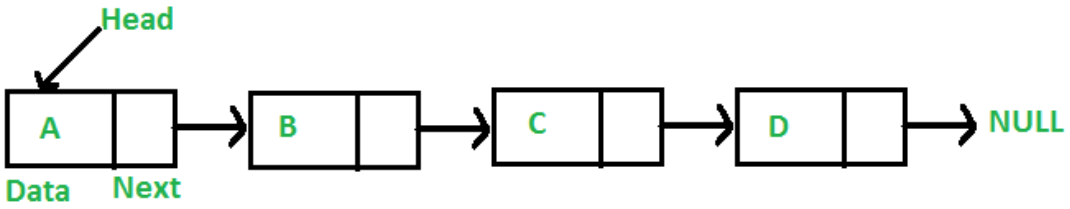
TOPIC - LINKED LISTS

Introduction:

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations. The elements in a linked list are linked using pointers as shown in the below image:



In simple words, a linked list consists of nodes where each node contains a data field and a reference(link) to the next node in the list.

Linked list Vs Array:

Arrays store elements in contiguous memory locations, resulting in easily calculable addresses for the elements stored and this allows a faster access to an element at a specific index. Linked lists are less rigid in their storage structure and elements are usually not stored in contiguous locations, hence they need to be stored with additional tags giving a reference to the next element. This difference in the data storage scheme decides which data structure would be more suitable for a given situation.

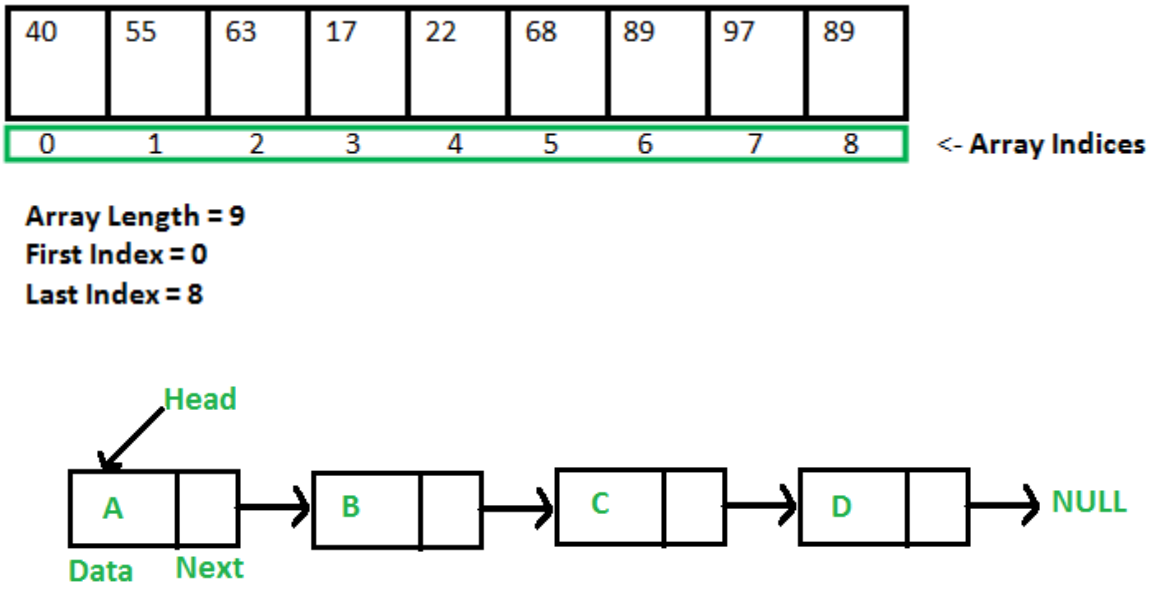


Fig: Data storage scheme of an array and a linked list

Advantages of Linked Lists:

1. The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, the upper limit is rarely reached.
2. Inserting a new element in an array of elements is expensive because a room has to be created for the new elements and to create room existing elements have to be shifted.

   For example, suppose we maintain a sorted list of IDs in an array id[ ].

   id[ ] = [1000, 1010, 1050, 2000, 2040, ...].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides the following two advantages over arrays

1) Dynamic size
2) Ease of insertion/deletion

Disadvantages of Linked Lists:

1. Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do a binary search with linked lists.
2. Extra memory space for a pointer is required with each element of the list.
3. Arrays have better cache locality that can make a pretty big difference in performance.

Classification:

Linked lists can be broadly classified into 3 categories:

1. Singly linked lists
2. Doubly linked lists
3. Circular linked lists

SINGLY LINKED LIST

A singly linked list is a type of linked list that is unidirectional, that is, it can be traversed in only one direction from head to the last node (tail).

Each element in a linked list is called a node. A single node contains data and a pointer to the next node which helps in maintaining the structure of the list.

```
// A linked list node
class Node
{
    public:
    int data;
    Node *next;
};
```

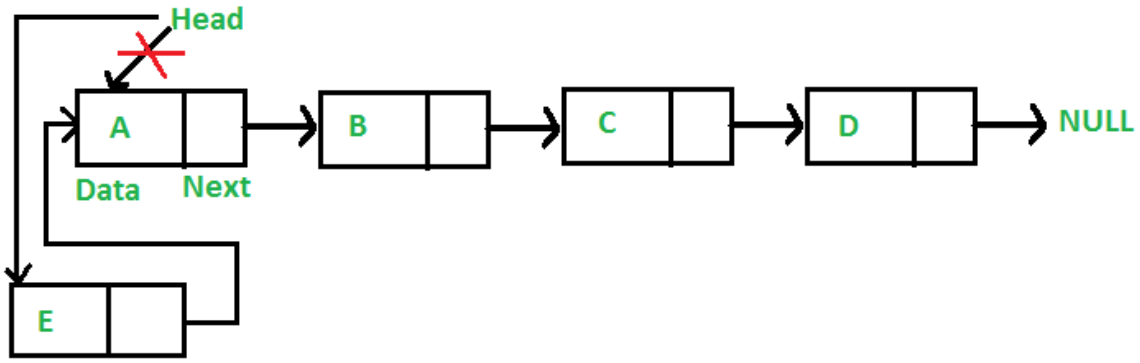
Insertion

A node can be added in three ways:

1) At the front of the linked list
2) After a given node.
3) At the end of the linked list.

1. Add a node at the front: (4 steps process)

The new node is always added before the head of the given Linked List. And the newly added node becomes the new head of the Linked List. For example, if the given Linked List is 10->15->20->25 and we add an item 5 at the front, then the Linked List becomes 5->10->15->20->25. Let us call the function that adds at the front of the list is push(). The push() must receive a pointer to the head pointer, because push must change the head pointer to point to the new node



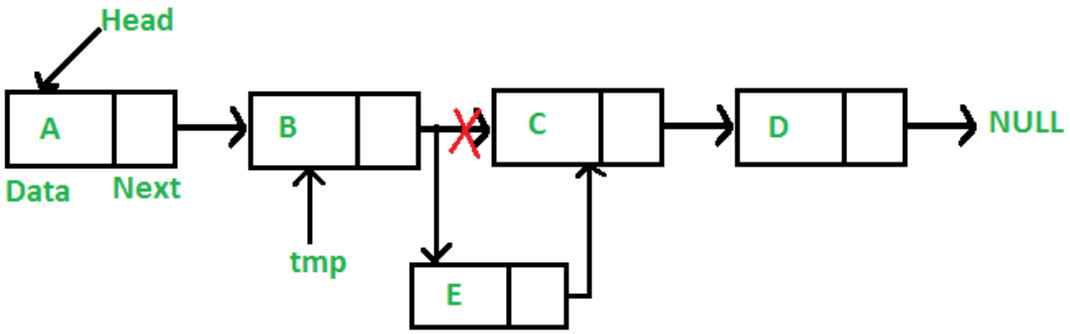
Code:

```
/* Given a reference (pointer to pointer)
to the head of a list and an int,
inserts a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
    /* 1. allocate node */
    Node* new_node = new Node();

    /* 2. put in the data */
    new_node->data = new_data;

    /* 3. Make next of new node as head */
    new_node->next = (*head_ref);

    /* 4. move the head to point to the new node */
    (*head_ref) = new_node;
}
```

Time complexity of push() is O(1) as it does a constant amount of work.

2. Add a node after a given node: (5 steps process)

We are given a pointer to a node, and the new node is inserted after the given node.



Code:

```
// Given a node prev_node, insert a
// new node after the given
// prev_node
void insertAfter(Node* prev_node, int new_data)
{

    // 1. Check if the given prev_node is NULL
    if (prev_node == NULL)
    {
        cout << "the given previous node cannot be NULL";
        return;
    }

    // 2. Allocate new node
    Node* new_node = new Node();
```

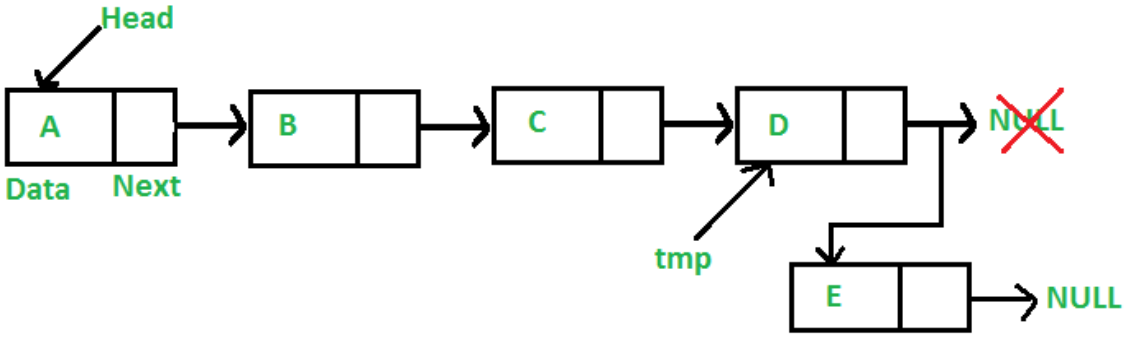
```
    // 3. Put in the data
    new_node->data = new_data;

    // 4. Make next of new node as next of prev_node
    new_node->next = prev_node->next;

    // 5. move the next of prev_node as new_node
    prev_node->next = new_node;
}
```

Time complexity of insertAfter() is O(1) as it does a constant amount of work.

3. Add a node at the end: (6 steps process)

The new node is always added after the last node of the given Linked List. For example if the given Linked List is 5->10->15->20->25 and we add an item 30 at the end, then the Linked List becomes 5->10->15->20->25->30.

Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then change the next to last node to a new node.



Code:

```
// Given a reference (pointer to pointer) to the head
// of a list and an int, appends a new node at the end
void append(Node** head_ref, int new_data)
{
    // 1. allocate node
    Node* new_node = new Node();

    // Used in step 5
    Node *last = *head_ref;

    // 2. Put in the data
    new_node->data = new_data;

    // 3. This new node is going to be
    // the last node, so make next of
    // it as NULL
    new_node->next = NULL;

    // 4. If the Linked List is empty,
    // then make the new node as head
    if (*head_ref == NULL)
    {
        *head_ref = new_node;
        return;
    }

    // 5. Else traverse till the last node
    while (last->next != NULL)
        last = last->next;

    // 6. Change the next of last node
```

```
    last->next = new_node;
    return;
}
```

Time complexity of append is O(n) where n is the number of nodes in the linked list. Since there is a loop from head to end, the function does O(n) work.

Note: This method can also be optimized to work in O(1) by keeping an extra pointer to the tail of the linked list.

Deletion

1. Deleting a given node

Given a 'key', delete the first occurrence of this key in the linked list.

Algorithm:

To delete a node from the linked list, we need to do the following steps.

1. Find the previous node of the node to be deleted.
2. Change the next of the previous nodes.
3. Free memory for the node to be deleted.

Code:

Refer to the link:

https://www.geeksforgeeks.org/linked-list-set-3-deleting-node/

Illustration:

```
Created Linked List:
 2  3  1  7
Linked List after Deletion of 1:
 2  3  7
```

2. Deleting a node at a given position

Algorithm:

1. If the node to be deleted is the root, simply delete it.
2. To delete a middle node, we must have a pointer to the node previous to the node to be deleted. So if 'position' is not zero, we run a loop position-1 times and get a pointer to the previous node.
3. Change the next of this node and free memory for the node to be deleted.

Code:

Refer to the link:

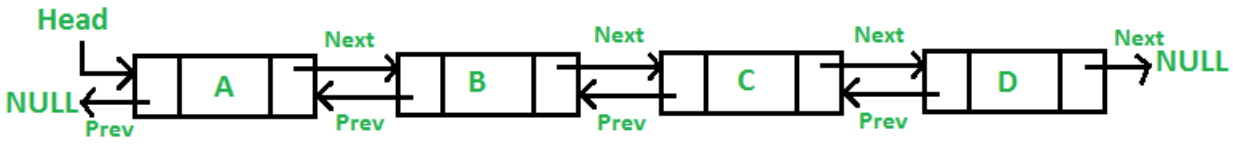
https://www.geeksforgeeks.org/delete-a-linked-list-node-at-a-given-position/

Illustration:

```
Input: position = 1, Linked List = 8->2->3->1->7
Output: Linked List =  8->3->1->7

Input: position = 0, Linked List = 8->2->3->1->7
Output: Linked List = 2->3->1->7
```

DOUBLY LINKED LIST

A Doubly Linked List (DLL) contains an extra pointer, typically called previous pointer, together with next pointer and data which are there in singly linked list.



Following is representation of a DLL node:

```
/* Node of a doubly linked list */
struct Node {
     int data;
     struct Node* next; // Pointer to next node in DLL
     struct Node* prev; // Pointer to previous node in DLL
};
```

Advantages over singly linked list:

1. A DLL can be traversed in both forward and backward direction.
2. The delete operation in DLL is more efficient if a pointer to the node to be deleted is given.
3. We can quickly insert a new node before a given node.

In a singly linked list, to delete a node, a pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In a DLL, we can get the previous node using previous pointer.

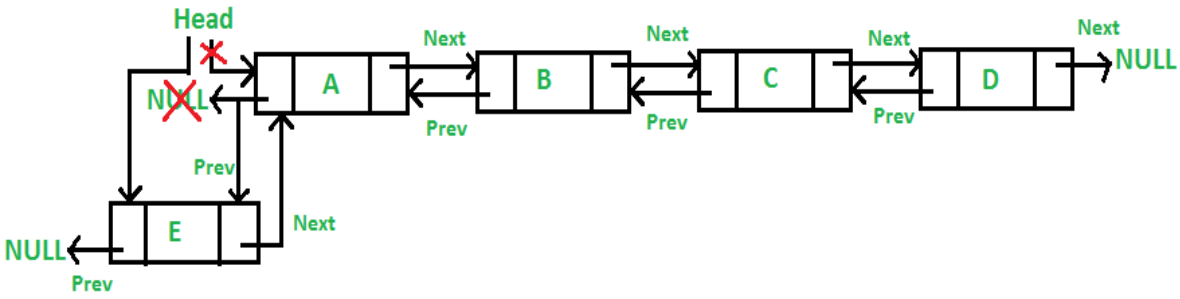
Disadvantages over singly linked list:

1. Every node of the DLL requires extra space for a previous pointer.
2. All operations require an extra pointer 'previous' to be maintained.

   For example, in insertion, we need to modify previous pointers together with next pointers.

Insertion in DLL

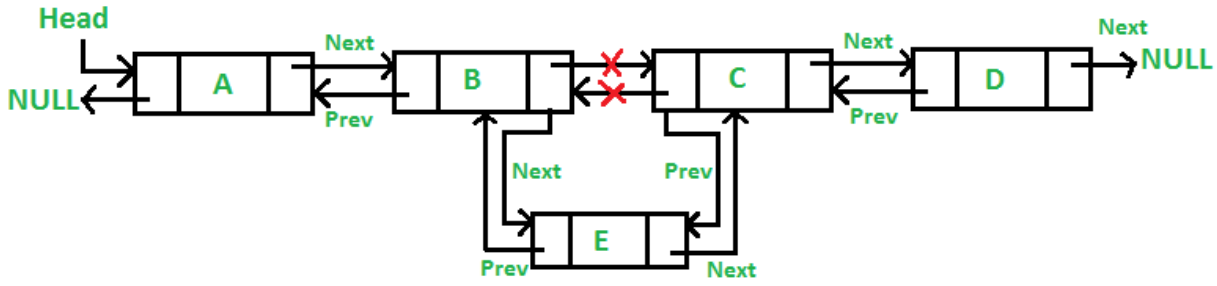
1. Add a node at the front (A 5 steps process):




```
/* Given a reference (pointer to pointer) to the head of a list
and an int, insert a new node on the front of the list. */
void push(struct Node** head_ref, int new_data)
{
     /* 1. allocate node */
     struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

     /* 2. put in the data */
     new_node->data = new_data;
```

```
     /* 3. Make next of new node as head and previous as NULL
*/
     new_node->next = (*head_ref);
     new_node->prev = NULL;

     /* 4. change prev of head node to new node */
     if ((*head_ref) != NULL)
          (*head_ref)->prev = new_node;

     /* 5. move the head to point to the new node */
     (*head_ref) = new_node;
}
```

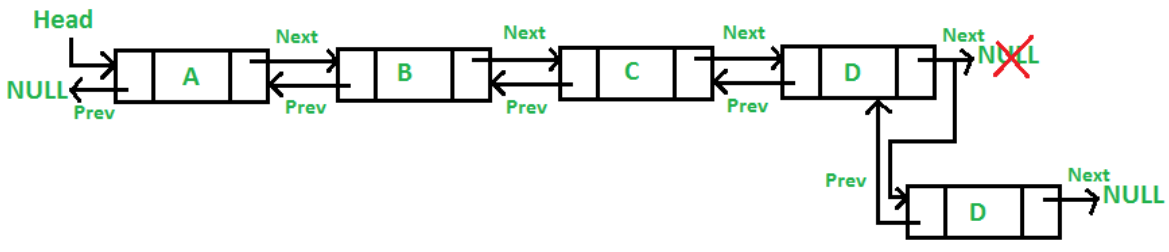
2. Add a node after a given node (A 7 steps process):




```
/* Given a node as prev_node, insert a new node after the given
node */
void insertAfter(struct Node* prev_node, int new_data)
{
     /*1. check if the given prev_node is NULL */
     if (prev_node == NULL) {
          printf("the given previous node cannot be NULL");
          return;
```

```
    }

    /* 2. allocate new node */
    struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

    /* 3. put in the data */
    new_node->data = new_data;

    /* 4. Make next of new node as next of prev_node */
    new_node->next = prev_node->next;

    /* 5. Make the next of prev_node as new_node */
    prev_node->next = new_node;

    /* 6. Make prev_node as previous of new_node */
    new_node->prev = prev_node;

    /* 7. Change previous of new_node's next node */
    if (new_node->next != NULL)
        new_node->next->prev = new_node;
}
```

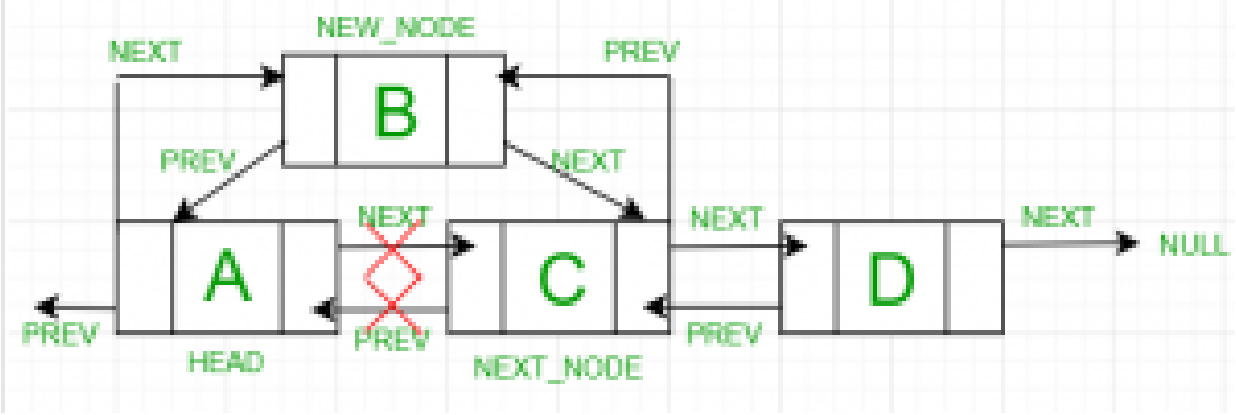
3. Add a node at the end (7 steps process):



```
/* Given a reference (pointer to pointer) to the head
of a DLL and an int, appends a new node at the end */
void append(struct Node** head_ref, int new_data)
{
     /* 1. allocate node */
     struct Node* new_node = (struct Node*)malloc(sizeof(struct
Node));

     struct Node* last = *head_ref; /* used in step 5*/

     /* 2. put in the data */
     new_node->data = new_data;

     /* 3. This new node is going to be the last node, so
          make next of it as NULL*/
     new_node->next = NULL;

     /* 4. If the Linked List is empty, then make the new
          node as head */
     if (*head_ref == NULL) {
          new_node->prev = NULL;
          *head_ref = new_node;
          return;
     }

     /* 5. Else traverse till the last node */
     while (last->next != NULL)
          last = last->next;

     /* 6. Change the next of last node */
     last->next = new_node;
```

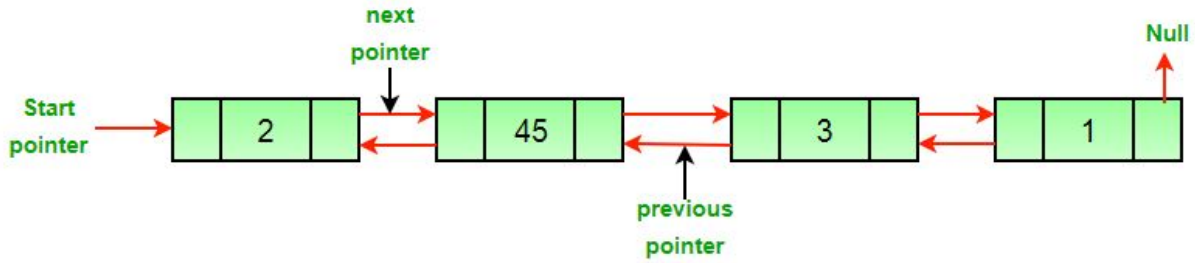
```
    /* 7. Make last node as previous of new node */
    new_node->prev = last;

    return;
}
```

4. Add a node before a given node:



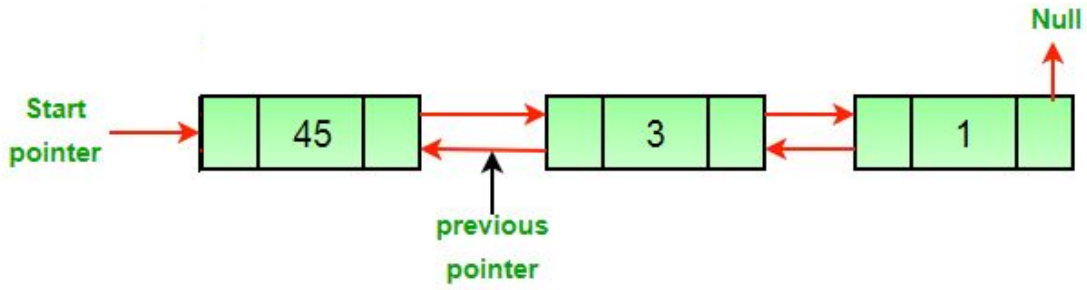
Deletion in DLL

Original DLL:

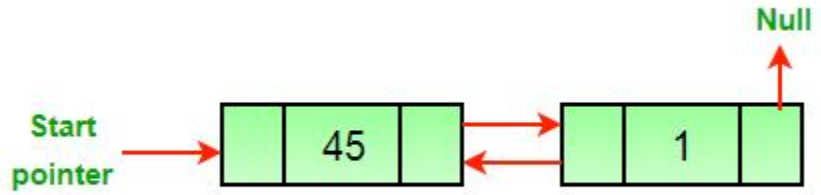


The deletion of a node in a doubly-linked list can be divided into three main categories:

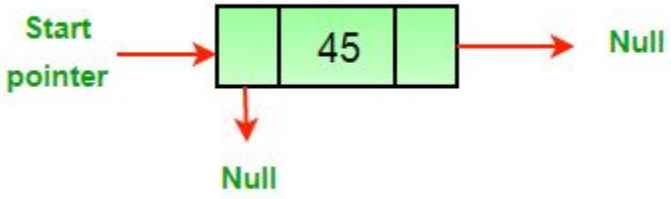
a. After the deletion of the head node.



b. After the deletion of the middle node.

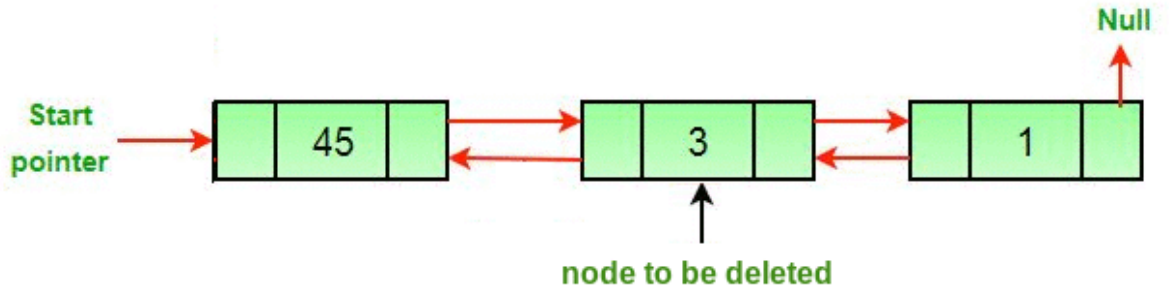


c. After the deletion of the last node.



All three mentioned cases can be handled in two steps if the pointer of the node to be deleted and the head pointer is known:

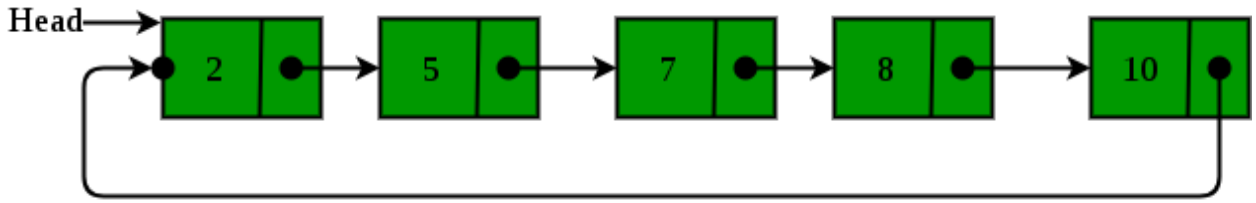
1. If the node to be deleted is the head node then make the next node as head.
2. If a node is deleted, connect the next and previous node of the deleted node.



Check the implementation from here.

CIRCULAR LINKED LIST

Circular linked list is a linked list where all nodes are connected to form a circle. There is no NULL at the end. A circular linked list can be a singly circular linked list or doubly circular linked list.



Advantages of Circular Linked Lists:

1. Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
2. Useful for implementation of the queue. Unlike this implementation, we don't need to maintain two pointers for front and rear if we use a circular linked list. We can maintain a pointer to the last inserted node and front can always be obtained as next to 'last'.
3. Circular lists are useful in applications to repeatedly go around the list. For example, when multiple applications are running on a PC, it is common for the operating system to put the running applications on a list and then to cycle through them, giving each of them a slice of time to execute, and then making them wait while the CPU is given to another application. It is convenient for the operating system to use a circular list

so that when it reaches the end of the list it can cycle around to the front of the list.

4. Circular Doubly Linked Lists are used for implementation of advanced data structures like Fibonacci Heap.

Traversal

In a conventional linked list, we traverse the list from the head node and stop the traversal when we reach NULL. In a circular linked list, we stop traversal when we reach the first node again. Following is the program for the linked list traversal.

(Class Node is the same class as the one used in Singly linked list)

```
/* Function to print nodes in
a given Circular linked list */
void printList(Node* head)
{
     Node* temp = head;

     // If linked list is not empty
     if (head != NULL) {

          // Print nodes till we reach first node again
          do {
               cout << temp->data << " ";
               temp = temp->next;
          } while (temp != head);
     }
}
```

Additional Links:

1. https://www.geeksforgeeks.org/check-if-a-linked-list-is-circular-linked-list
2. https://www.geeksforgeeks.org/write-a-c-function-to-detect-loop-in-a-linked-list/
3. https://www.geeksforgeeks.org/write-a-function-to-reverse-the-nodes-of-a-linked-list/
4. https://www.geeksforgeeks.org/merge-sort-for-doubly-linked-list/
5. https://www.geeksforgeeks.org/reverse-a-doubly-linked-list/