A **singly linked list** is the most simple type of linked list, with each node containing some data as well as a pointer to the next node. That is a singly linked list allows traversal of data only in one way. There are several linked list operations that allow us to perform different tasks.

The basic linked list operations are:

- Traversal – Access the nodes of the list.
- Insertion – Adds a new node to an existing linked list.
- Deletion – Removes a node from an existing linked list.
- Search – Finds a particular element in the linked list.

Traverse a Linked List

Accessing the nodes of a linked list in order to process it is called **traversing** a linked list. Normally we use the traverse operation to display the contents or to search for an element in the linked list. The algorithm for traversing a linked list is given below.

Algorithm: Traverse

```
Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: Apply process to PTR -> DATA
Step 4: SET PTR = PTR->NEXT
[END OF LOOP]
Step 5: EXIT
```

- We first initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.
- A while loop is executed, and the operation is continued until PTR reaches the last node (PTR = NULL).
- Apply the process(display) to the current node.
- Move to the next node by making the value of PTR to the address of next node.

The following block of code prints all elements in a linked list in C.

```
struct node *ptr = head;
printf("Elements in the linked list are : ");
while (ptr != NULL)
{
printf("%d ", ptr->data);
ptr = ptr->next;
}
```

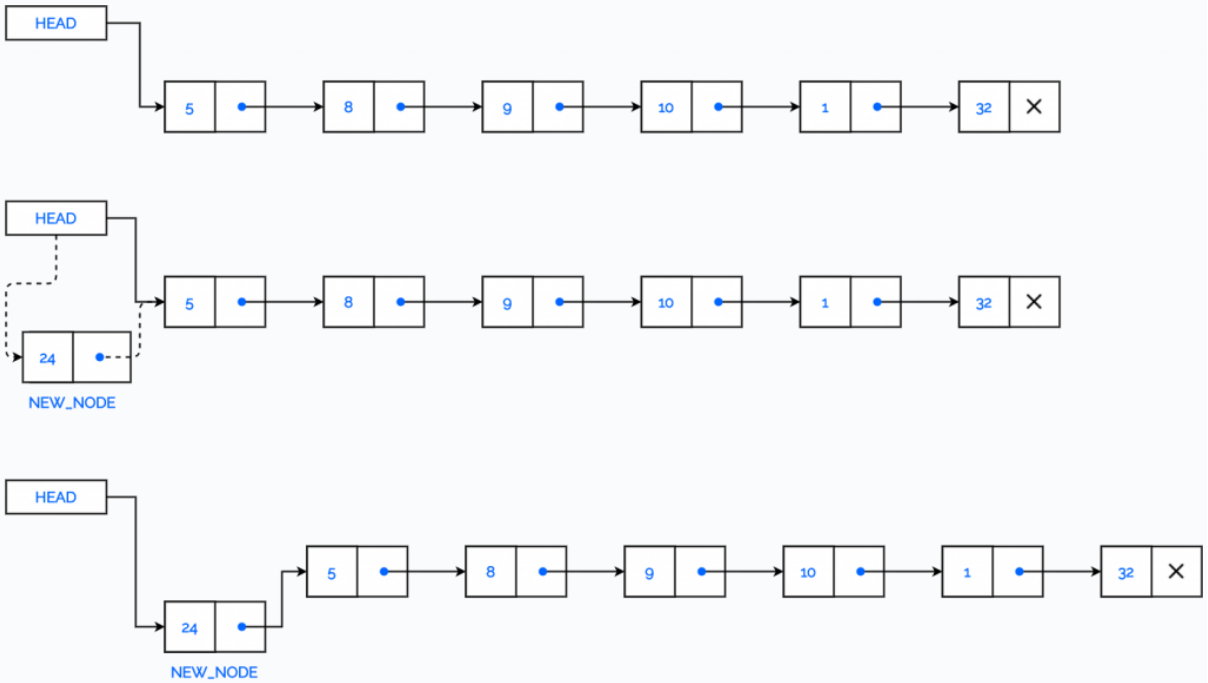
Inserting Elements to a Linked List

We will see how a new node can be added to an existing linked list in the following cases.

1. The new node is inserted at the beginning.
2. The new node is inserted at the end.
3. The new node is inserted after a given node.

Insert a Node at the beginning of a Linked list

Consider the linked list shown in the figure. Suppose we want to create a new node with data 24 and add it as the first node of the list. The linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Add the new node as the first node of the list by pointing the NEXT part of the new node to HEAD.
- Make HEAD to point to the first node of the list.

Algorithm: InsertAtBeginning

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 7
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = HEAD
Step 6: SET HEAD = NEW_NODE
Step 7: EXIT
```

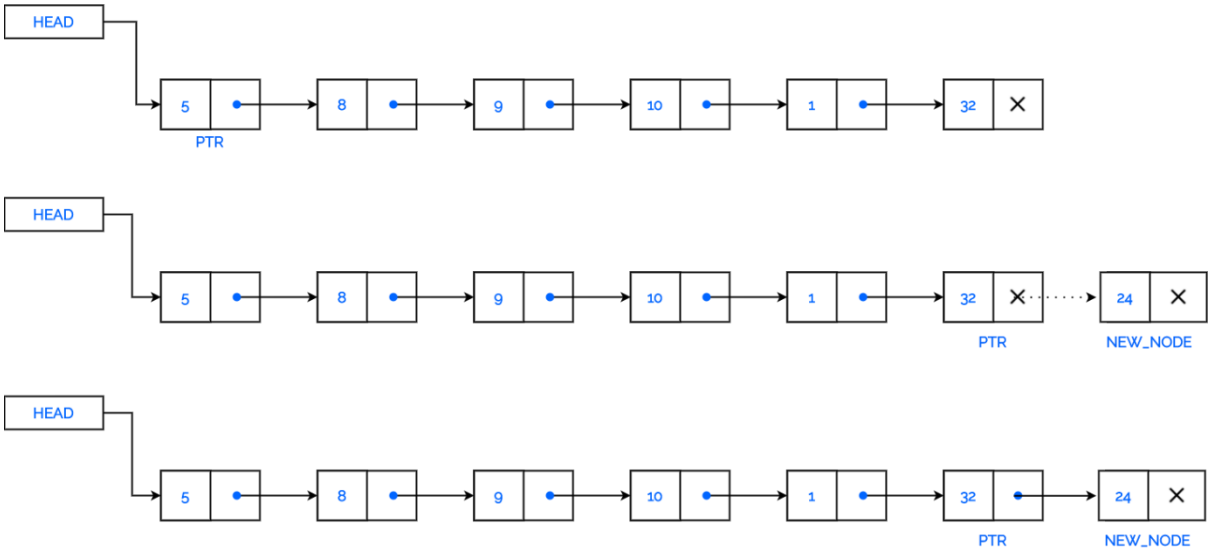
Note that the first step of the algorithm checks if there is enough memory available to create a new node. The second, and third steps allocate memory for the new node.

This algorithm can be implemented in C as follows:

```
struct node *new_node;
new_node = (struct node*) malloc(sizeof(struct node));
new_node->data = 24;
new_node->next = head;
head = new_node;
```

Insert a Node at the end of a Linked list

Take a look at the linked list in the figure. Suppose we want to add a new node with data 24 as the last node of the list. Then the linked list will be modified as follows.



- Allocate memory for new node and initialize its DATA part to 24.
- Traverse to last node.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to NULL.

Algorithm: InsertAtEnd

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 10
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET NEW_NODE -> NEXT = NULL
```

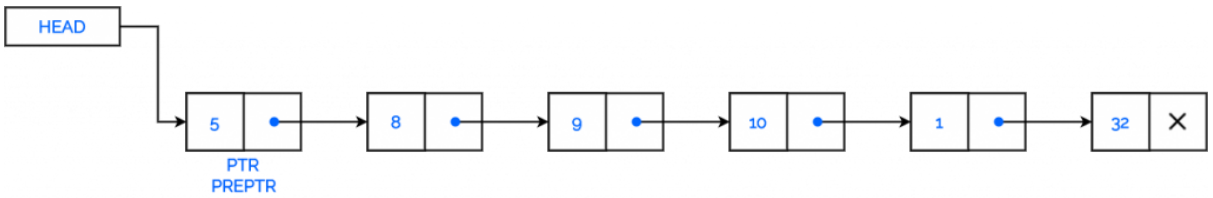
```
Step 6: SET PTR = HEAD
Step 7: Repeat Step 8 while PTR -> NEXT != NULL
Step 8: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 9: SET PTR -> NEXT = NEW_NODE
Step 10: EXIT
```

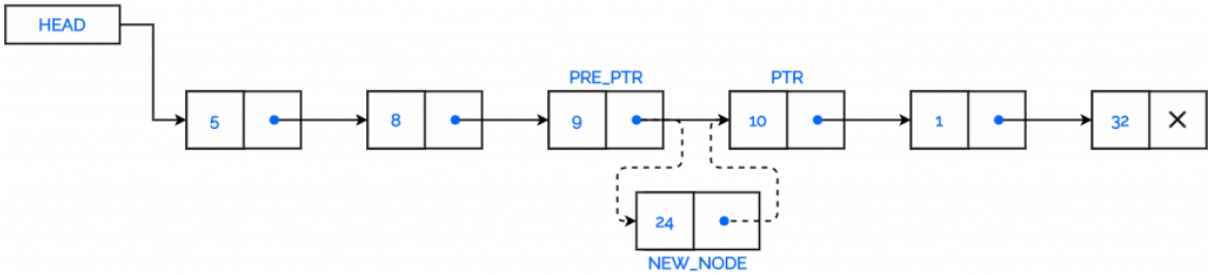
This can be implemented in C as follows,

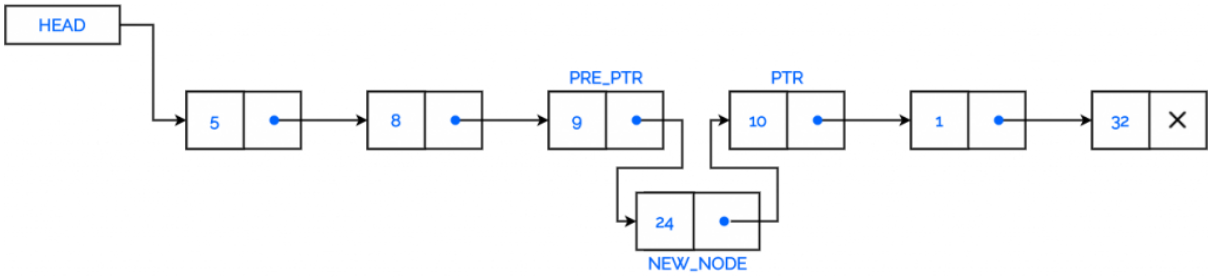
```
struct node *new_node;
new_node = (struct node*) malloc(sizeof(struct node));
new_node->data = 24;
new_node->next = NULL;
struct node *ptr = head;
while(ptr->next != NULL){
ptr = ptr->next;
}
ptr->next = new_node;
```

Insert a Node after a given Node in a Linked list

The last case is when we want to add a new node after a given node. Suppose we want to add a new node with value 24 after the node having data 9. These changes will be done in the linked list.







- Allocate memory for new node and initialize its DATA part to 24.
- Traverse the list until the specified node is reached.
- Change NEXT pointers accordingly.

Algorithm: InsertAfterAnElement

```
Step 1: IF AVAIL = NULL
Write OVERFLOW
Go to Step 12
[END OF IF]
Step 2: SET NEW_NODE = AVAIL
Step 3: SET AVAIL = AVAIL -> NEXT
Step 4: SET NEW_NODE -> DATA = VAL
Step 5: SET PTR = HEAD
Step 6: SET PREPTR = PTR
Step 7: Repeat Steps 8 and 9 while PREPTR -> DATA != NUM
Step 8: SET PREPTR = PTR
Step 9: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 1 : PREPTR -> NEXT = NEW_NODE
Step 11: SET NEW_NODE -> NEXT = PTR
Step 12: EXIT
```

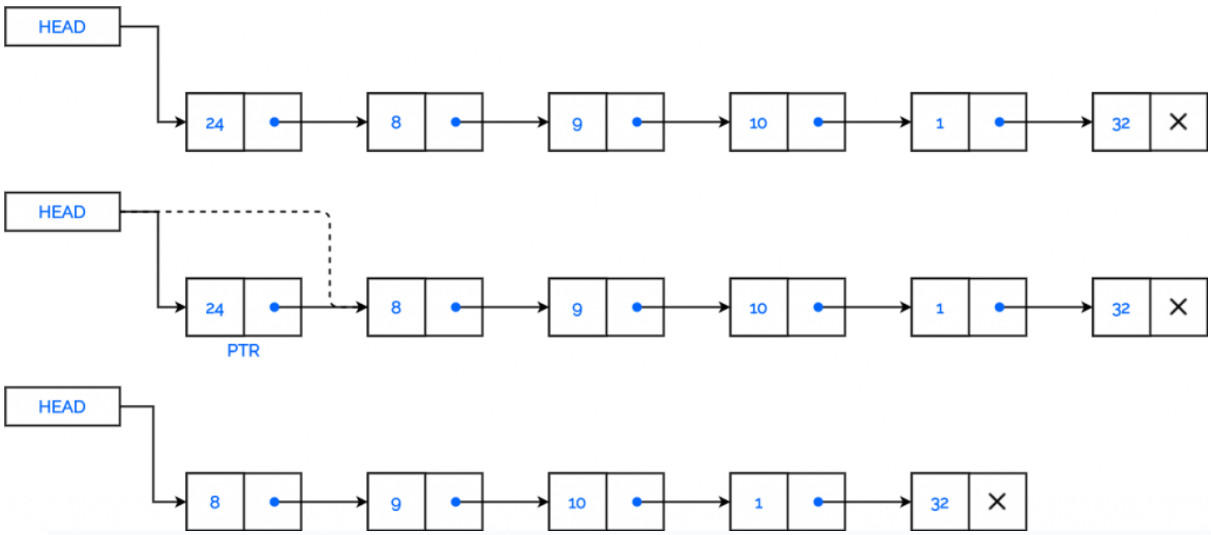
Deleting Elements from a Linked List

Let’s discuss how a node can be deleted from a linked listed in the following cases.

1. The first node is deleted.
2. The last node is deleted.
3. The node after a given node is deleted.

Delete a Node from the beginning of a Linked list

Suppose we want to delete a node from the beginning of the linked list. The list has to be modified as follows:



- Check if the linked list is empty or not. Exit if the list is empty.
- Make HEAD points to the second node.
- Free the first node from memory.

Algorithm: DeleteFromBeginning

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 5
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET HEAD = HEAD -> NEXT
Step 4: FREE PTR
Step 5: EXIT
```

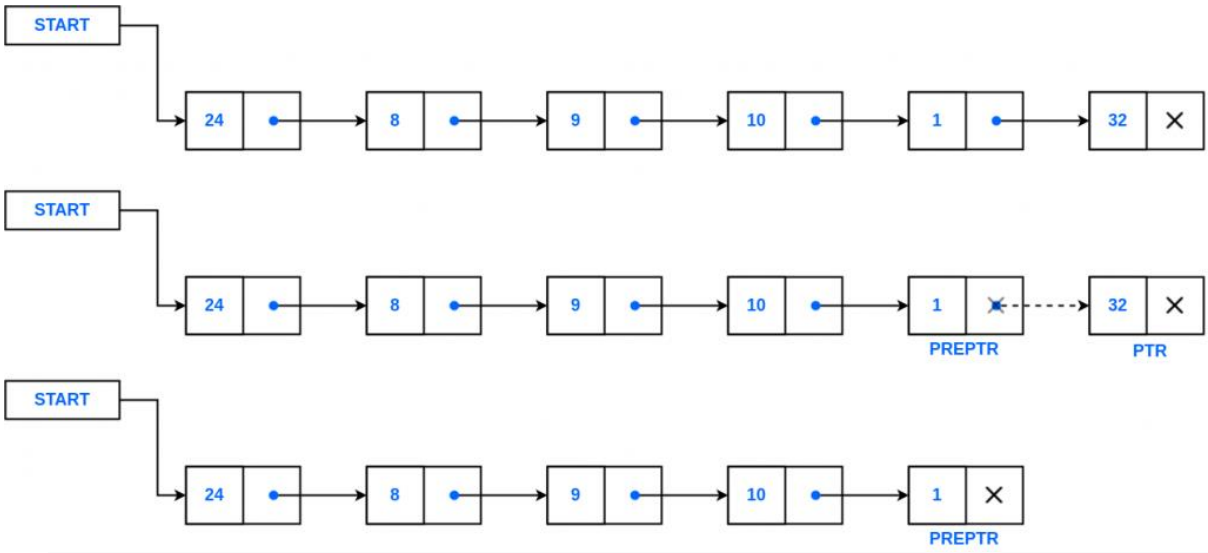
This can be implemented in C as follows,

```
if(head == NULL)
{
printf("Underflow");
}
else
```

```
{
ptr = head;
head = head -> next;
free(ptr);
}
```

Delete last Node from a Linked list

Suppose we want to delete the last node from the linked list. The linked list has to be modified as follows:



- Traverse to the end of the list.
- Change value of next pointer of second last node to NULL.
- Free last node from memory.

Algorithm: DeleteFromEnd

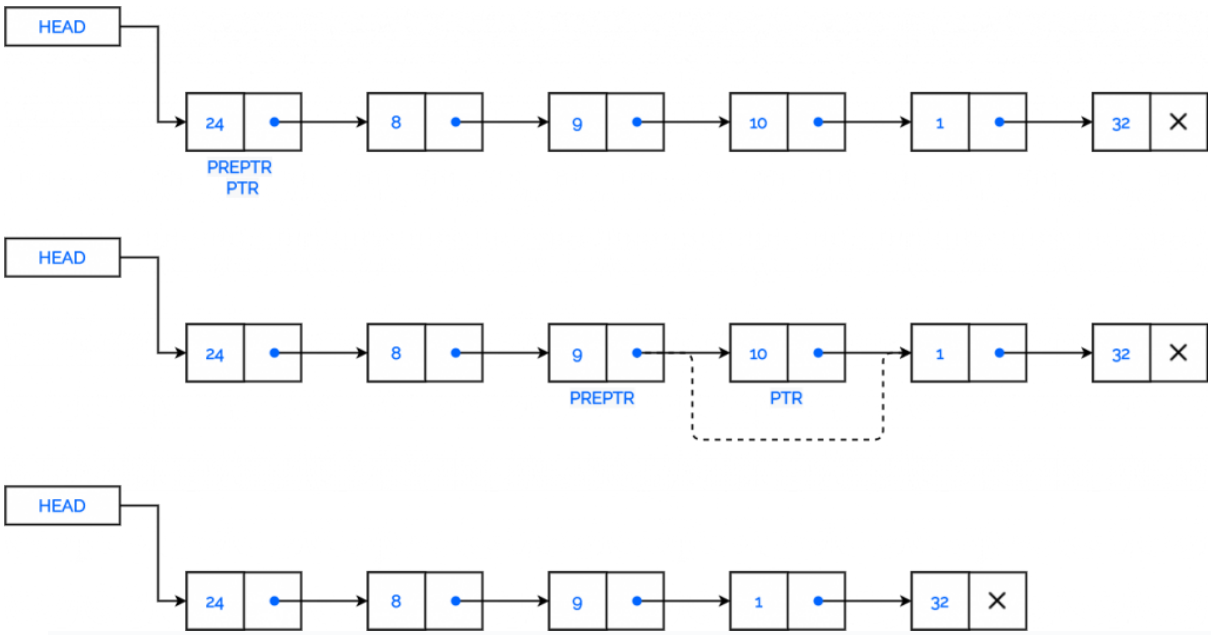
```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 8
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: Repeat Steps 4 and 5 while PTR -> NEXT != NULL
Step 4: SET PREPTR = PTR
Step 5: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 6: SET PREPTR -> NEXT = NULL
Step 7: FREE PTR
Step 8: EXIT
```

Here we use two pointers PTR and PREPTR to access the last node and the second last node. This can be implemented in C as follows,

```
if(head == NULL)
{
printf("Underflow");
}
else
{
struct node* ptr = head;
struct node* preptr = NULL;
while(ptr->next!=NULL){
preptr = ptr;
ptr = ptr->next;
}
preptr->next = NULL;
free(ptr);
}
```

Delete the Node after a given Node in a Linked list

Suppose we want to delete the that comes after the node which contains data 9.



- Traverse the list upto the specified node.
- Change value of next pointer of previous node(9) to next pointer of current node(10).

Algorithm: DeleteAfterANode

```
Step 1: IF HEAD = NULL
Write UNDERFLOW
Go to Step 10
[END OF IF]
Step 2: SET PTR = HEAD
Step 3: SET PREPTR = PTR
Step 4: Repeat Steps 5 and 6 while PREPTR -> DATA != NUM
Step 5: SET PREPTR = PTR
Step 6: SET PTR = PTR -> NEXT
[END OF LOOP]
Step 7: SET TEMP = PTR
Step 8: SET PREPTR -> NEXT = PTR -> NEXT
Step 9: FREE TEMP
Step 10 : EXIT
```

Implementation in C takes the following form:

```
if(head == NULL)
{
printf("Underflow");
}
else
{
struct node* ptr = head;
struct node* preptr = ptr;
while(ptr->data!=num){
preptr = ptr;
ptr = ptr->next;
}
struct node* temp = ptr;
preptr -> next = ptr -> next;
free(temp);
}
```

Search

Finding an element is similar to a traversal operation. Instead of displaying data, we have to check whether the data matches with the **item** to find.

- Initialize PTR with the address of HEAD. Now the PTR points to the first node of the linked list.

- A while loop is executed which will compare data of every node with item.
- If item has been found then control goes to last step.

Algorithm: Search

```
Step 1: [INITIALIZE] SET PTR = HEAD
Step 2: Repeat Steps 3 and 4 while PTR != NULL
Step 3: If ITEM = PTR -> DATA
SET POS = PTR
Go To Step 5
ELSE
SET PTR = PTR -> NEXT
[END OF IF]
[END OF LOOP]
Step 4: SET POS = NULL
Step 5: EXIT
```

Search operation can be implemented in C as follows:

```
struct node* ptr = head;
struct node* pos = NULL;
while (ptr != NULL) {
if (ptr->data == item)
pos = ptr
printf("Element Found");
break;
else
ptr = ptr -> next;
}
```

Doubly Linked List

A doubly linked list or two way linked list is a type of linked list that contains a pointer to the **next** node as well as the **previous** node in the sequence.

A doubly linked list node

That is, each node in a doubly-linked list consists of:

- `data` – Data Item.
- `prev` – Address of previous node.
- `next` – Address of next node.

Representation of Doubly Linked List

In C, we can represent a doubly linked list node using **structures** as:

```
struct node

{

struct node *prev;

int data;

struct node *next;

};
```

Each struct node contains a data item, a pointer to the previous struct node, and a pointer to the next struct node.

The following example creates a basic Linked List with three items and displays its elements.

Example: Creating a Doubly Linked List

```
#include <stdio.h>

#include <stdlib.h>

struct node
```

```
{
int data;
struct node *next;
struct node *prev;
};
int main()
{
// Create and initialize nodes
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;
struct node *current = NULL;
// Allocate memory for the nodes
one = (struct node*) malloc(sizeof(struct node));
two = (struct node*) malloc(sizeof(struct node));
three = (struct node*) malloc(sizeof(struct node));
// Assign data to nodes
one->data = 1;
two->data = 2;
three->data = 3;
// Connect first node
// with the second node
one->next = two;
// Connect second node
// with the third node
two->next = three;
```

```
// make next pointer of

//third node to NULL

//indicates last node

three->next = NULL;

//connect previous nodes

one->prev = NULL;

two->prev = one;

three->prev = two;

// Save address of first node in head

head = one;

current = head;

// print the linked list values forward

while(current != NULL)

{

printf("%d ", current->data);

current = current->next;

}

return 0;

}
```

Now we've made a doubly-linked list with three nodes.

Output

```
1 2 3
```

More

Doubly Linked List

A doubly linked list is a type of linked list in which each node consists of 3 components:

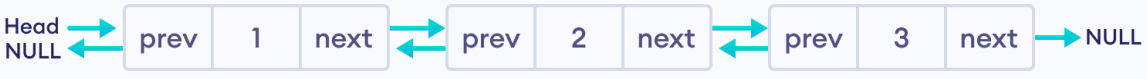
- `*prev` - address of the previous node
- `data` - data item
- `*next` - address of next node

A doubly linked list node

> Note: Before you proceed further, make sure to learn about pointers and structs.

Representation of Doubly Linked List

Let's see how we can represent a doubly linked list on an algorithm/code. Suppose we have a doubly linked list:



Newly created doubly linked list

Here, the single node is represented as

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

```
    struct node *next;
    struct node *prev;
}
```

Each struct node has a data item, a pointer to the previous struct node, and a pointer to the next struct node.

Now we will create a simple doubly linked list with three items to understand how this works.

```
/* Initialize nodes */
struct node *head;
struct node *one = NULL;
struct node *two = NULL;
struct node *three = NULL;

/* Allocate memory */
one = malloc(sizeof(struct node));
two = malloc(sizeof(struct node));
three = malloc(sizeof(struct node));

/* Assign data values */
one->data = 1;
two->data = 2;
three->data = 3;

/* Connect nodes */
one->next = two;
one->prev = NULL;

two->next = three;
two->prev = one;

three->next = NULL;
three->prev = two;

/* Save address of first node in head */
head = one;
```

In the above code, `one`, `two`, and `three` are the nodes with data items **1**, **2**, and **3** respectively.

- **For node one**: `next` stores the address of `two` and `prev` stores `null` (there is no node before it)
- **For node two**: `next` stores the address of `three` and `prev` stores the address of `one`
- **For node three**: `next` stores `null` (there is no node after it) and `prev` stores the address of `two`.

> **Note**: In the case of the head node, `prev` points to `null`, and in the case of the tail pointer, `next` points to null. Here, `one` is a head node and `three` is a tail node.

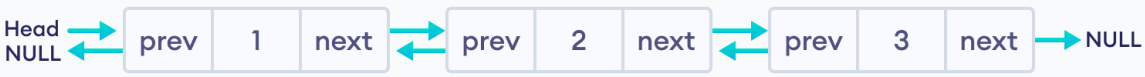
Insertion on a Doubly Linked List

Pushing a node to a doubly-linked list is similar to pushing a node to a linked list, but extra work is required to handle the pointer to the previous node.

We can insert elements at 3 different positions of a doubly-linked list:

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

Suppose we have a double-linked list with elements **1**, **2**, and **3**.



Original doubly linked list

---

1. Insertion at the Beginning

Let's add a node with value **6** at the beginning of the doubly linked list we made above.

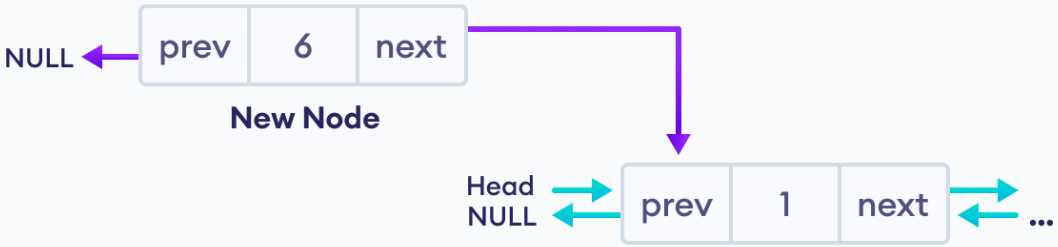
1. Create a new node

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

New node

2. Set prev and next pointers of new node

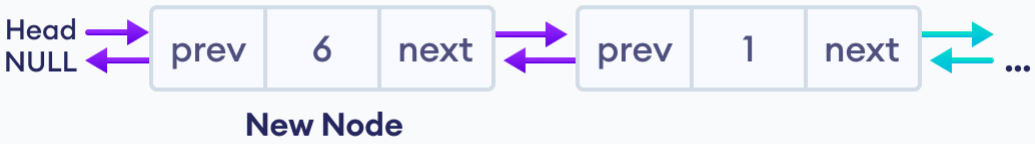
- point `next` of `newNode` to the first node of the doubly linked list
- point `prev` to `null`



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

Code for Insertion at the Beginning

```
// 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;
}
```

---

2. Insertion in between two nodes

Let's add a node with value 6 after node with value 1 in the doubly linked list.

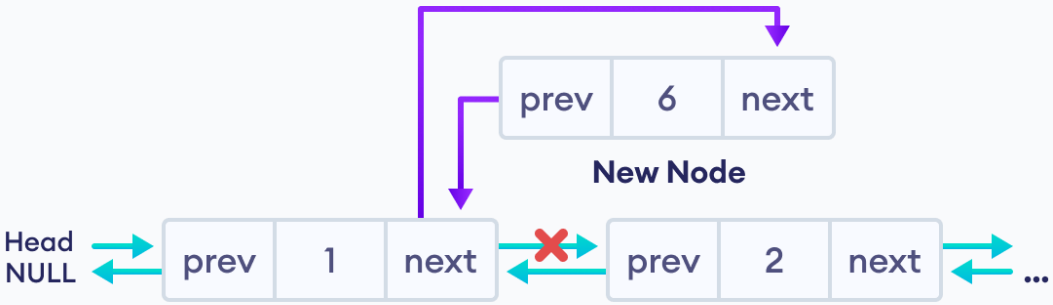
1. Create a new node

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

New node

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

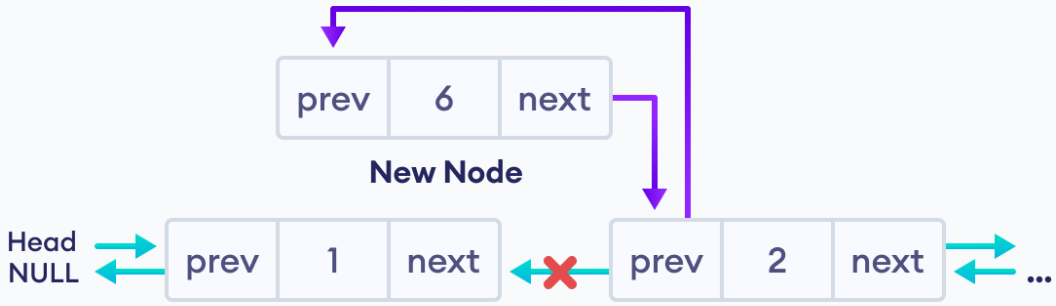
- assign the value of `next` from previous node to the `next` of `newNode`
- assign the address of `newNode` to the `next` of previous node



Reorganize the pointers

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



Reorganize the pointers

The final doubly linked list is after this insertion is:

Final list

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

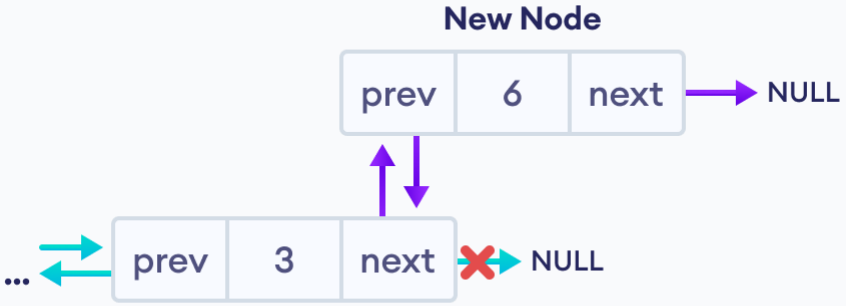
Let's add a node with value 6 at the end of the doubly linked list.

1. Create a new node

New node

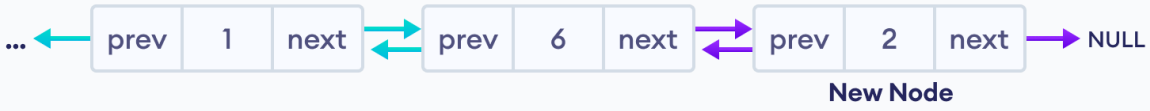
2. Set prev and next pointers of new node and the previous node

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



Reorganize the pointers

The final doubly linked list looks like this.



The final list

Code for Insertion at the End

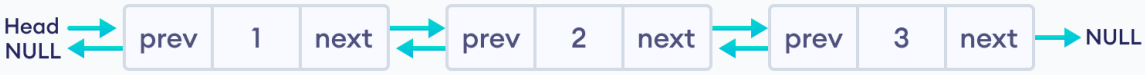
```
// insert a newNode at the end of the list
void insertEnd(struct Node** head, int data) {
    // allocate memory for node
    struct Node* newNode = new Node;

    // 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**.



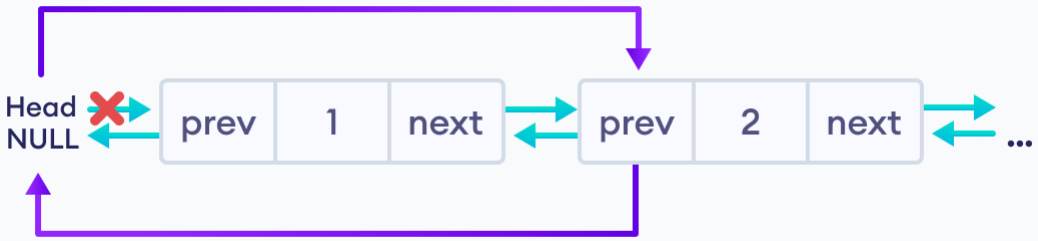
Original doubly linked list

---

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)



Reorganize the pointers

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

Free the space of the first node

Final list

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.

Reorganize the pointers

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

Final list

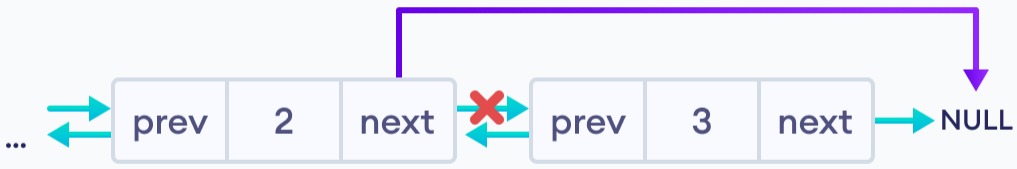
Code for Deletion of the Inner Node

```
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

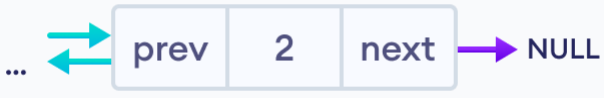
In this case, we are deleting the last node with value **3** of the doubly linked list.

Here, we can simply delete the `del_node` and make the `next` of node before `del_node` point to `NULL`.



Reorganize the pointers

The final doubly linked list looks like this.



Final list

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`.

> **Note**: We can also solve this using the first condition (for the node before `del_node`) of the second case (Delete the inner node).

Doubly Linked List Code in Python, Java, C, and C++

Python

Java

C

C++

```python
import gc

# node creation

class Node:

    def __init__(self, data):
        self.data = data
        self.next = None
        self.prev = None


class DoublyLinkedList:

    def __init__(self):
        self.head = None

    # insert node at the front
    def insert_front(self, data):

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)
```

```
        # make newNode as a head
        new_node.next = self.head

        # assign null to prev (prev is already none in the constructore)

        # previous of head (now head is the second node) is newNode
        if self.head is not None:
            self.head.prev = new_node

        # head points to newNode
        self.head = new_node

    # insert a node after a specific node
    def insert_after(self, prev_node, data):

        # check if previous node is null
        if prev_node is None:
            print("previous node cannot be null")
            return

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)

        # set next of newNode to next of prev node
        new_node.next = prev_node.next

        # set next of prev node to newNode
        prev_node.next = new_node

        # set prev of newNode to the previous node
        new_node.prev = prev_node

        # set prev of newNode's next to newNode
        if new_node.next:
            new_node.next.prev = new_node

    # insert a newNode at the end of the list
    def insert_end(self, data):

        # allocate memory for newNode and assign data to newNode
        new_node = Node(data)

        # assign null to next of newNode (already done in constructor)

        # if the linked list is empty, make the newNode as head node
```

```
        if self.head is None:
            self.head = new_node
            return

        # store the head node temporarily (for later use)
        temp = self.head

        # if the linked list is not empty, traverse to the end of the linked list
        while temp.next:
            temp = temp.next

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

        # assign next of the last node (temp) to newNode
        temp.next = new_node

        # assign prev of newNode to temp
        new_node.prev = temp

        return

    # delete a node from the doubly linked list
    def deleteNode(self, dele):

        # if head or del is null, deletion is not possible
        if self.head is None or dele is None:
            return

        # if del_node is the head node, point the head pointer to the next of
del_node
        if self.head == dele:
            self.head = dele.next

        # if del_node is not at the last node, point the prev of node next to
del_node to the previous of del_node
        if dele.next is not None:
            dele.next.prev = dele.prev

        # if del_node is not the first node, point the next of the previous node to
the next node of del_node
        if dele.prev is not None:
            dele.prev.next = dele.next

        # free the memory of del_node
        gc.collect()
```

```
    # print the doubly linked list
    def display_list(self, node):

        while node:
            print(node.data, end="->")
            last = node
            node = node.next


# initialize an empty node
d_linked_list = DoublyLinkedList()

d_linked_list.insert_end(5)
d_linked_list.insert_front(1)
d_linked_list.insert_front(6)
d_linked_list.insert_end(9)

# insert 11 after head
d_linked_list.insert_after(d_linked_list.head, 11)

# insert 15 after the seond node
d_linked_list.insert_after(d_linked_list.head.next, 15)

d_linked_list.display_list(d_linked_list.head)

# delete the last node
d_linked_list.deleteNode(d_linked_list.head.next.next.next.next.next)

print()
d_linked_list.display_list(d_linked_list.head)
```

Doubly Linked List Complexity

Doubly Linked List Complexity	**Time Complexity**	**Space Complexity**
Insertion Operation	O(1) or O(n)	O(1)
Deletion Operation	O(1)	O(1)

1. Complexity of Insertion Operation

- The insertion operations that do not require traversal have the time complexity of `O(1)`.
- And, insertion that requires traversal has time complexity of `O(n)`.
- The space complexity is `O(1)`.

2. Complexity of Deletion Operation

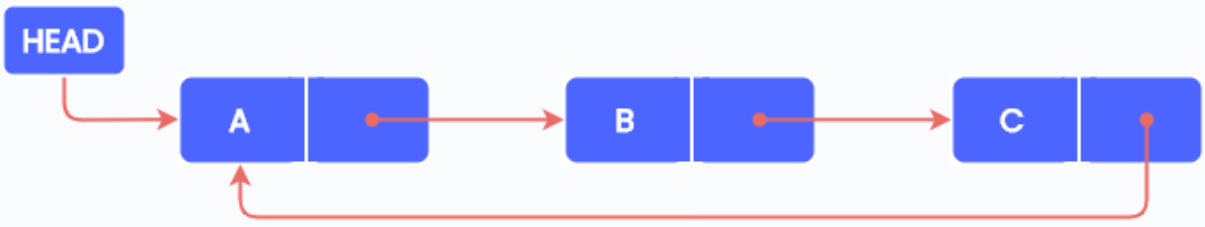
- All deletion operations run with time complexity of `O(1)`.
- And, the space complexity is `O(1)`.

Doubly Linked List Applications

1. Redo and undo functionality in software.
2. Forward and backward navigation in browsers.
3. For navigation systems where forward and backward navigation is required.

Circular Linked List

A circular linked list is a type of linked list in which the last node is also connected to the first node to form a circle. Thus a circular linked list has no end.



Circular Linked List Data Structure

There are two types of linked lists:

- Circular Singly Linked List.
- Circular Doubly Linked List.

Representation of Circular Linked List

Each of the nodes in the linked list consists of two parts:

- A data item.
- An address that points to another node.

A single node can be represented using structure as

```
struct node {

int data;

struct node *next;

};
```

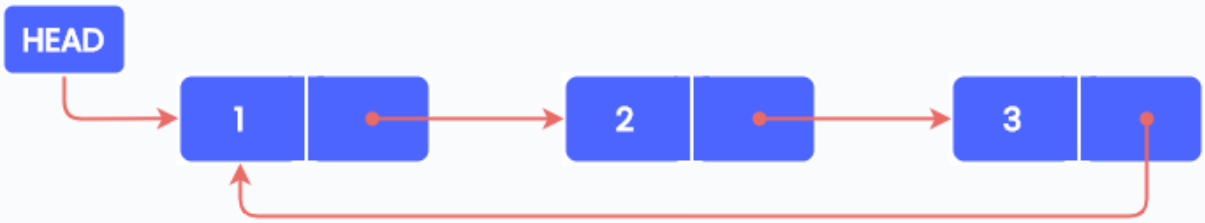
Each struct node contains a data item as well as a pointer to another struct node. The following example creates a Circular Linked List with three items and display its elements.

Example: Creating a Circular Linked List

```
#include <stdio.h>

#include <stdlib.h>

struct node

{

int data;

struct node *next;

};

int main()

{

// Create and initialize nodes

struct node *head;

struct node *one = NULL;

struct node *two = NULL;

struct node *three = NULL;

struct node *current = NULL;

// Allocate memory for the nodes
```

```
one = (struct node*) malloc(sizeof(struct node));

two = (struct node*) malloc(sizeof(struct node));

three = (struct node*) malloc(sizeof(struct node));

// Assign data to nodes

one->data = 1;

two->data = 2;

three->data = 3;

// Connect first node

// with the second node

one->next = two;

// Connect second node

// with the third node

two->next = three;

// Connect second node

// with the third node

three->next = one;

// Save address of first node in head

head = one;

current = head;

// print the linked list values

while(current != NULL)

{

printf("%d ", current->data);

current = current->next;

}

return 0;

}
```

Now we've made a circular linked list with three nodes.



Representation of Circular Linked List

Output

```
1  2  3
```

Inserting a New Node to a Circular Linked List

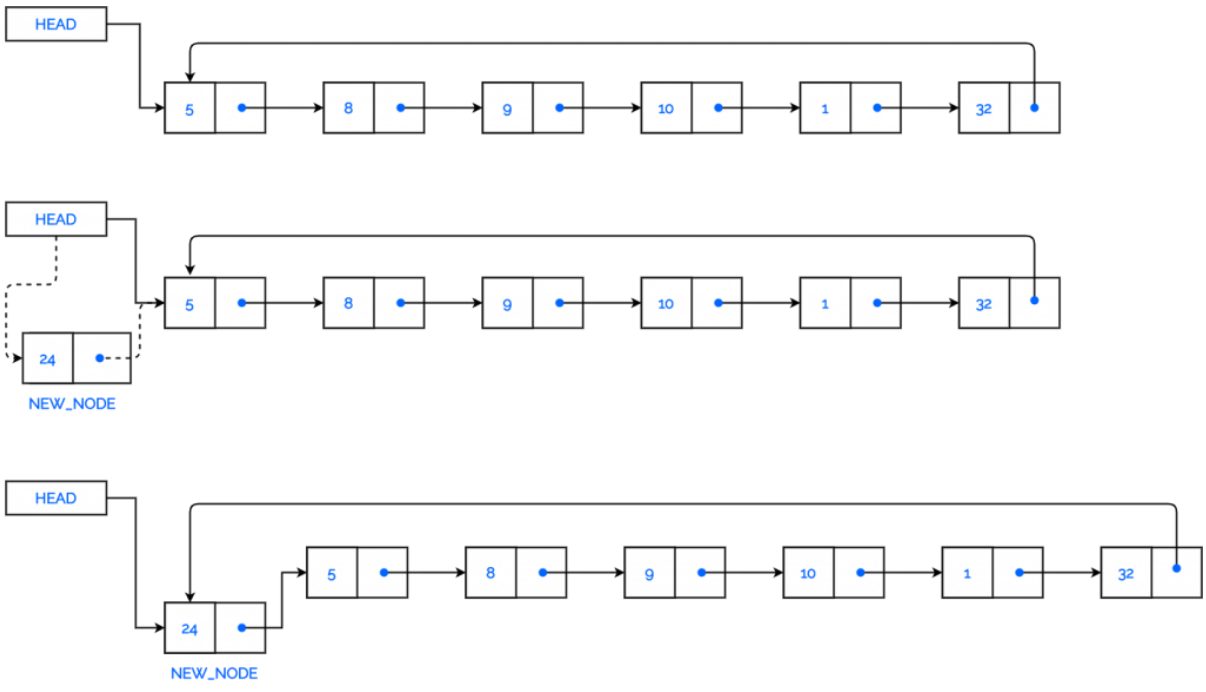
A new node can be inserted anywhere in a circular linked list. Here, we are discussing the following cases.

- Inserting a new Node at the beginning of a Circular Linked List.
- Inserting a new Node at the end of a Circular Linked List.

The remaining cases are the same as those described for a singly linked list.

Insert at the beginning of a list

Suppose we want to add a new node with data 24 as the first node in the following linked list. The following changes will be done in the linked list.



- Allocate memory for new node and initialize its DATA part to 24.

- Add the new node as the first node of the list by pointing the NEXT part of the new node to HEAD.
- Make HEAD to point to the first node of the list.
- Make HEAD point to the new node.

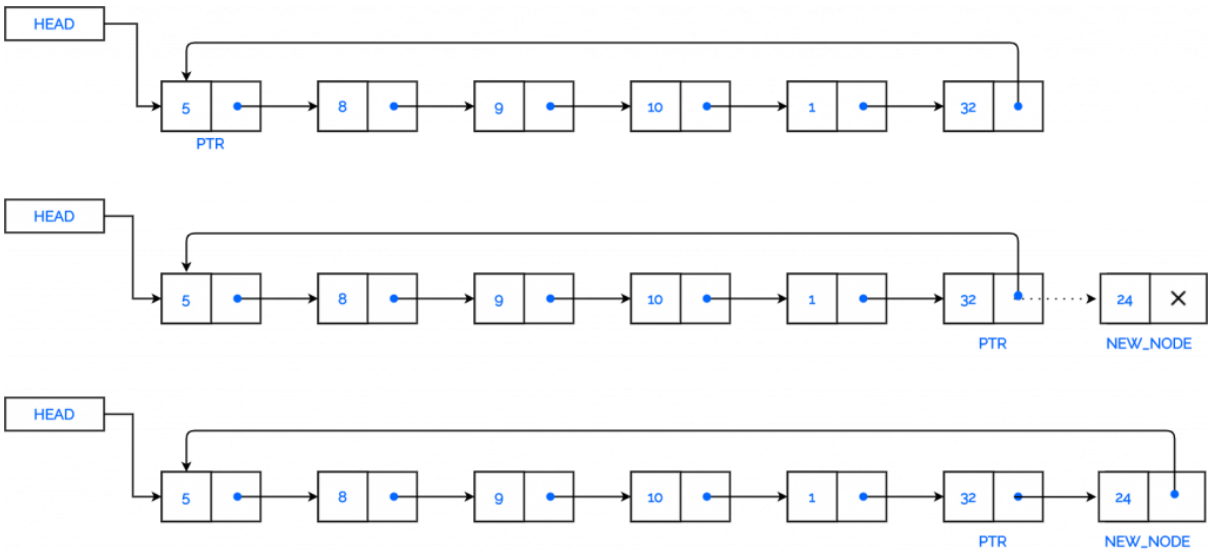
Algorithm: InsertAtBeginning

```
Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 11

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET PTR = HEAD

Step 6: Repeat Step 7 while PTR -> NEXT != HEAD

Step 7: PTR = PTR -> NEXT

[END OF LOOP]

Step 8: SET NEW_NODE -> NEXT = HEAD

Step 9: SET PTR -> NEXT = NEW_NODE

Step 10: SET HEAD = NEW_NODE

Step 11: EXIT
```

Note that the first step of the algorithm checks if there is enough memory available to create a new node. The second, and third steps allocate memory for the new node.

Insert at the end of the list

Suppose we want to add a new node with data 24 as the last node of the following circular linked list. The following changes have to be made in the linked list.



- Allocate memory for the new node and initialize data.
- Traverse to the end of the list.
- Point the NEXT part of the last node to the newly created node.
- Make the value of next part of last node to HEAD.

Algorithm: InsertAtLast

```
Step 1: IF AVAIL = NULL

Write OVERFLOW

Go to Step 1

[END OF IF]

Step 2: SET NEW_NODE = AVAIL

Step 3: SET AVAIL = AVAIL -> NEXT

Step 4: SET NEW_NODE -> DATA = VAL

Step 5: SET NEW_NODE -> NEXT = HEAD

Step 6: SET PTR = HEAD

Step 7: Repeat Step 8 while PTR -> NEXT != HEAD

Step 8: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 9: SET PTR -> NEXT = NEW_NODE

Step 1 : EXIT
```

Deleting a Node from a Circular Linked List

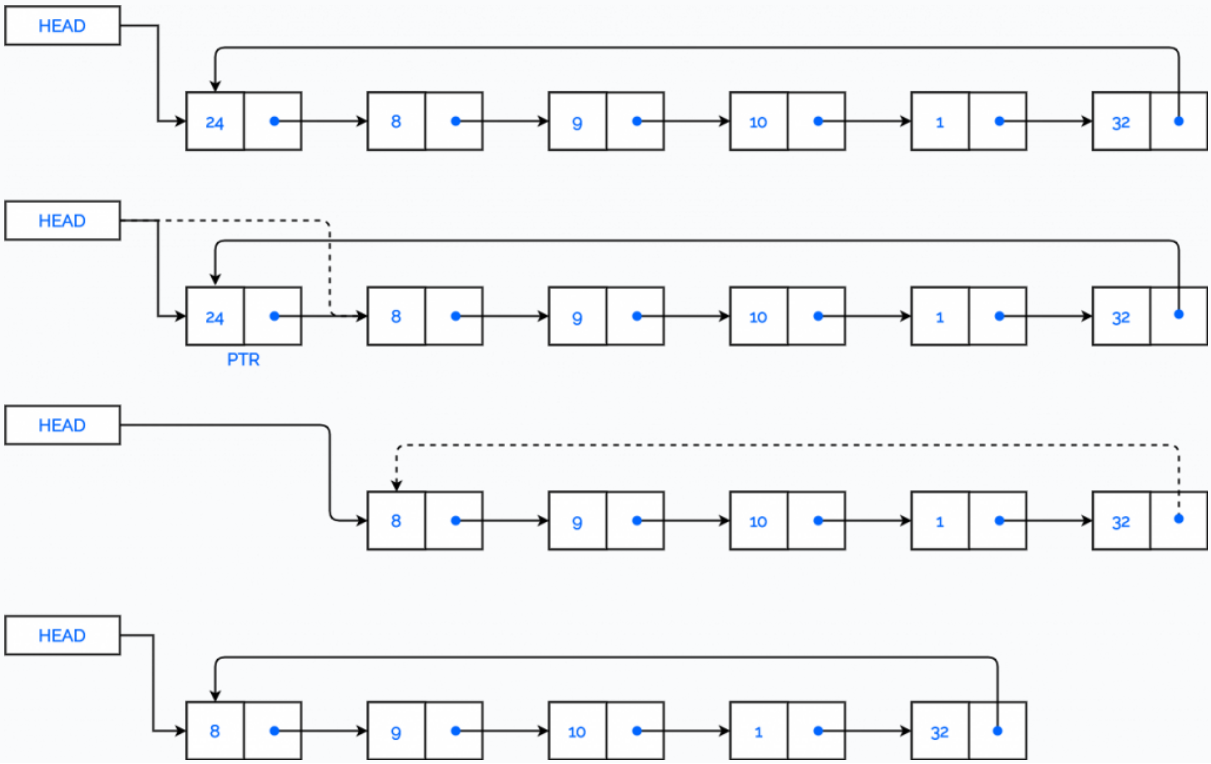
Let's look at how a node is removed from a circular linked list in the following scenarios.

- The first node is deleted.
- The last node is deleted.

The remaining cases are the same as those described for a singly linked list.

Deleting first node from a list

The following changes have to be made to the linked list if we want to remove the first node having data 24.



- Check if the linked list is empty or not. Exit if the list is empty.
- Make HEAD points to the second node.
- Traverse to the end of the list and point the next part of last node to second node.
- Free the first node from memory.

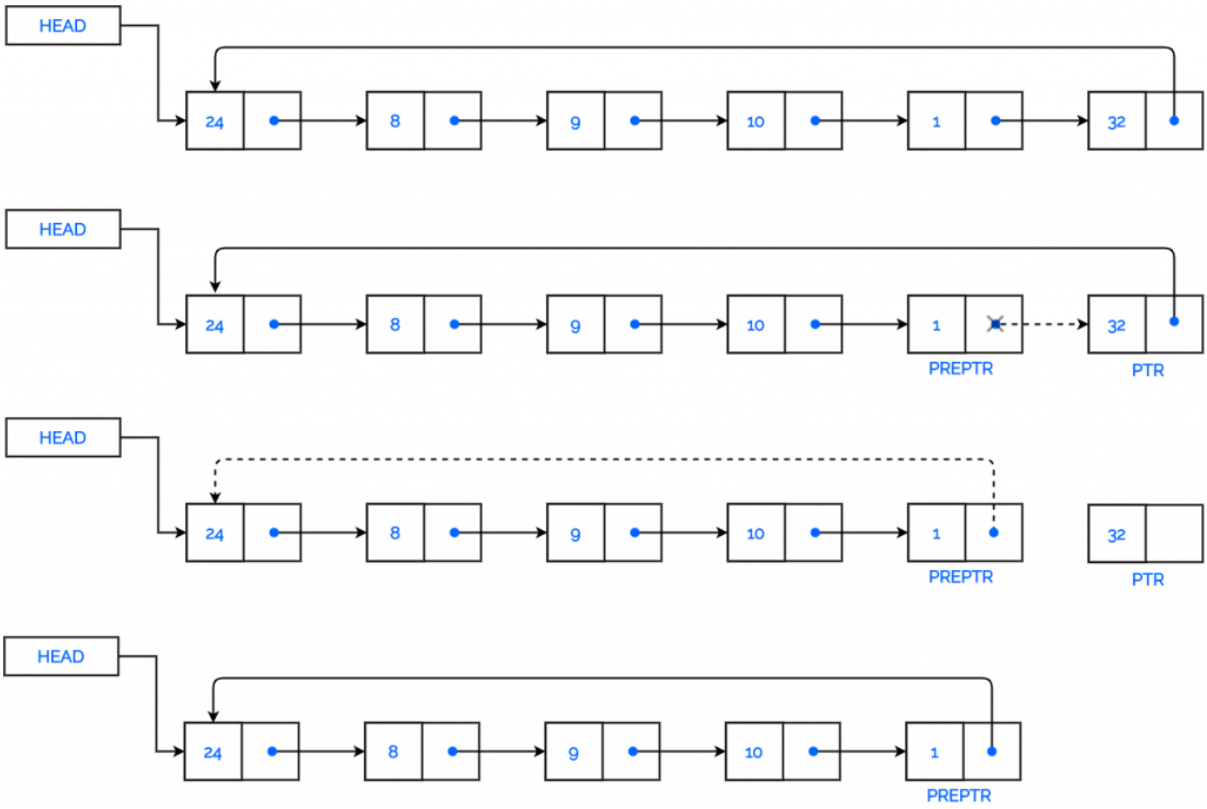
Algorithm: DeleteFirst

```
Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8

[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Step 4 while PTR -> NEXT != HEAD
```

```
Step 4: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 5: SET PTR -> NEXT = HEAD -> NEXT

Step 6: FREE HEAD

Step 7: SET HEAD = PTR -> NEXT

Step 8: EXIT
```

Deleting last node from the list

Suppose we want to delete the last node from the circular linked list. The following changes will be done in the list.



- Traverse to the end of the list.
- Change value of next pointer of second last node to HEAD.
- Free last node from memory.

Algorithm: DeleteFirst

```
Step 1: IF HEAD = NULL

Write UNDERFLOW

Go to Step 8
```

```
[END OF IF]

Step 2: SET PTR = HEAD

Step 3: Repeat Steps 4 and 5 while PTR NEXT != HEAD

Step 4: SET PREPTR = PTR

Step 5: SET PTR = PTR -> NEXT

[END OF LOOP]

Step 6: SET PREPTR -> NEXT = HEAD

Step 7: FREE PTR

Step 8: EXIT
```