Linked List

Linked List

- 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

Types of Linked List

Following are the types of linked list

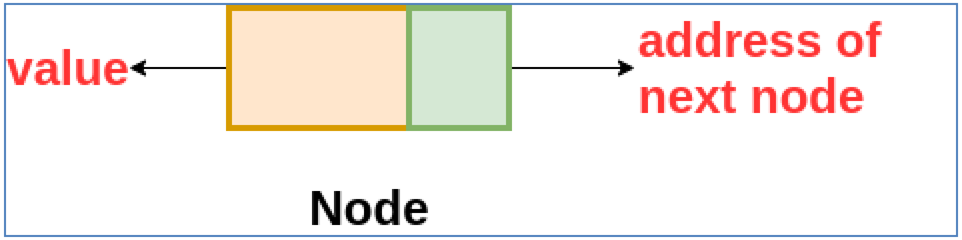
- Singly Linked List.
- Doubly Linked List.
- Circular Linked List.

Singly Linked List

- A Singly-linked list is a collection of nodes linked together in a sequential way where each node of the singly linked list contains a data field and an address field that contains the reference of the next node.

Singly Linked List

The structure of the node in the Singly Linked List is



class Node { **int** data *// variable to store the data of the node*

Node next *// variable to store the address of the next node* }

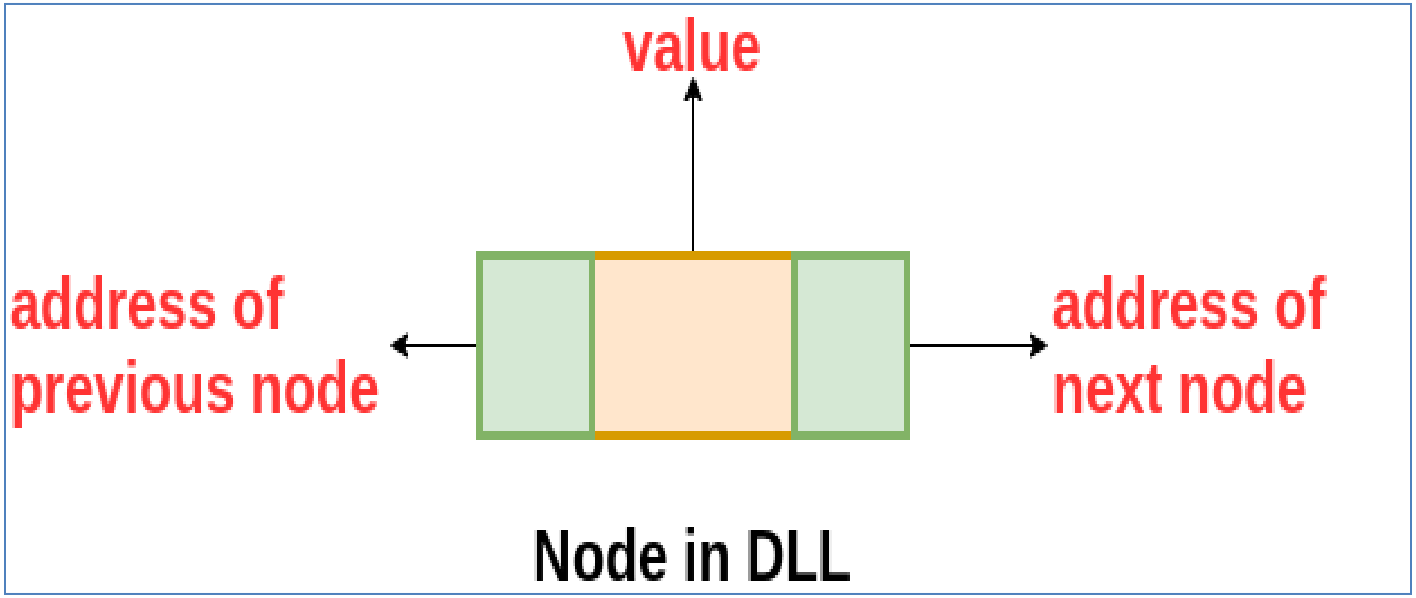
SLL

- The nodes are connected to each other in this form where the value of the next variable of the last node is NULL i.e. ***next = NULL***, which indicates the end of the linked list.

Doubly Linked List

- A Doubly Linked List contains an extra memory to store the address of the previous node, together with the address of the next node and data which are there in the singly linked list.
- So, here we are storing the address of the next as well as the previous nodes.

Structure of DLL



class DLLNode {

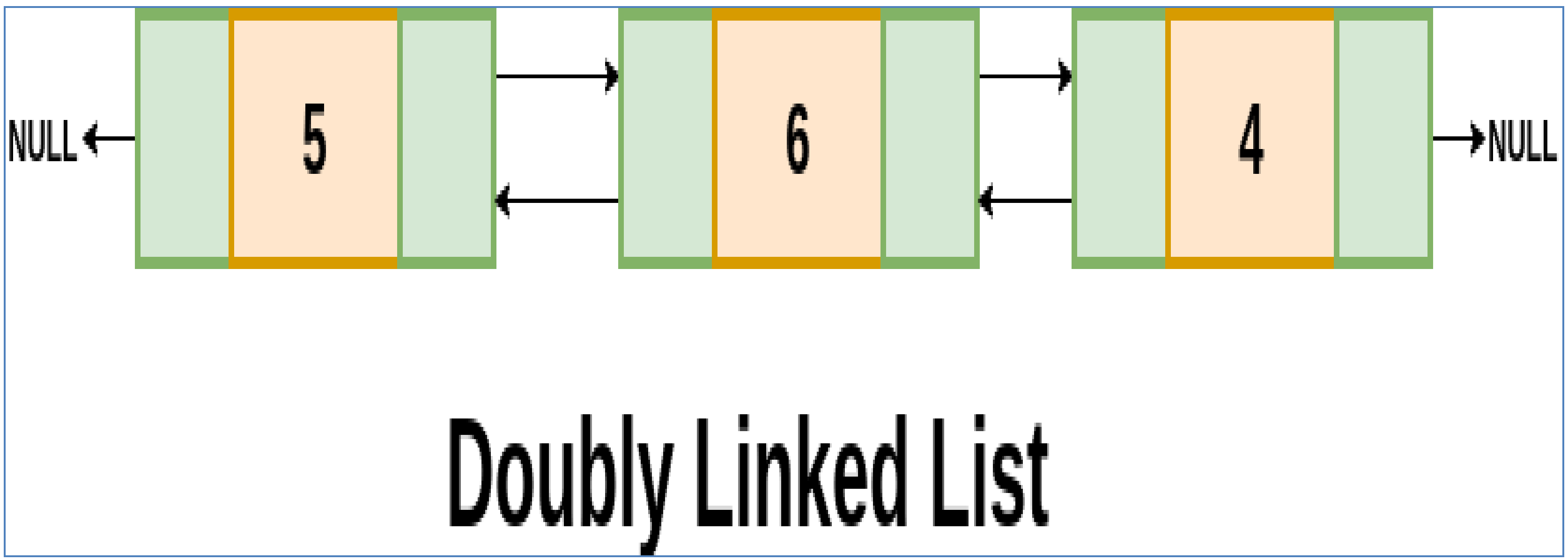
int val *// variable to store the data of the node*

DLLNode prev *// variable to store the address of the previous node*

DLLNode next *// variable to store the address of the next node* }

- The nodes are connected to each other in this form where the first node has **prev = NULL** and the last node has **next = NULL**.

The nodes are connected to each other in this form where the first node has **prev = NULL** and the last node has **next = NULL**



Advantages over Singly Linked List-

- t can be traversed both forward and backward direction.
- The delete operation is more efficient if the node to be deleted is given.
- The insert operation is more efficient if the node is given before which insertion should take place

Disadvantages over Singly Linked List-

- It will require more space as each node has an extra memory to store the address of the previous node.
- The number of modification increase while doing various operations like insertion, deletion, etc.

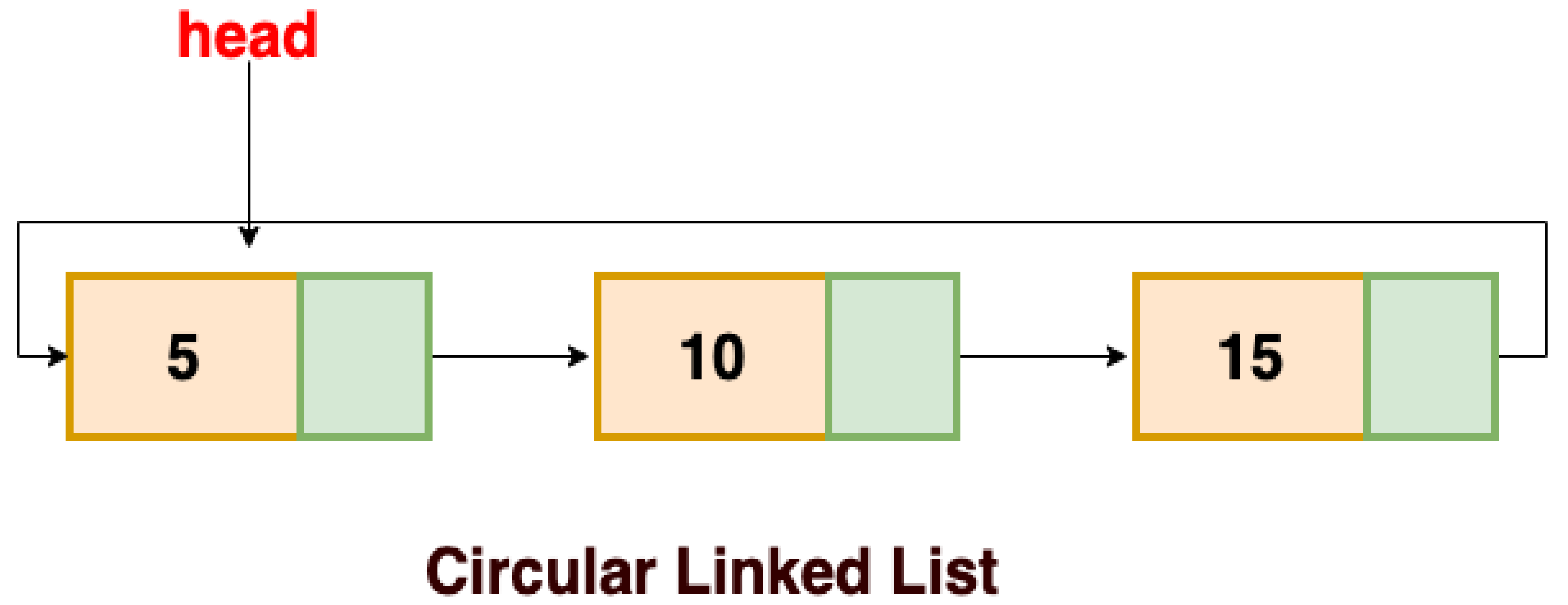
Circular Linked List

- A circular linked list is either a singly or doubly linked list in which there are no ***NULL*** values. Here, we can implement the Circular Linked List by making the use of Singly or Doubly Linked List.

CLL

- In the case of a singly linked list, the next of the last node contains the address of the first node
- And in case of a doubly-linked list, the next of last node contains the address of the first node and prev of the first node contains the address of the last node.

CLL



Advantages of a Circular linked list

- The list can be traversed from any node.
- Circular lists are the required data structure when we want a list to be accessed in a circle or loop.
- We can easily traverse to its previous node in a circular linked list, which is not possible in a singly linked list.

Disadvantages of Circular linked list

- If not traversed carefully, then we could end up in an infinite loop because here we don't have any ***NULL*** value to stop the traversal.
- Operations in a circular linked list are complex as compared to a singly linked list and doubly linked list like reversing a circular linked list, etc.

Basic Operations on Linked List

- **Traversal**: To traverse all the nodes one after another.
- **Insertion**: To add a node at the given position.
- **Deletion**: To delete a node.
- **Searching**: To search an element(s) by value.
- **Updating**: To update a node.
- **Sorting:** To arrange nodes in a linked list in a specific order.
- **Merging:** To merge two linked lists into one.

// A linked list node

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

Linked List Traversal

- The algorithm for traversing a list
- Start with the head of the list. Access the content of the head node if it is not null.
- Then go to the next node(if exists) and access the node information
- Continue until no more nodes (that is, you have reached the null node)

Traversal

```
void traverseLL(Node head) {
    while(head != NULL)
    {
        print(head.data)
        head = head.next
    }
}
```

Traversal of a linked list

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


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

```
// This function prints contents of linked list
   starting from // the given node
void printList(struct Node* n)
{
   while (n != NULL) {
       printf(" %d ", n->data);
       n = n->next;
   }
}
```

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

```
// allocate 3 nodes in the heap
   head = (struct Node*)malloc(sizeof(struct
   Node));
   second = (struct Node*)malloc(sizeof(struct
   Node));
   third = (struct Node*)malloc(sizeof(struct
   Node));
```

```
head->data = 1; // assign data in first node
   head->next = second; // Link first node with
                                              second
   second->data = 2; // assign data to second
                                              node
second->next = third;
   third->data = 3; // assign data to third node
   third->next = NULL;
   printList(head);
   return 0;
}
```

Adding a Node

A node can be added in three ways

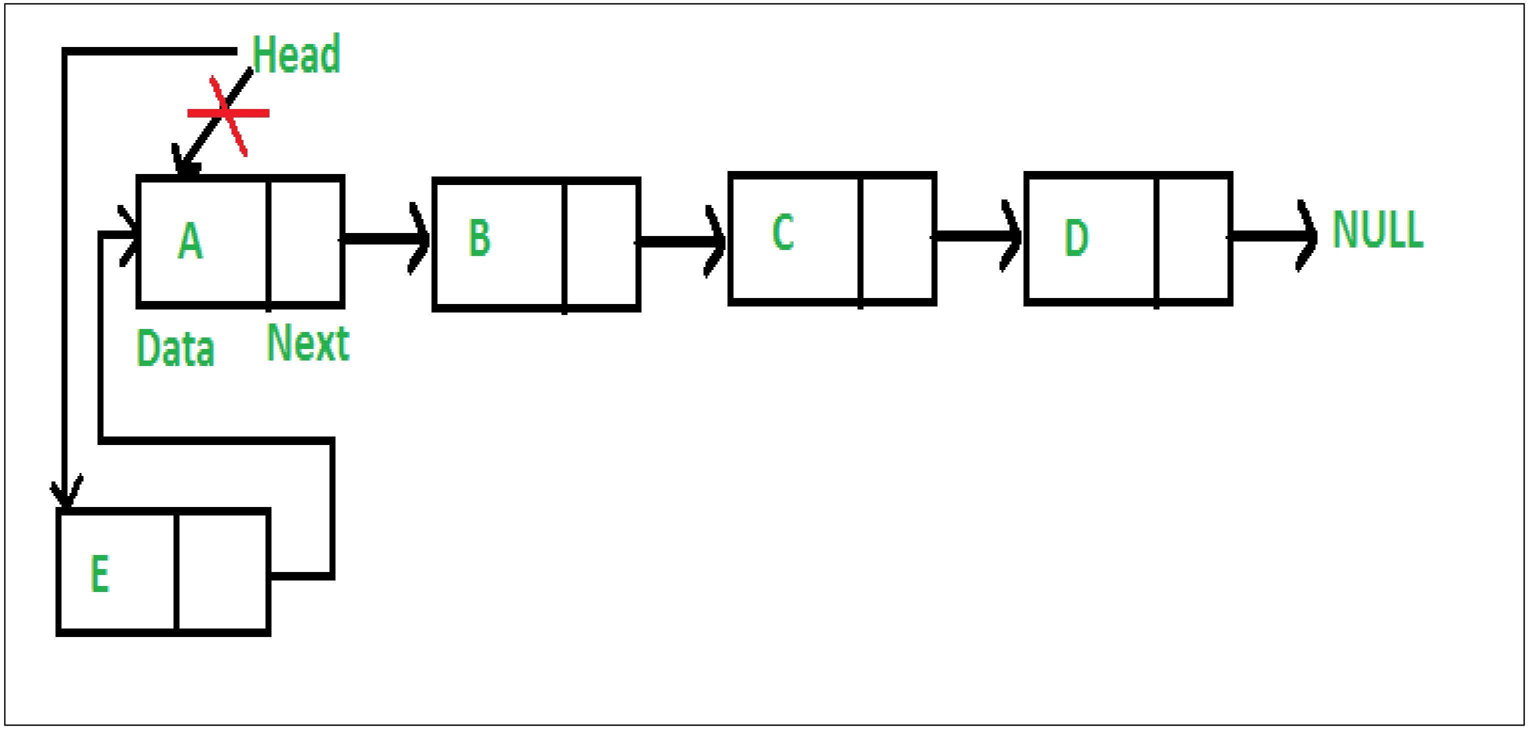
1) At the front of the linked list

2) After a given node.

3) At the end of the linked list.

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

- The new node is always added before the head of the given Linked List. And newly added node becomes the new head of the Linked List.
- 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



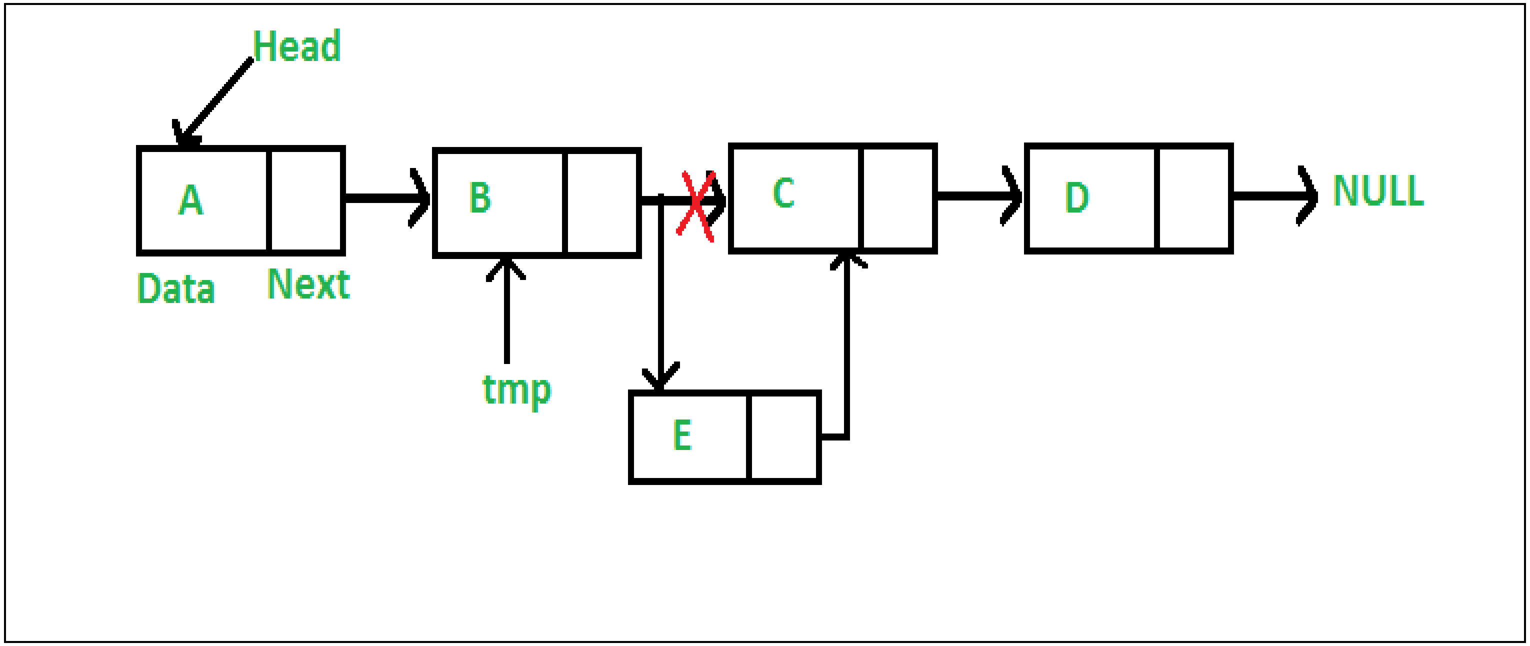
Head
A
B
C
D
NULL
Data
Next
E

Adding at the beginning

```
/* 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(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 */
   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.

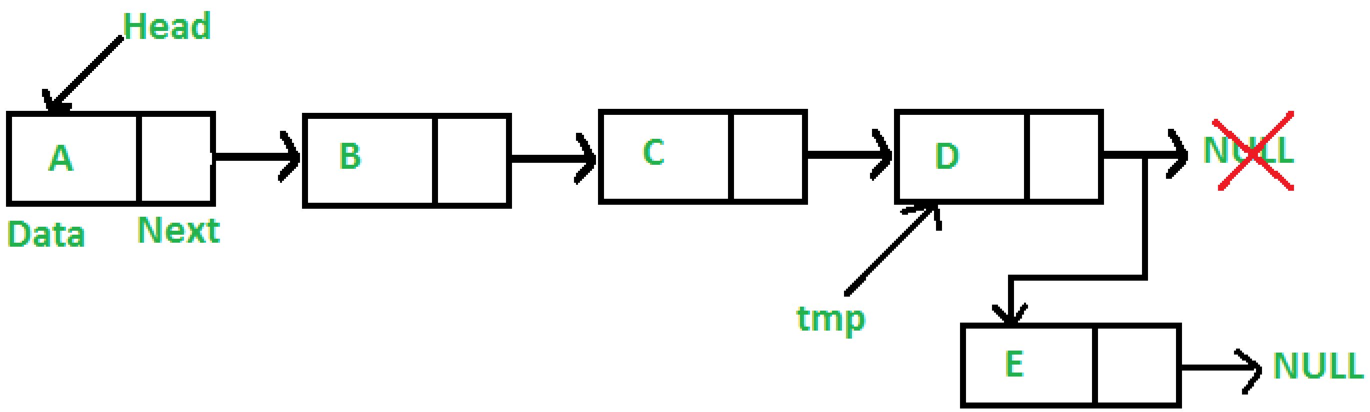
Add a node after a given node: (5 steps process)



Insert after a Previous Node

```
/* Given a node prev_node, insert a new node after the given
prev_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. move the next of prev_node as new_node */
        prev_node->next = new_node;
}
```

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



As Last Node

```
/* Given a reference (pointer to pointer) to the head
  of a list 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)
    {
       *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;
}
```

```
// A complete working C program to demonstrate all insertion methods
// on Linked List
#include <stdio.h>
#include <stdlib.h>

// A linked list node
struct Node
{
int data;
struct Node *next;
};

/* 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(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 */
	new_node->next = (*head_ref);
```

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

/* Given a node prev_node, insert a new node after the given
prev_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. move the next of prev_node as new_node */
```

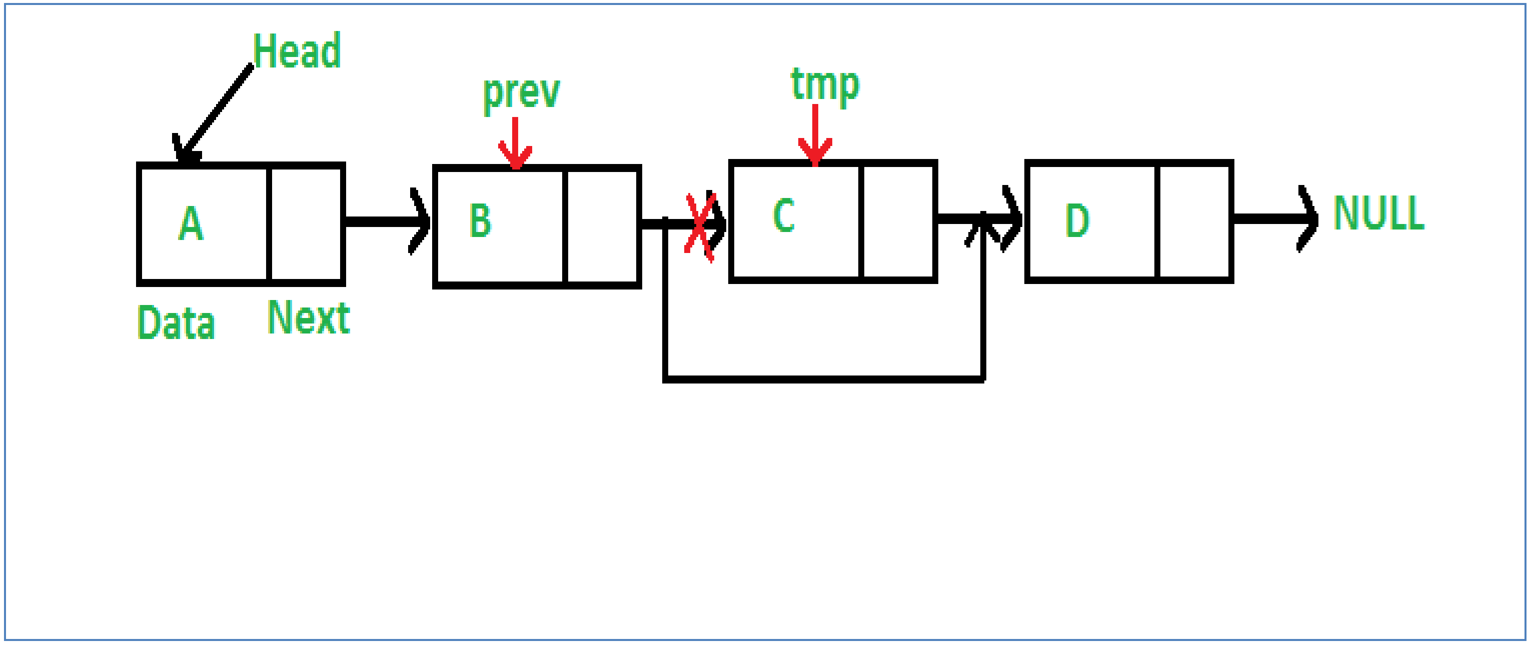
```
prev_node->next  = new_node;
}

/* Given a reference  (pointer to pointer) to the head
of a list 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)
        {
        *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;
}

// This function prints contents of linked list starting from head
void printList(struct Node *node)
{
while (node != NULL)
{
	printf(" %d ", node->data);
	node = node->next;
}
}

/* Driver program to test above functions*/
int main()
{
/* Start with the empty list */
struct Node* head = NULL;
```

```
// Insert 6. So linked list becomes 6->NULL
append(&head, 6);

// Insert 7 at the beginning. So linked list becomes 7->6->NULL
push(&head, 7);

// Insert 1 at the beginning. So linked list becomes 1->7->6->NULL
push(&head, 1);

// Insert 4 at the end. So linked list becomes 1->7->6->4->NULL
append(&head, 4);

// Insert 8, after 7. So linked list becomes 1->7->8->6->4->NULL
insertAfter(head->next, 8);

printf("\n Created Linked list is: ");
printList(head);

return 0;
}
```

Deletion of a Node

- 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 node.
  3) Free memory for the node to be deleted.


Head
prev
tmp
A
B
C
D
NULL
Data
Next

Demonstration of deletion in singly linked list

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


// A linked list node
struct Node {
    int data;
    struct Node* next;
};
```

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

- /* Given a reference (pointer to pointer) to the head of a list and a key, deletes the first occurrence of key in linked list */

```
void deleteNode(struct Node** head_ref, int key)
{
    // Store head node
    struct Node *temp = *head_ref, *prev;
```

```
// If head node itself holds the key to be
deleted
if (temp != NULL && temp->data == key) {
    *head_ref = temp->next; // Changed head
    free(temp); // free old head
    return;
}
```

```
// Search for the key to be deleted, keep track of
   the previous node as we need to change
   'prev->next'
   while (temp != NULL && temp->data != key) {
        prev = temp;
        temp = temp->next;
   }
```

- // If key was not present in linked list

```
if (temp == NULL)
      return;

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

- // This function prints contents of linked list starting // from the given node

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

- // Driver code

```
int main()
{
   /* Start with the empty list */
   struct Node* head = NULL;
   push(&head, 7);
   push(&head, 1);
   push(&head, 3);
```

```
push(&head, 2);
    puts("Created Linked List: ");
    printList(head);
    deleteNode(&head, 1);
    puts("\nLinked List after Deletion of 1: ");
    printList(head);
    return 0;
}
```

Searching in Singly Linked List

Algorithm

1. Iterate the linked list using a loop.
2. If any node has the given key value, return 1.
3. If the program execution comes out of the loop (the given key is not present in the linked list), return -1.

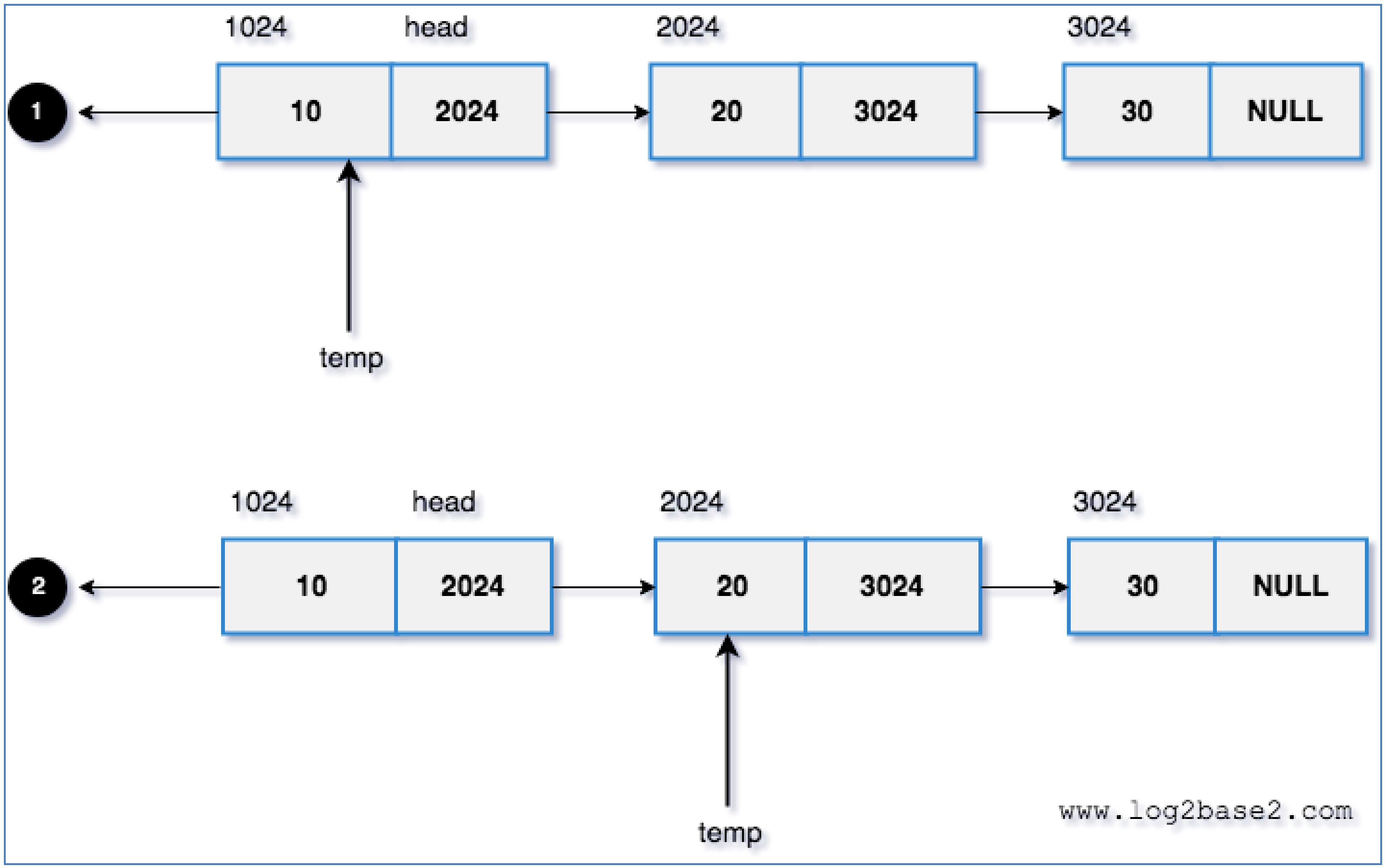
- ***Search Found => return 1.***
- ***Search Not Found => return -1.***

Iterate the linked list using a loop.

```
int searchNode(struct node *head, int key)
{
      struct node *temp = head;
    while(temp != NULL)
     { temp = temp->next; }
}
```

Return 1 on search found

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


1024
head
2024
3024
1
10
2024
20
3024
30
NULL
temp
1024
head
2024
3024
2
10
2024
20
3024
30
NULL
temp
www.log2base2.com

1. temp holding the address of the head node.

temp->data = 10. key = 20. temp->data != key,

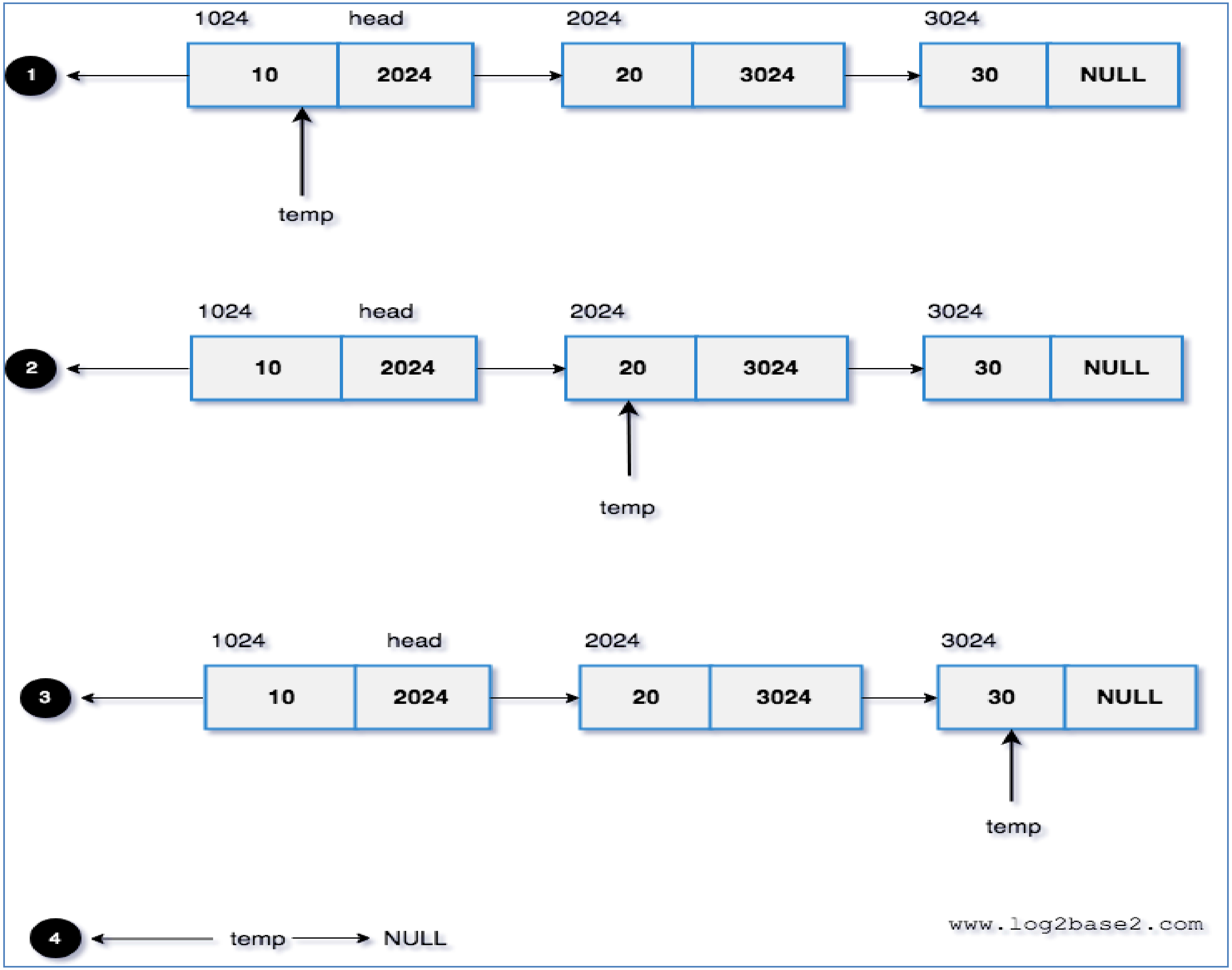
so move the temp variable to the next node.

2. Now, temp->data = 20. key = 20.

temp->key == key. "Seach Found".

Return -1 on search not found

```
• int searchNode(struct node *head, int key)
{
    struct node *temp = head;
   while(temp != NULL)
 {
 if(temp->data == key) return 1;
temp = temp->next; } return -1;
• }
```


1024
head
2024
3024
1
10
2024
20
3024
30
NULL
temp
2
3
4
www.log2base2.com

Key value =100

1. temp->data = 10. key = 100. temp->data != key. Hence move the temp variable to the next node.
2. temp->data = 20. key = 100. temp->data != key. Hence move the temp variable to the next node.

Searching an Element

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


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

```c
void addLast(struct node **head, int val)
{
    //create a new node
    struct node *newNode = malloc(sizeof(struct
    node));
    newNode->data = val;
    newNode->next     = NULL;
```

```
//if head is NULL, it is an empty list
  if(*head == NULL)
     *head = newNode;
  //Otherwise, find the last node and add the n
  ewNode
  else
  {
     struct node *lastNode = *head;
     //last node's next address will be NULL.
```

```
while(lastNode->next != NULL)
        {
           lastNode = lastNode->next;
        }
//add the newNode at the end of the linked list
        lastNode->next = newNode;
    }


}
```

```
int searchNode(struct node *head,int key)
{
    struct node *temp = head;
    //iterate the entire linked list and print the da
   ta
```

```
while(temp != NULL)
    {
        //key found return 1.
        if(temp->data == key)
            return 1;
        temp = temp->next;
    }
    //key not found
    return -1;
}
```