Data Structure & Algorithms

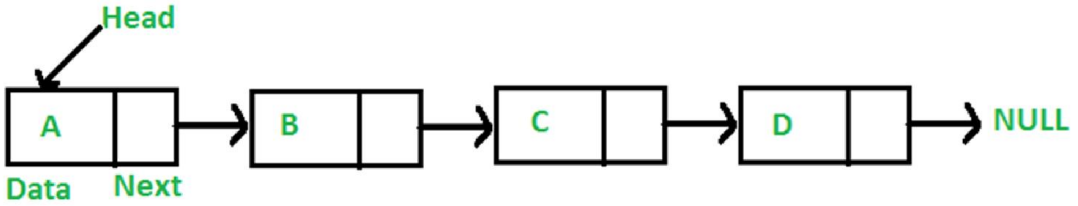
Linked List

What is Linked List?

- A linked list is a data structure in which the objects are arranged in a linear order.
- Unlike an array, the order in a linked list is determined by a pointer in each object, which means the elements **are not stored** at contiguous memory locations.
- In simple words, a linked list consists of nodes where each node contains a **data** field and a **reference** (pointer) to the next node in the list.

Representation

- A linked list is represented by a pointer to the first node of the linked list. The first node is called the **head**.
- Each node in a list consists of at least two parts:
    - Data
    - Pointer (or Reference) to the next node
- Last node carries a reference as NULL to mark **the end** of the list.
- **Note**: If the linked list is empty, then the value of the head is **NULL**.



Implementation of Linked List

- Each node can be defined as a struct:

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

- For specifying the first node (head) in an empty list:

```
struct node *head = NULL;
```

Implementation of Linked List (cont.)

- Memory allocation:

```
struct node* getNode(){
   struct node *p = (struct node*) malloc (sizeof(struct node));
   return p;
}

struct node *p = getNode();
p -> data = 5;          // or (*p).data = 5
p -> next = NULL;      // or (*p).next = NULL
```

Operations on Linked List

- Basically there are 3 main operations performed in a Linked List:

    1. Search
    2. Insert Node (at the front, at the end, at a specified location)
    3. Delete Node (from the front, from the end, from a specified location)

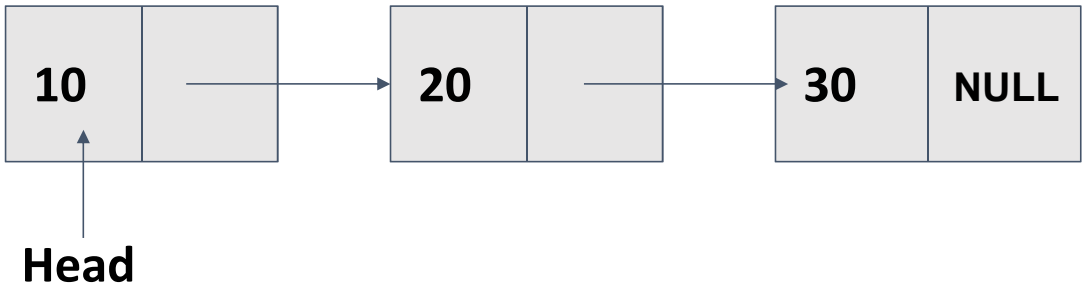
Search Operation

This procedure finds the first element with key k in a linked list by a simple linear search, returning a pointer to this element. If no object with key k appears in the list or if the list is empty, the procedure returns NULL. See the algorithm below:

```
struct node* search (struct node* head, int key)
{
   struct node *p = head;
   while (p != NULL && p -> data != key){
       p = p -> next;
   }
   return p;
}
```

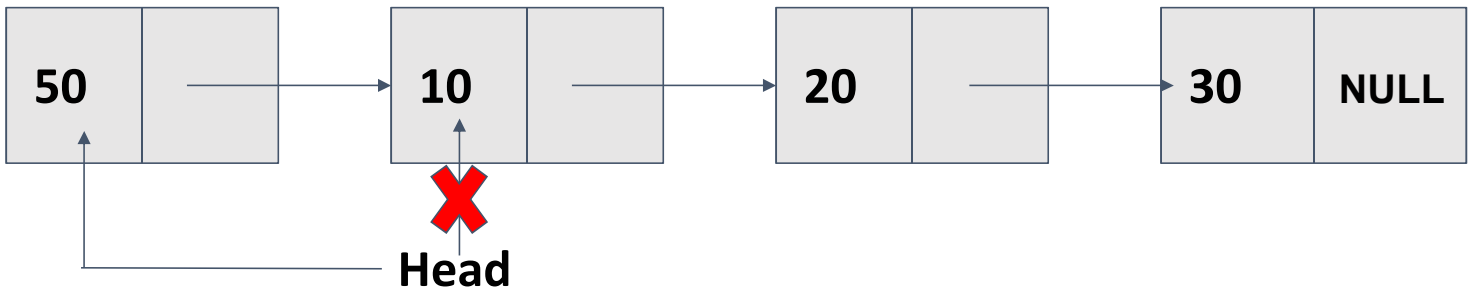
Insertion Operation

- Insertion to linked list can be done in three ways:
  - Inserting a node at the **front** of linked list.
  - Inserting a node at the **end** of linked list.
  - Inserting a node at a **specified location** of linked list.
- Say we have a linked list containing the elements 10, 20 and 30. Need to insert an element **50:**



1.Inserting a node at the front

The new node is added **before the head** of the given Linked List and the newly added node becomes the new head of the Linked List. In the previous example, a new node will be created containing element 50 pointing to the first node above list (containing 10). Head will now point to this new node:

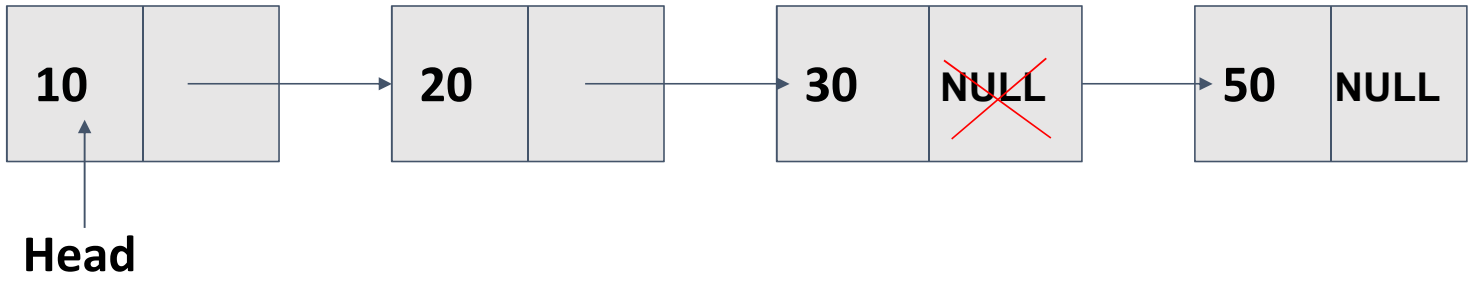


Inserting a node at the front - Algorithm

```
void insert_front (struct node** head, int new_data) {

   /* 1. allocate node */
   struct node* new_node = getNode();

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

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

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

2.Inserting a node at the End

The new node is added **after the last node** of the given Linked List. Since a Linked List is typically represented by the head of it, we have to traverse the list till the end and then add the new node to then end of the list. In the previous example, a new node will be created containing element 50 which points to NULL and the previous last node of the list (containing 30) points to this new node:

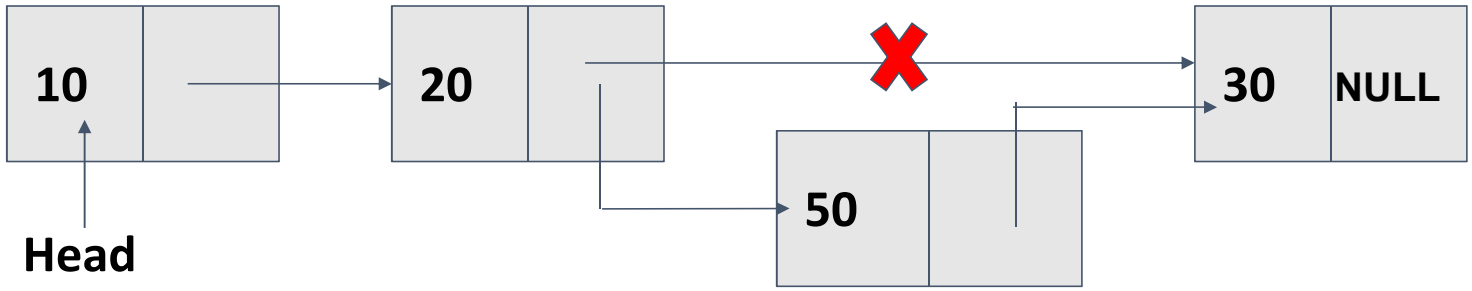


Inserting a node at the End - Algorithm

```
void insert_end (struct node** head, int new_data){
   /* 1. allocate node */
   struct node* new_node = getNode();
   struct node* last = (*head);  /* 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 == NULL){
       (*head) = 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;
}
```

3.Inserting a node at the Specified Location

In order to insert an element after the specified number of nodes into the linked list, we need to skip the desired number of elements in the list to move the pointer at the position after which the node will be inserted. In the previous example, say want to insert **element 50** after element 20. A new node containing 50 will be created and node containing 20 will point to the new node and that new node will point to the node containing 30:



Insert at a Specified location - Algorithm

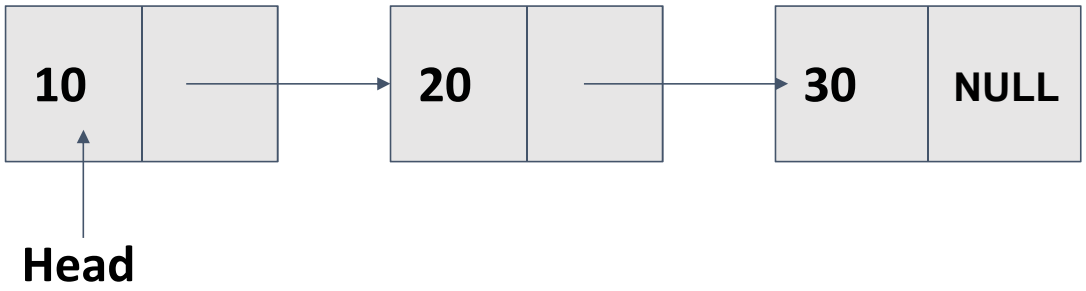
```
void insert_at_loc (struct node** head, int item, int loc){
   /* 1. allocate node */
   struct node *ptr = getNode();
   /* 2.Store the head to a temporary variable*/
   struct node *temp = *head;
   /* 3. put in the data */
   ptr->data = item;
   /* 4. traverse till the desired node */
   for (int i=0; i<loc; i++) {
       temp = temp->next;
       if (temp == NULL) {
           print("can't insert");
           return;
       }
   }
```

Insert at a Specified location (cont.)

```
   /* make the next part of the ptr, point to the next part of temp and ptr
will be in between temp and the      next of the temp. */
   ptr ->next = temp ->next;
   /* make the next part of the temp, point to the new node ptr.This will
insert the new node ptr, at the specified position. */
   temp ->next = ptr;
   return;
}
```

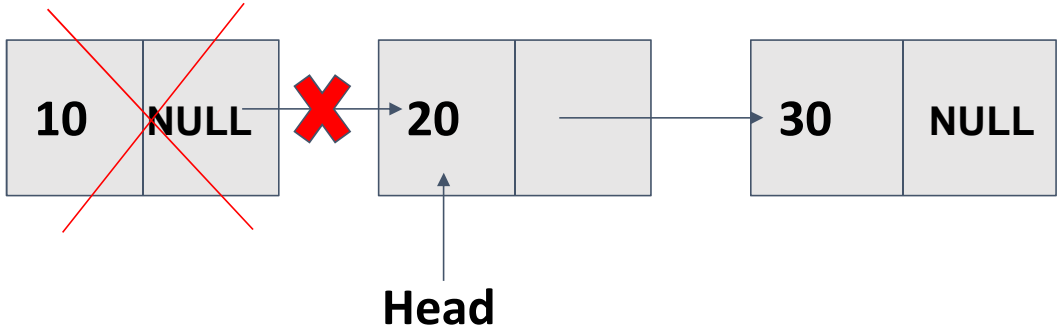
Deletion Operation

- Like insertion, deletion from linked list can also be done in three ways:
  - Deleting the **first** node of Linked List.
  - Deleting the **last** node of Linked List.
  - Deleting a node from a **specified location** of linked list.
- Again, consider the previous example**:**



1. Deleting the first node

Since the first node of the list is to be deleted, therefore, we just need to make the head point to the next of the head. After moving the head, we can free up the memory of the first node. In the previous example, after removing the first node, the node containing 20 will be the new head:



Deleting the first node - Algorithm

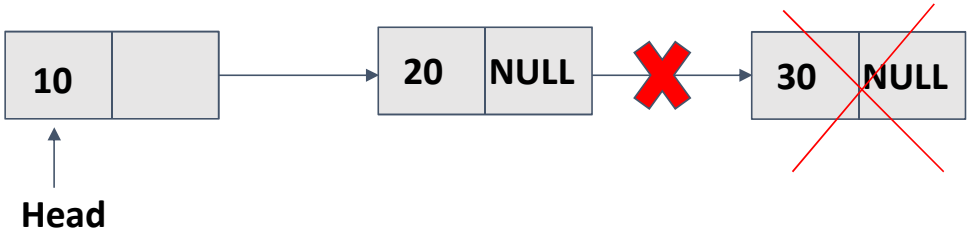
```
void delete_first_node (struct node** head){
   struct node* temp;

   /*Linked list is empty*/
   if (*head == NULL || head == NULL) return;

   /*Storing the head to a temporary variable*/
   temp = *head;

   /*Moving head to the next node*/
   (*head) = (*head)->next;

   /*Deleting the first node*/
   free(temp);
}
```

2. Deleting the last node

There are two scenarios in which a node is deleted from the end of the linked list:

1. There is only one node in the list that needs to be deleted. In this scenario, the only node of the list (head) will be assigned to null.
2. There are more than one node in the list and the last node of the list will be deleted. In this scenario, we have to traverse the nodes in order to reach the last node of the list.

In the previous example:



Deleting the last node - Algorithm

```
void delete_last_node (struct node** head) {
   struct node *prev = NULL, *cur = NULL;
   /*Linked list is empty*/
   if (*head == NULL || head == NULL) return;
   /*if there is only one node in the list, we just have to remove the head*/
   if ((*head) -> next == NULL) {
       free(*head);
       *head = NULL;
       return;
   }
   /*store the head for reaching the second last node. */
   prev = head;
   /*store the next of the head for reaching the last node..*/
   cur = prev -> next;
```

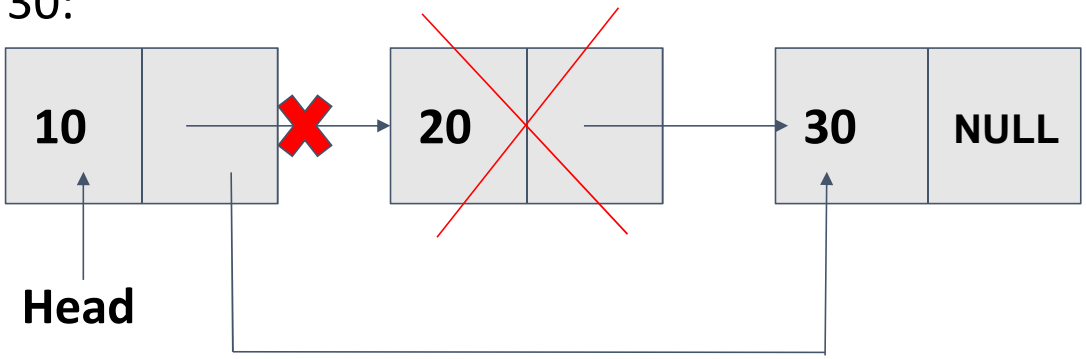
Deleting the last node (cont.)

```
    /*traverse the linked list till the second last node.*/
    while (cur -> next != NULL) {
        prev = prev -> next;
        cur = cur -> next;
    }
    /* link the reference of second last node point to NULL. */
    prev -> next = NULL;
    free(cur);
}
```

3. Deleting a node from a specified location

To do so, we need to traverse the linked list to the node just before the nth node, then change the pointer of that node to next of the next node and then delete the nth node.

For instance, if we want to delete the node containing 20 in the previous example, we have to move to node containing 10 and change the pointer of it to point to the node containing 30:



Deleting a node from a specified location - Algorithm

```
void delete_nth_node (struct node** head, int position) {
   // If linked list is empty
   if (*head == NULL || head == NULL)
       return;

   // Store head node
   struct node* temp = *head;

   // If head needs to be removed
   if (position == 0) {
       (*head) = temp->next;   // Change head
       free(temp);               // free old head
       return;
   }
```

Deleting a node from a specified location – Algorithm (cont.)

```
    // Find previous node of the node to be deleted
    for (int i=0; temp!=NULL && i<position-1; i++)
        temp = temp->next;

    // If position is more than number of nodes
    if (temp == NULL || temp->next == NULL)
        return;

    // Node temp->next is the node to be deleted
    // Store pointer to the next of node to be deleted
    struct node *next = temp->next->next;

    // Unlink the node from linked list
    free(temp->next);  // Free memory

    temp->next = next;  // Unlink the deleted node from list
}
```

Types of Linked List

Mainly there are 4 types of Linked List:

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

Until here, we have discussed about Singly Linked List which is the commonly used linked list in programs.

Doubly Linked List

A doubly linked list is a list that has **three parts** in a single node:

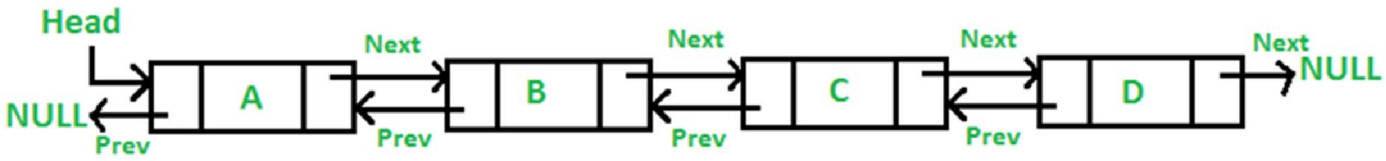
- a data part
- a pointer to **its previous node**
- a pointer to the **next node**

So a node is represented as:

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

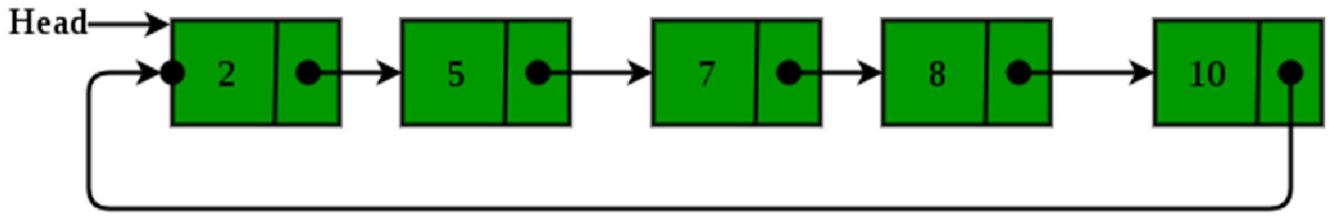
Advantages of Doubly Linked List

- A DLL can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.
- In singly linked list, to delete a node, pointer to the previous node is needed. To get this previous node, sometimes the list is traversed. In DLL, we can get the previous node using previous pointer.



Circular Linked List

- A circular linked list is a list in which **the last node connects to the first node**, so the link part of the last node holds the first node's address.
- The circular linked list has no starting and ending node. We can traverse in any direction, either backward or forward.
- The representation of the circular linked list will be similar to the singly linked list.



Doubly Circular Linked List

- The doubly circular linked list has the features of both the **circular linked list** and **doubly linked list.**
- It is circular because the last node is attached to the first node and thus creates a circle and it is doubly because each node holds the address of the previous node and the next node.
- As the doubly circular linked list contains three parts (two address parts and one data part), so its representation is similar to the doubly linked list.