node *head){
-     struct node *head, *newnode, *temp;
-     head = NULL;
-         // Allocate memory for the new node
-         newnode = (struct node *)malloc(sizeof(struct node));
-         if (newnode == NULL) {
-             printf("Memory allocation failed.\n");
-             return head;
-         }
-
-         // Input data for the new node
-         printf("Enter data: ");
-         scanf("%d", &newnode->data);
-
-         // Point the new node's 'next' to the current head
-         newnode->next = head;
-         head = newnode; // Update head to the new node
-
-     return head;
- }

2.Delete from the beginning
Algorithm:

Input:

- head: Pointer to the first node of the linked list.

Output:

- Updated head after deleting the node at the beginning of the linked list.

Steps:

1. **Check if the List is Empty**:

    - If head == NULL, print **"List is empty"** and return the current head (which is NULL).
2. **Delete the First Node**:
    - Save the current head node in a temporary pointer temp.
    - Set head = head->next to move the head pointer to the next node.
    - Free the memory of the node stored in temp (the old head node) using free(temp).
3. **Return the New Head**:
    - Return the updated head pointer, which could be NULL if the list is now empty.

- struct node {
- int data;
- struct node *next;
- };
-
- // Function to delete node from the beginning of linked list
- struct node* deleteFromBeginning(struct node *head) {
- struct node *head, *temp;
- if (head ==0){
- printf(“List is empty.”);
- return head;
- }
- else: {
- // Save the head node in temp and move head to the next node
- temp = head;
- head = head -> next;
- // Free the memory of the old head node
- free(temp);
- }
-
- return head;
- }

3.Insertion at the end:

Algorithm to Insert a Node at the End of a Singly Linked List

1. **Initialize**:
   - Create a pointer newnode to store the address of the new node.
   - Create a pointer temp for traversing the linked list.
2. **Allocate Memory for the New Node**:
   - Use malloc() to allocate memory for newnode.
   - If malloc() fails, print **"Memory allocation failed"** and return the current value of head.
3. **Input Data**:
   - Prompt the user to input the data to store in the new node.
   - Assign this data to newnode->data.
4. **Set next Pointer**:
   - Set newnode->next to NULL since it will be the last node in the list.
5. **Check if the List is Empty**:
   - If head == NULL (the list is empty):
     - Set head = newnode to make the new node the first node.
     - Return the updated head.
6. **Traverse to the End**:
   - If the list is not empty:
     - Initialize temp = head.
     - While temp->next != NULL:
       - Move to the next node by setting
       - temp = temp->next.
7. **Insert the New Node**:
   - Once the end of the list is reached (temp->next == NULL):
     - Set temp->next = newnode to link the new node at the end.
8. **Return the Updated head**:
   - Return the head pointer of the linked list.

- struct node {
-     int data;
-     struct node *next;
- };
-
- // Function to insert an element at the end of the linked list
- struct node* insertAtEnd(struct node *head){
-     struct node *head, *newnode, *temp;
-     head = NULL;
-         // Allocate memory for the new node
-         newnode = (struct node *)malloc(sizeof(struct node));
-         if (newnode == NULL) {
-             printf("Memory allocation failed.\n");
-             return head;
-         }
-
-         // Input data for the new node
-         printf("Enter data: ");
-         scanf("%d", &newnode->data);
-         newnode->next = Null;
-
-         // Check if the list is empty
-           if (head == NULL) {
-                 head = newnode; // Make the new node the first node return head;
-             }
-
-         while (temp -> next != Null) // to traverse the temp
-         {
-                 temp = temp -> next;
-           }
-           temp ->next = newnode; // address is stored in newnode
-
-     return head;
- }

4. **Delete from the end:**

Algorithm to delete a Node from the End of a Singly Linked List

1. **Initialize:**
   - Define struct node with data and next fields.
   - Declare pointers: head = NULL, temp, newnode.
   - Set choice = 1.
2. **Create Linked List:**
   - While choice == 1:
     - Allocate memory: newnode = (struct node *)malloc(sizeof(struct node)).
     - Input newnode->data and set newnode->next = NULL.
     - If head == NULL:
       - Set head = newnode and temp = newnode.
     - Else:
       - Update temp->next = newnode and set temp = newnode.
3. **Display Linked List:**
   - Set temp = head.
   - While temp != NULL:
     - Print temp->data and move to the next node (temp = temp->next).
4. **Free Memory (Optional):**
   - Traverse the list, deallocating each node with free(temp).
5. **End Program.**

- head.

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

struct node* deleteFromEnd(struct node *head) {
    struct node *temp, *prev;
```

```
// Check if the list is empty
if (head == NULL) {
    printf("The list is already empty.\n");
    return head;
}

// If the list has only one node
if (head->next == NULL) {
    free(head);
    head = NULL;
    printf("The last node has been deleted. The list is now empty.\n");
    return head;
}

// Traverse the list to find the second-to-last node
temp = head;
while (temp->next != NULL) {
    prev = temp;
    temp = temp->next;
}

// Unlink the last node and free its memory
prev->next = NULL;
free(temp);
printf("The last node has been deleted.\n");

return head;
```

```
}
```

5. **Insert after a given position:**

Algorithm: Insert a Node After a Given Position

1. **Count the Nodes:**
   - Traverse the linked list to count the total number of nodes (count).
2. **Allocate Memory:**
   - Use malloc() to allocate memory for a new node (newnode).
   - If memory allocation fails, print an error message and return head.
3. **Input Position:**
   - Ask the user to input the position (pos) where the new node will be inserted.
4. **Validate Position:**
   - Check if pos is within the range 1 ≤ pos ≤ count.
   - If invalid, print an error message, free the allocated memory, and return head.
5. **Input Node Data:**
   - Prompt the user to input the data for the new node
   - (newnode->data).
6. **Insert the Node:**
   - **Case 1: At the End of the List**
     - Traverse to the last node.
     - Set newnode->next = NULL and link the last node's next pointer to newnode.
   - **Case 2: At a Specific Position**
     - Traverse to the node at position pos using a loop.
     - Update newnode->next to point to the next node.
     - Link the current node's next to newnode.
7. **Return Updated List:**
   - Return the updated head pointer.

```
// Define the structure of a node
struct node {
    int data;
    struct node *next;
};

// Function to insert a node after a given position
struct node* insertAfterLocation(struct node *head) {
    struct node *newnode, *temp;
    int pos, i = 1, count = 0;

    // Count total nodes in the list
    temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    // Allocate memory for the new node
    newnode = (struct node *)malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }

    // Get position input from user
    printf("Enter the position after which to insert the new node: ");
```

```
scanf("%d", &pos);

// Validate position
if (pos > count || pos < 0) {
    printf("Invalid position. Must be between 0 and %d.\n", count);
    free(newnode);
    return head;
}

// Get data input for the new node
printf("Enter data for the new node: ");
scanf("%d", &newnode->data);

// If inserting at the end
if (pos == count) {
    newnode->next = NULL;
    temp = head;

    // If list is empty
    if (head == NULL) {
        head = newnode;
    } else {
        while (temp->next != NULL) {
            temp = temp->next;
        }
        temp->next = newnode;
    }
} else {
```

```
        // Traverse to the given position
        temp = head;
        while (i < pos) {
            temp = temp->next;
            i++;
        }

        // Insert the new node
        newnode->next = temp->next;
        temp->next = newnode;
    }

    printf("Node inserted successfully.\n");
    return head;
}
```

6. **Delete from a spicified position:**

Algorithm: Delete Node at a Specific Position in Linked List

1. **Check if the List is Empty:**
   - If head == NULL, print "The list is empty" and return.
2. **Input the Position:**
   - Prompt the user to enter the position (pos) of the node to delete.
3. **Special Case: Deleting the First Node:**
   - If pos == 1:
     - Store the current head in a temporary pointer (temp).
     - Move head to the next node (head = head->next).
     - Free the memory of temp.
     - Print "Node at position 1 deleted."
     - Return the updated head.
4. **Traverse to the Node Before the Target Position:**

    - Initialize a pointer temp = head and a counter i = 1.
    - While i < pos - 1 and temp != NULL:
        - Move temp to the next node (temp = temp->next).
        - Increment i.
5. **Check for Invalid Position:**
    - If temp == NULL or temp->next == NULL, print "Invalid position" and return the unchanged head.
6. **Delete the Node:**
    - Store the target node (temp->next) in a pointer (nextnode).
    - Update the next pointer of temp to skip the target node (temp->next = nextnode->next).
    - Free the memory of nextnode.
    - Print "Node at position X deleted."
7. **Return Updated List:**
    - Return the updated head pointer.

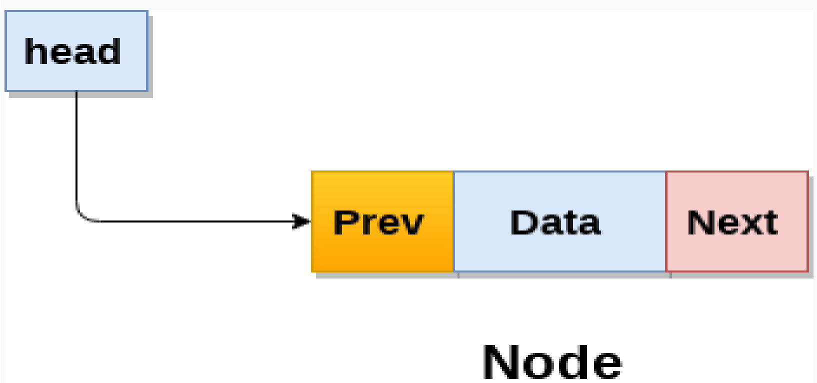
```
struct node {
    int data;
    struct node *next;
};

// Function to delete a node from a specific position in the linked list
struct node* deleteFromPosition(struct node *head) {
    struct node *temp, *prev, *nextnode;
    int pos, i = 1;

    // Check if the list is empty
    if (head == NULL) {
        printf("The list is empty.\n");
        return head;
    }
```

```
// Input the position to delete
printf("Enter the position to delete: ");
scanf("%d", &pos);

// If the position is the first node
if (pos == 1) {
    temp = head;
    head = head->next; // Move head to the next node
    free(temp); // Free the memory of the old head
    printf("Node at position 1 deleted.\n");
    return head;
}

temp = head;

// Traverse to the node just before the desired position
while (i < pos - 1 ) {
    temp = temp->next;
    i++;
}

// Delete the node at the given position
nextnode = temp->next;
temp->next = nextnode->next; // Unlink the node
free(nextnode); // Free the memory of the deleted node
printf("Node at position %d deleted.\n", pos);
```

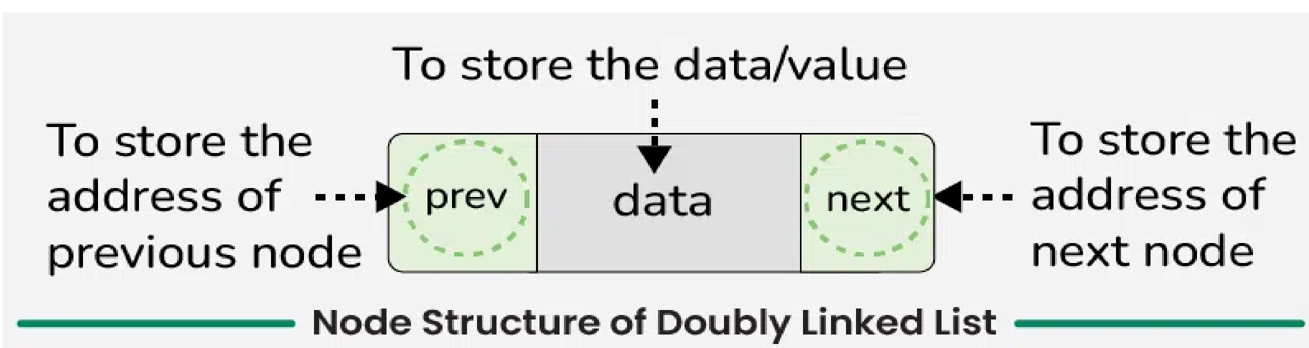
```
    return head;
}
```

Doubly linked list

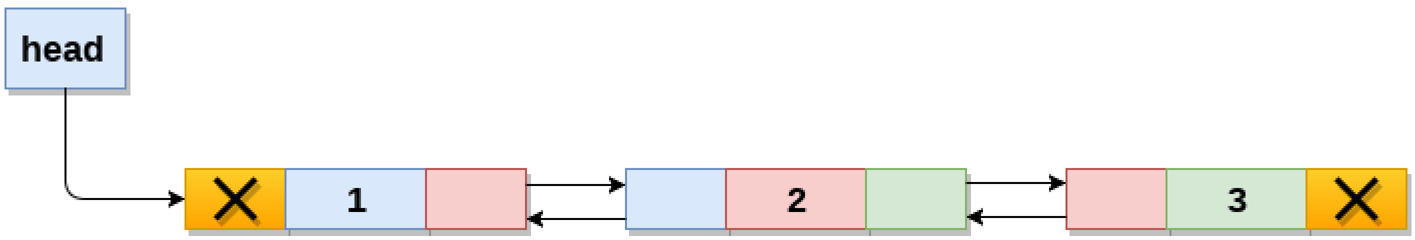
Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence. Therefore, in a doubly linked list, a node consists of three parts: node data, pointer to the next node in sequence (next pointer) , pointer to the previous node (previous pointer). A sample node in a doubly linked list is shown in the figure.



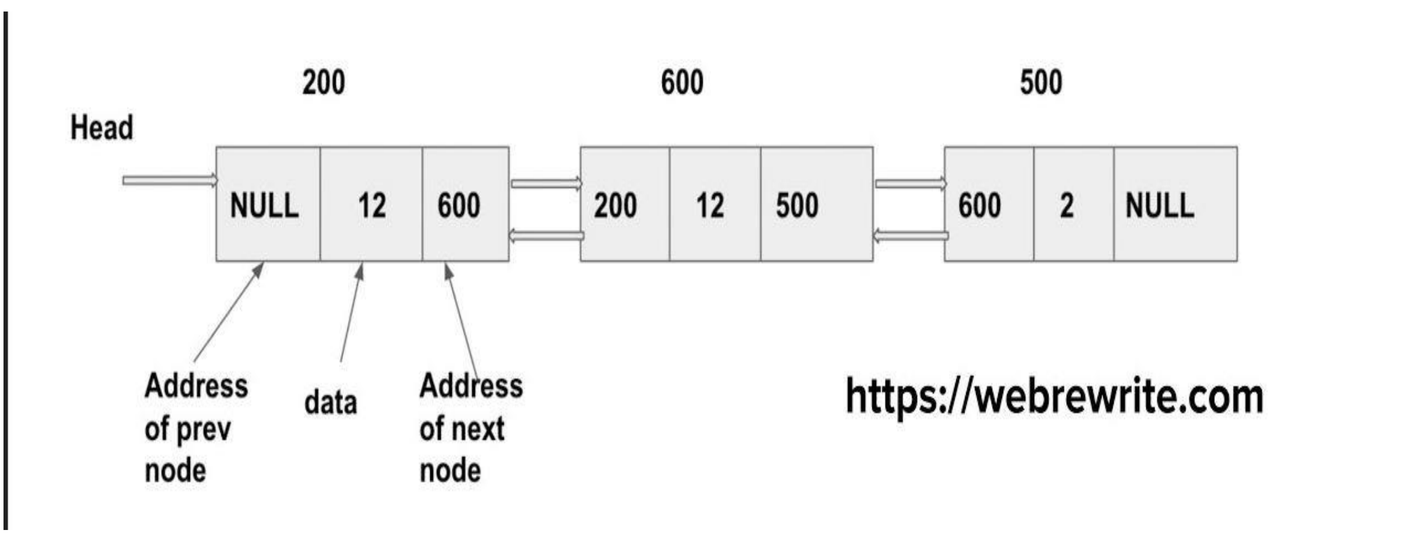
1. Data
2. A pointer to the next node (**next**)
3. A pointer to the previous node (**prev**)



A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List



In C, structure of a node in doubly linked list can be given as :

```
struct node
{
    int data;
    struct node *prev;
    struct node *next;
}
```

The **prev** part of the first node and the **next** part of the last node will always contain null indicating end in each direction.

Create and display doubly linked list:

Algorithm: Create a Doubly Linked List

1. **Initialize Variables**
   - Declare head as NULL (indicates the start of the list).
   - Declare temp for traversing and updating the list.
2. **Allocate Memory for a New Node**
   - Use malloc to dynamically allocate memory for the newnode.
3. **Input Data for the New Node**
   - Prompt the user to enter data for the node.
   - Store the user input in newnode->data.
4. **Set Pointers for the New Node**
   - Assign newnode->prev = NULL (as it's the first node or to be updated later).
   - Assign newnode->next = NULL (as it's the last node or to be updated later).
5. **Check If the List is Empty**
   - If head == NULL:
     - Assign head = temp = newnode.
     - This sets the first node as both the head and the current pointer.
   - Else:
     - Update temp->next to point to newnode.
     - Update newnode->prev to point to temp.
     - Move temp to the newnode (temp = newnode).
6. **Repeat Steps 2–5 for Additional Nodes**
   - Continue creating nodes and linking them until the user decides to stop (loop controlled externally).
7. **Return the Head of the List**
   - The function returns head, which points to the first node of the doubly linked list.

```
struct node {
   int data;
```

```
    struct node* next;
    struct node* prev;
};
struct node *head, *newnode, *temp;
void create() {   // Create new node
        head = 0;

        newnode = (struct node *)malloc(sizeof(structnode);
        printf("Enter data.");
        scanf("%d", &newnode -> data);
        newnode ->prev = 0;
        newnode -> next = 0;

        if(head == 0)
        {
                head = temp = newnode;
        }
        else
        {
                temp -> next = newnode;
                newnode -> prev = temp;
                temp = newnode;
        }
        return head;
}
```

1. Insertion at the beginning of the doubly linked list:

Algorithm: Insert Node at the Beginning of a Doubly Linked List

1. **Initialize Variables**
   - head points to the current first node of the list (if any).
2. **Create a New Node**
   - Dynamically allocate memory for newnode using malloc.
   - Input data for the new node (newnode->data).
   - Set newnode->prev = NULL (as it will be the new first node).
   - Set newnode->next = NULL initially.
3. **Check if the List is Empty**
   - If head == NULL (list is empty):
     - Assign head = newnode.
   - Else:
     - Set newnode->next = head (link newnode to the current first node).
     - Update head->prev = newnode (link current first node back to the newnode).
     - Update head = newnode (make newnode the new first node).
4. **End the Function**
   - Return the head pointer, now pointing to the updated first node of the doubly linked list.

```
struct node {
   int data;
   struct node* next;
   struct node* prev;
};
struct node *head, *newnode, *temp;
void create() {   // Create new node
       head = 0;
```

```
        newnode = (struct node *)malloc(sizeof(structnode);
        printf("Enter data.");
        scanf("%d", &newnode -> data);
        newnode ->prev = 0;
        newnode -> next = 0;

        if(head == 0)
        {
                head = temp = newnode;
        }
        else
        {
                head -> prev = newnode;
                temp -> next = head;
                head = newnode;
        }
        return head;
}
```

2. Delete from the beginning:

Algorithm: Delete a Node from the Beginning of a Doubly Linked List

1. **Start**
2. **Check if the list is empty**:
   - If head == NULL, print "The list is empty. Nothing to delete" and exit.
3. **Store the current head node**:
   - Assign temp = head.
4. **Update the head pointer**:
   - Move head to the next node (head = head->next).

5. **Check if the updated head is not NULL**:
    - If head != NULL, set head->prev = NULL to remove the backward link to the deleted node.
6. **Free the memory of the deleted node**:
    - Use free(temp) to release the memory occupied by the node.
7. **Print a confirmation message**:
    - Print "Node deleted from the beginning."
8. **Stop**.

```
struct node {
    int data;
    struct node* next;
    struct node* prev;
};

struct node *head, *temp;
head = NULL; // Head pointer for the doubly linked list

// Function to delete a node from the beginning
void deleteFromBeginning() {

    // Check if the list is empty
    if (head == NULL) {
        printf("The list is empty. Nothing to delete.\n");
        return;
    }

    // Point temp to the head
    temp = head;
```

```
    // Update head to the next node
    head = head->next;

    // If the list has more than one node
    if (head != NULL) {
        head->prev = NULL; // Remove the backward link to the old head
    }

    // Free the memory of the deleted node
    free(temp);

    printf("Node deleted from the beginning.\n");
}
```

3.Insert at the end of the doubly linked list

Algorithm: Insert Node at the End of a Doubly Linked List

1. **Initialize Variables**
   - head points to the first node of the list (if any).
   - tail points to the last node of the list (if any).
2. **Create a New Node**
   - Dynamically allocate memory for newnode using malloc.
   - Input data for the new node (newnode->data).
   - Set newnode->next = NULL (as it will be the new last node).
   - Set newnode->prev = NULL initially.
3. **Check if the List is Empty**
   - If head == NULL (list is empty):
     - Assign head = newnode and tail = newnode (newnode becomes the only node in the list).
   - Else:
     - Set tail->next = newnode (link the current last node to the newnode).
     - Set newnode->prev = tail (link newnode back to the current last node).

            ▪ Update tail = newnode (make newnode the new last node).

4. **End the Function**
   - Return the head pointer, which points to the first node of the doubly linked list.

```
struct node {
   int data;
   struct node* next;
   struct node* prev;
};
struct node *head, *tail *newnode;
void create() {   // Create new node
        head = 0;

        newnode = (struct node *)malloc(sizeof(structnode);
        printf("Enter data.");
        scanf("%d", &newnode -> data);
        newnode ->prev = 0;
        newnode -> next = 0;

        if(head == 0)
        {
                head = tail = newnode;
        }
        else
        {
```

```
        tail -> next = newnode;
        newnode -> prev = tail;
        tail = newnode;
    }
    return head;
}
```

4.Delete from the end:

Algorithm to Delete a Node from the End of a Doubly Linked List

1. **Check if the list is empty**:
   - If head == NULL or tail == NULL, print "The list is empty. Nothing to delete."
   - Exit the function.
2. **Store the last node**:
   - Assign temp = tail.
3. **Update the tail pointer**:
   - Set tail = tail->prev.
4. **Unlink the last node**:
   - If tail != NULL, set tail->next = NULL.
5. **Free the memory of the last node**:
   - Use free(temp) to release the memory.
6. **If the list becomes empty**:
   - If tail == NULL, also set head = NULL.
7. **End the function**.

```
struct node {
  int data;
  struct node *next;
  struct node *prev;
};
struct node *temp, *tail;
```

```
// Function to delete a node from the end of the list
void deleteFromEnd() {
    if (head == NULL || tail == NULL) { // Check if the list is empty
        printf("The list is empty. Nothing to delete.\n");
        return;
    }

    else {
        temp = tail;
        tail -> prev -> next = 0;
        tail = tail -> prev;
        free(temp);
    }
}
```

5.Insert at a position in doubly linked list:

Algorithm to Insert a Node After a Given Position in a Doubly Linked List

1. **Input the position**:
   - Prompt the user to enter the position where the new node will be inserted.
2. **Check if the list is empty**:
   - If head == NULL, print "The list is empty."
   - Exit the function.
3. **Count the total number of nodes in the list**:
   - Traverse the list to calculate the total number of nodes.
4. **Validate the position**:
   - If the position is less than 1 or greater than the total number of nodes, print "Invalid position."
   - Exit the function.
5. **Allocate memory for the new node**:
   - Create a new node using malloc.
   - Input data for the new node.

    - Set newnode->next = NULL and newnode->prev = NULL.
6. **Traverse to the desired position**:
    - Start from head and traverse the list until reaching the node just before the specified position (pos - 1).
7. **Insert the new node**:
    - Set newnode->prev = temp.
    - Set newnode->next = temp->next.
    - Update temp->next to point to the new node.
    - Update newnode->next->prev to point to the new node (if it exists).
8. **End the function**.

```
struct node {
    int data;
    struct node *next;
    struct node *prev;
};

// Function to insert an element after a given position in the linked list
void insertAtPos(int pos) {
    struct node *newnode, *temp;
    int pos, i = 1, count = 0;
    // Input the position
    printf("Enter the position: ");
    scanf("%d", &pos);


    // Calculate the total number of nodes in the list
    temp = head;
    while (temp != NULL) {
```

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

// Validate the position
if (pos > count || pos <1 ) {
    printf("Invalid position.\n");
    free(newnode); // Free the allocated memory for the new node
    return head;
}
else if (pos == 1) {
     insertAtBeginning();
}
// Insert at the correct position

} else {

// Allocate memory for the new node
    newnode = (struct node *)malloc(sizeof(struct node);
    printf("Enter data.");
    scanf("%d", &newnode -> data);
    newnode -> next = 0;
    newnode -> prev = 0;
    while (  i< pos -1)
     {
       temp = temp->next;
       i++;
    }
```

```
        // Insert the new node
        newnode->prev = temp;
        newnode -> next = temp -> next;
        temp -> next = newnode;
        newnode -> next -> prev = newnode;
    }

    return head;
}
```

Introduction to Circular Linked List

-

A **circular linked list** is a data structure where the last node connects back to the first, forming a loop. This structure allows for continuous traversal without any interruptions. Circular linked lists are especially helpful for tasks like **scheduling** and **managing playlists,** this allowing for smooth navigation. In this tutorial, we'll cover the basics of circular linked lists, how to work with them, their advantages and disadvantages, and their applications.

What is a Circular Linked List?

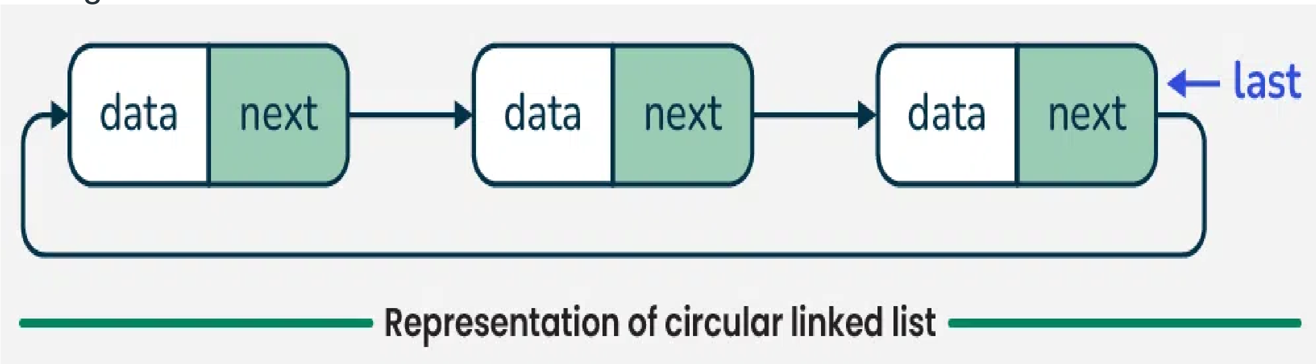
A **circular linked list** is a special type of linked list where all the nodes are connected to form a circle. Unlike a regular linked list, which ends with a node pointing to **NULL**, the last node in a circular linked list points back to the first node. This means that you can keep traversing the list without ever reaching a **NULL** value.

Types of Circular Linked Lists

We can create a circular linked list from both singly linked lists and doubly linked lists. So, circular linked list are basically of two types:

1. Circular Singly Linked List

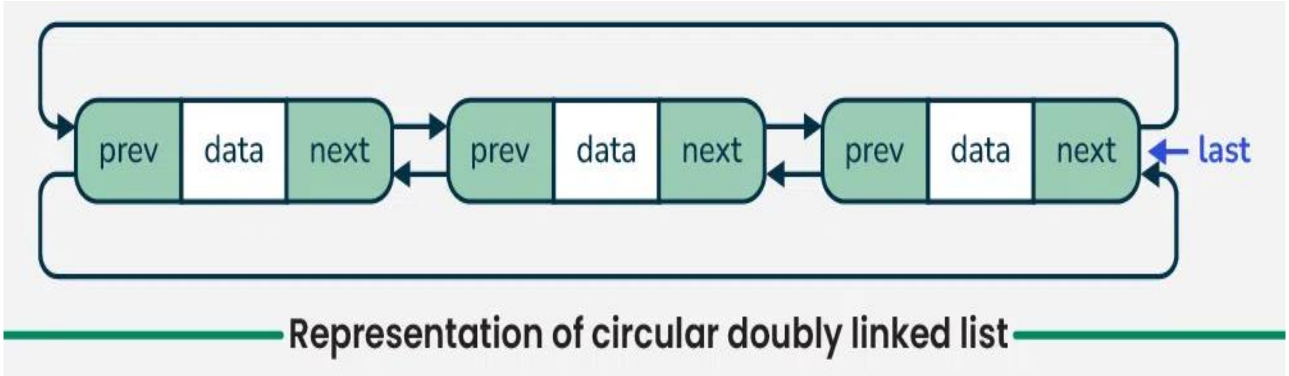
In **Circular Singly Linked List**, each node has just one pointer called the "**next**" pointer. The next pointer of **last node** points back to the **first node** and this results in forming a circle. In this type of Linked list we can only move through the list in one direction.



Representation of Circular Singly Linked List

2. Circular Doubly Linked List:

In **circular doubly linked list,** each node has two pointers **prev** and **next,** similar to doubly linked list. The **prev** pointer points to the previous node and the **next** points to the next node. Here, in addition to the **last** node storing the address of the first node, the **first node** will also store the address of the **last node**.



Representation of Circular Doubly Linked List

Note: In this article, we will use the circular singly linked list to explain the working of circular linked lists.

Insertion in Circular Linked List:

- Insert at the beginning
- Insert at end
- Insert at a position

Create and display circular linked list

Algorithm: Create a Circular Linked List

1. **Initialize Variables**:
   - Declare a pointer head and set it to NULL (represents the start of the linked list).
   - Declare pointers newnode (for the newly created node) and tail (to track the end of the list).
2. **Input Node Creation**:
   - Allocate memory for the new node using malloc().
   - Check if the memory allocation was successful:
     - If not, print an error message: "Memory allocation failed."
     - Exit the function.
3. **Input Data**:
   - Prompt the user to enter data for the new node: "Enter data:".
   - Store the input in newnode->data.
   - Set newnode->next to NULL.
4. **Insert Node into the List**:
   - Check if head is NULL:
     - If head is NULL (list is empty):
       - Set both head and tail to newnode.
     - Otherwise:
       - Link the current tail node to the newnode by setting tail->next = newnode.
       - Update tail to newnode.

5. **Make the List Circular**:
    - After inserting the node, set tail->next to head to complete the circular linkage.
6. **End**:
    - The linked list is created, and it is circular with tail->next pointing to head.

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

// Function to create the linked list
Void create() {
    struct node *head, *newnode, *tail;
    head = NULL;
        // Allocate memory for the new node
        newnode = (struct node *)malloc(sizeof(struct node));
        if (newnode == NULL) {
            printf("Memory allocation failed.\n");
            return head;
        }
```

```
        // Input data for the new node

        printf("Enter data: ");

        scanf("%d", &newnode->data);

        newnode->next = NULL;


        // Add the new node to the list

        if (head == NULL) {

            head = tail = newnode; // First node

        } else {

            tail->next = newnode; // Add at the end

            tail = newnode;       // Update tail

        }

      tail -> next = head;

}
```

1. Insert at the beginning

Algorithm: Insert at Beginning in a Circular Linked List

1. **Initialize Variables**:
   - Declare head, newnode, and tail. Assume tail points to the last node, and tail->next is the head of the circular linked list.
2. **Allocate Memory for New Node**:
   - Allocate memory for newnode using malloc().

    - If memory allocation fails, print "Memory allocation failed" and exit the function.
3. **Input Data**:
    - Prompt the user to enter data.
    - Store the input in newnode->data.
    - Set newnode->next to NULL.
4. **Insert New Node**:
    - If tail is NULL (list is empty):
        - Set newnode->next to itself (it points to itself, forming a single-node circular list).
        - Update tail to newnode.
    - Else (list is non-empty):
        - Set newnode->next to tail->next (current head).
        - Update tail->next to newnode (newnode becomes the new head).
5. **End**:
    - The new node is inserted at the beginning, and the circular linkage is maintained.

```
struct node {

   int data;

   struct node *next;

};



// Function to create the linked list

Void insertAtBegining() {

   struct node *head, *newnode, *tail;

   head = NULL;

      // Allocate memory for the new node
```

```c
    newnode = (struct node *)malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }

    // Input data for the new node
    printf("Enter data: ");
    scanf("%d", &newnode->data);
    newnode->next = NULL;

    // Add the new node to the list
    if (tail == NULL) {
        tail -> next = newnode;
        tail -> next = newnode;
    } else {
        newnode ->next = tail -> next;
        tail -> next = newnode;
    }
}
```

2. Delete from the beginning

Algorithm: Delete from Beginning in a Circular Linked List

1. **Check if List is Empty**:
   - If tail == NULL, print "List is empty" and exit the function.
2. **Handle Single Node Case**:
   - If the list contains only one node (tail->next == tail):
     - Set tail to NULL (empty the list).
     - Free the memory of the single node.
3. **Delete the First Node**:
   - Otherwise:
     - Set temp to tail->next (current head).
     - Update tail->next to temp->next (the next node becomes the new head).
     - Free the memory of temp.
4. **End**:
   - The first node is successfully deleted, and the circular linkage is maintained.

```
struct node {

   int data;

   struct node *next;

};



// Function to create the linked list

Void deleteFromBegining() {

   struct node *temp,  *tail;

   temp = tail -> next;
```

```
        if (tail == NULL) {

         print("List is empty");

        } else if( temp -> next = = temp)  // if only one node

          {

            tail = 0;

            free(temp);

         }

        Else

         {

            tail - > next = temp -> next;

            free(temp);

    }

    }
```

3. Insert at the end:

Algorithm: Insert at the End in a Circular Linked List

1. **Initialize Variables**:
   - Declare pointers head, newnode, and tail. Assume tail points to the last node, and tail->next is the head of the circular linked list.
2. **Allocate Memory for the New Node**:

- Allocate memory for newnode using malloc().
    - If memory allocation fails, print "Memory allocation failed" and exit the function.
3. **Input Data**:
    - Prompt the user to enter data.
    - Store the input in newnode->data.
    - Set newnode->next to NULL.
4. **Insert New Node**:
    - If tail == NULL (list is empty):
        - Set newnode->next to itself (it points to itself, forming a single-node circular list).
        - Update tail to newnode.
    - Otherwise (list is non-empty):
        - Set newnode->next to tail->next (current head).
        - Update tail->next to newnode (add the new node at the end).
        - Update tail to newnode (newnode becomes the new tail).
5. **End**:
    - The new node is inserted at the end, and the circular linkage is maintained.

```
struct node {

    int data;

    struct node *next;

};



// Function to create the linked list

Void insertAtEnd() {

    struct node *head, *newnode, *tail;

    head = NULL;
```

```
// Allocate memory for the new node
newnode = (struct node *)malloc(sizeof(struct node));
if (newnode == NULL) {
    printf("Memory allocation failed.\n");
    return head;
}

// Input data for the new node
printf("Enter data: ");
scanf("%d", &newnode->data);
newnode->next = NULL;

// Add the new node to the list
if (tail == NULL) {
    tail = newnode;
    tail -> next = newnode;
} else {
    newnode ->next = tail -> next;
    tail -> next = newnode;
    tail = newnode;
}
```

}

4. Delete from end of the list

Algorithm: Delete from End in a Circular Linked List

1. **Check if the List is Empty**:
   - If tail == NULL, print "List is empty" and exit the function.
2. **Check if the List Contains Only One Node**:
   - If current->next == current (only one node in the list):
     - Set tail = NULL (make the list empty).
     - Free the memory allocated to the node.
3. **Traverse the List to Find the Second Last Node**:
   - Initialize current = tail->next (start at the head).
   - Use a while loop to traverse the list until current->next != tail->next (stop at the last node).
     - Inside the loop, update previous = current and move current = current->next.
4. **Delete the Last Node**:
   - Update previous->next = tail->next (link the second last node to the head).
   - Set tail = previous (update the tail pointer to the second last node).
   - Free the memory allocated to current (the original last node).
5. **End**:
   - The last node is removed, and the list remains circular. If only one node was present, the list is now empty.

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

```
// Function to create the linked list
Void deleteFromEnd() {
    struct node *current, *previous,  *tail;
    current = tail -> next;

        if (tail == NULL) {
         print(“List is empty”);
        } else if( current -> next = = current)  // if only one node
          {
            tail = 0;
            free(temp);
         }
        else
         {
            while(current -> next != tail -> next)
           {
                  previous = current  -> next;
                  current = current -> next;
            }
                  previous - > next = tail -> next;
                  tail = previous;
```

```
                free(current);
        }
}
```

5. Insert at given position:

Algorithm: Insert at a Given Position in a Circular Linked List

1. **Count the Total Nodes**:
    - Initialize count = 0 and temp = head (or tail->next for circular lists).
    - Traverse the list to count the total nodes until temp == head again.
2. **Allocate Memory for the New Node**:
    - Allocate memory for newnode using malloc().
    - If memory allocation fails, print "Memory allocation failed" and exit the function.
3. **Input Position and Validate**:
    - Prompt the user to input the position pos.
    - If pos > count + 1, print "Invalid position" and free the allocated memory for newnode.
4. **Input Data for the New Node**:
    - Prompt the user to input the data for newnode.
    - Set newnode->next to NULL.
5. **Insert at the Beginning if Position is 1**:
    - If pos == 1, call the insertAtBegining() function to handle insertion at the beginning.
6. **Insert at a Given Position**:
    - Initialize temp = tail->next (current head) and i = 1.
    - Traverse the list until you reach the (pos-1)th node.
        - Update temp to temp->next and increment i.
    - Update newnode->next to temp->next.
    - Set temp->next to newnode to insert the new node.
7. **End**:
    - The new node is inserted at the given position, and the circular linkage is maintained.

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

// Function to insert an element after a given position in the linked list
Void insertAtPos() {
    struct node *newnode, *temp, *tail;
    int pos, i = 1, count = 0;

    // Calculate the total number of nodes in the list
    temp = head;
    while (temp != NULL) {
        count++;
        temp = temp->next;
    }

    // Allocate memory for the new node
    newnode = (struct node *)malloc(sizeof(struct node));
    if (newnode == NULL) {
        printf("Memory allocation failed.\n");
        return head;
    }

    // Input the position
    printf("Enter the position: ");
    scanf("%d", &pos);
```

}

```
// Validate the position
if (pos > count ) {
    printf("Invalid position.\n");
    free(newnode); // Free the allocated memory for the new node
    return head;
}

// Input data for the new node
printf("Enter data: ");
scanf("%d", &newnode->data);

// Insert at the correct position
if (pos ==1)
{
     insertAtBeg();

} else {
     newnode -> next = 0;
    temp = tail -> next;
    while (i < pos-1) {
        temp = temp->next;
        i++;
    }

    // Insert the new node
    newnode->next = temp->next;
    temp->next = newnode;
}
```

```
    return head;
}
```

Tutorial

1. Delete node from a given position of doubly linked list and circular linked list.
2. Implementation of stack and queue using linked list.