SNS COLLEGE OF ENGINEERING
Kurumbapalayam (Po), Coimbatore – 641 107
AN AUTONOMOUS INSTITUTION
Accredited by NBA–AICTE and Accredited by NAAC–UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND ENGINEERING

COURSE NAME : 19CS307 - DATA STRUCTURES

II YEAR / III SEMESTER

UNIT I – LINEAR STRUCTURES-LIST

1.1 INTRODUCTION

Data Structures

Data structure is a representation of logical relationship existing between individual elements of data. In other words, a data structure defines a way of organizing all data items that considers not only the elements stored but also their relationship to each other. The term data structureis used to describe the way data is stored.

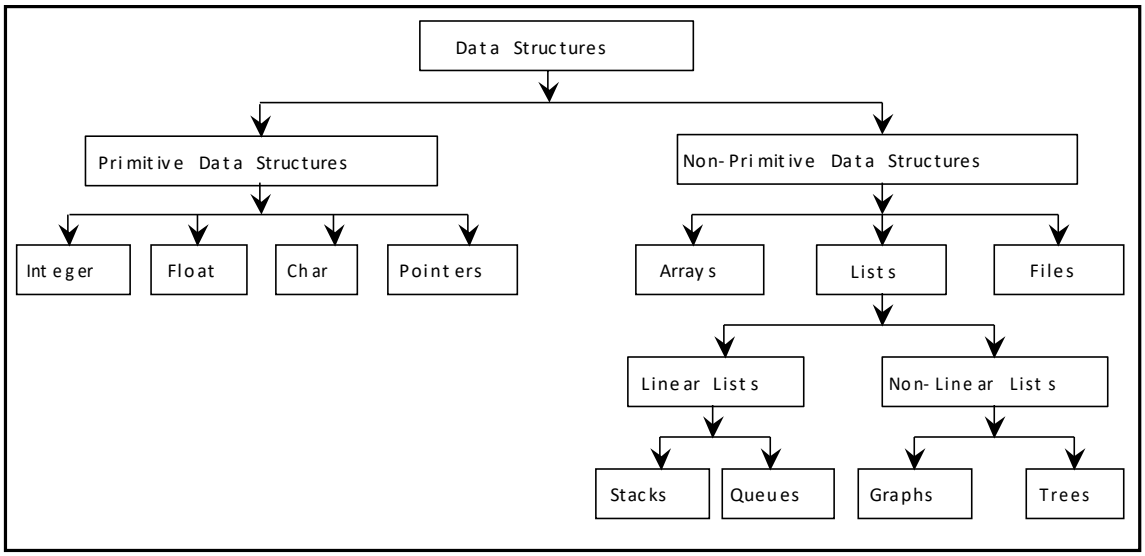
To develop a program of an algorithm we should select an appropriate data structure for that algorithm. Therefore, data structure is represented as: **Algorithm + Data structure = Program**

Types of Data Structures

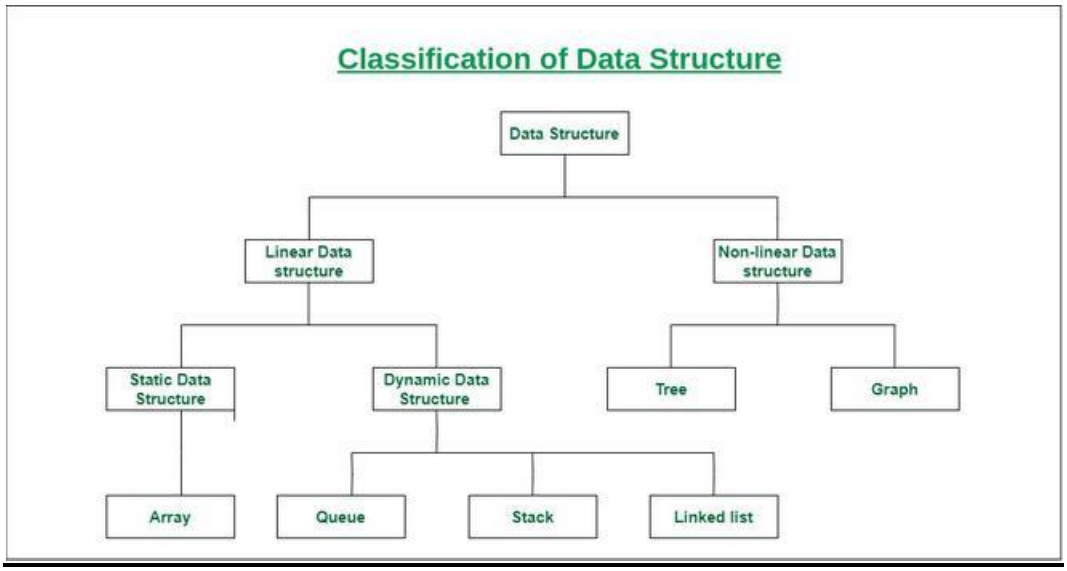
- **Primitive data structures.**
- **Non-primitive data structures.**

Primitive Data Structures are the basic data structures that directly operate upon the machine instructions. They have different representations on different computers.

Non-primitive data structures are more complicated data structures and are derived from primitive data structures. They emphasize on grouping same or different data items with relationship between each data item.



Classification of Data Structure



Types of Data organization in memory

Contiguous- Continuous memory allocation

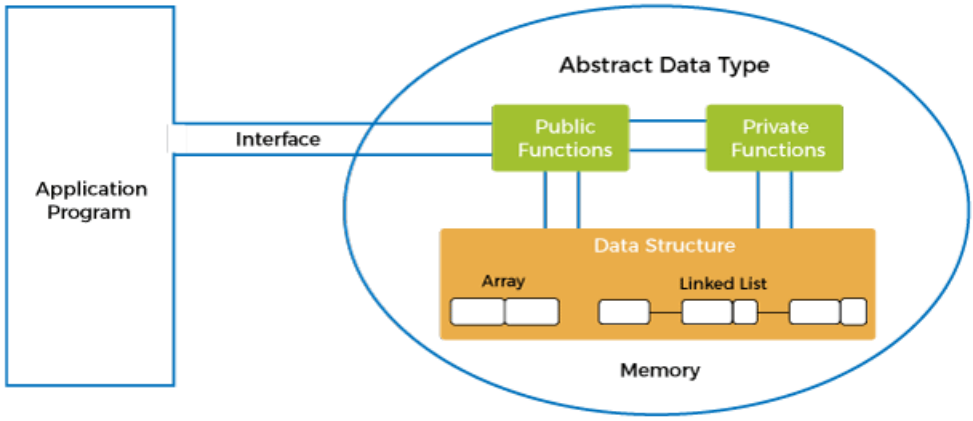
Ex: Array

Non Contiguous – Data can be scattered in memory, but we linked to each other in some way

Ex: Linked list,Tree,Graph

1.2 ABSTRACT DATA TYPE (ADT):

- An abstract data type is an abstraction of a data structure that provides only the interface to which the data structure must adhere The interface does not give any specific details about something should be implemented or in what programming language



- The abstract data type is special kind of data type, whose behavior is defined by a set of values and set of operations.
- The keyword "Abstract" is used as we can use these data types, we can perform different operations. But how those operations are working that is totally hidden from the user.
- The ADT is made of with primitive data types, but operation logics are hidden.

- Examples of ADT are Stack, Queue, and List etc.

  - A List is an abstract data type that is implemented using a dynamic array and linked list.
  - A queue is implemented using linked list-based queue, array-based queue, and stack-based queue.
  - A Map is implemented using Tree map, hash map, or hash table.

..

Abstract data type with a real-world example

If we consider the smartphone. We look at the high specifications of the smartphone, such as:

- 4 GB RAM

- Snapdragon 2.2ghz processor
- 5 inch LCD screen
- Dual camera
- Android 8.0

The above specifications of the Smartphone are the data, and we can also perform the following operations on the Smartphone:

- call(): We can call through the smartphone.
- text(): We can text a message.
- photo(): We can click a photo.
- video(): We can also make a video.

The Smartphone is an entity whose data or specifications and operations are given above. The abstract/logical view and operations are the abstract or logical views of a Smartphone.

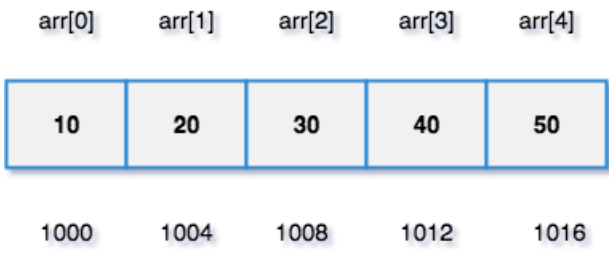
1.3 ARRAY BASED IMPLEMENTATION OF LIST

Array

An array is a collection of variables in the same datatype. We can't group different data types in the array. Like, a combination of integer and char, char and float etc.

Hence array is called as the homogeneous data type.

Ex: **int** arr[**5**]={**10**,**20**,**30**,**40**,**50**};



Using index value, we can directly access the desired element in the array.

Array index starts from 0, not 1.

To access the 1st element, we can directly use index 0. i.e a[0]

To access the 5th element, we can directly use index 4. i.e a[4]

We can manipulate the Nth element by using the index N - 1. {Where N > 0}

In general, an array of size N will have elements from index 0 to N-1.

Insertion operation

Insert a given element at a specific position in an array.

Algorithm

1. Get the **element** value which needs to be inserted.

2. Get the **position** value.

3. Check whether the position value is valid or not.

4. If it is valid,

   Shift all the elements from the last index to position index by 1 position to the right.

   insert the new element in arr[position]

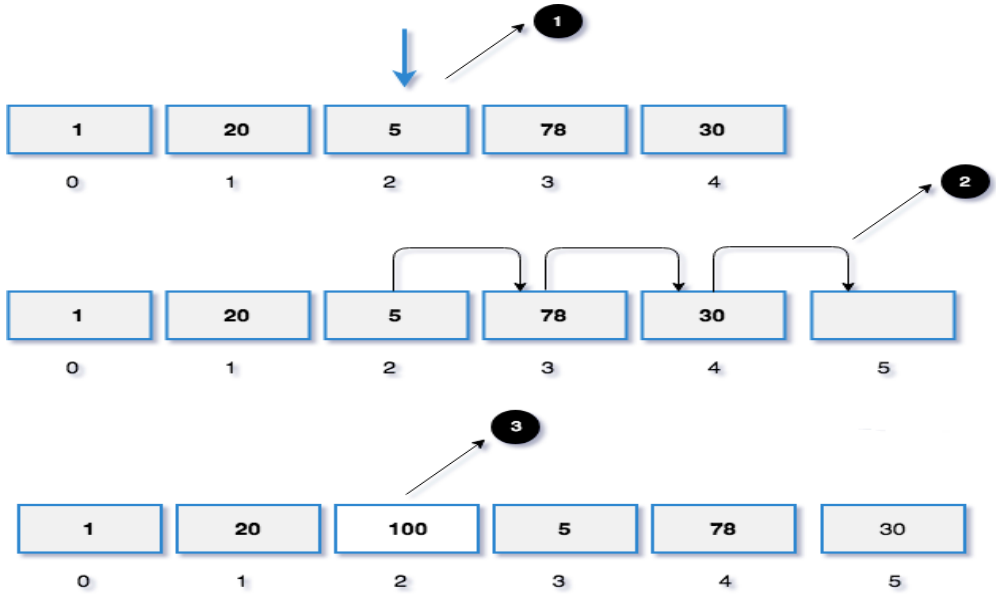
5. Otherwise,

   Invalid Position

Input

int arr[5] = {10, 20, 30, 40, 50}

Element = 100 position = 2.



Implementation:

```
#include<stdio.h>
#define size 5
int main()
{
int arr[size] = {1, 20, 5, 78, 30};
int element, pos, i;
printf("Enter position and element\n");
scanf("%d%d",&pos,&element);
if(pos <= size && pos >= 0)
{
//shift all the elements from the last index to pos by 1 position to right
for(i = size; i > pos; i--)
arr[i] = arr[i-1];
//insert element at the given position
```

```
arr[pos] = element;
for(i = 0; i <= size; i++)
printf("%d ", arr[i]);
}else
printf("Invalid Position\n");
return 0;
}
```

Delete operation

Delete a given element from an array.

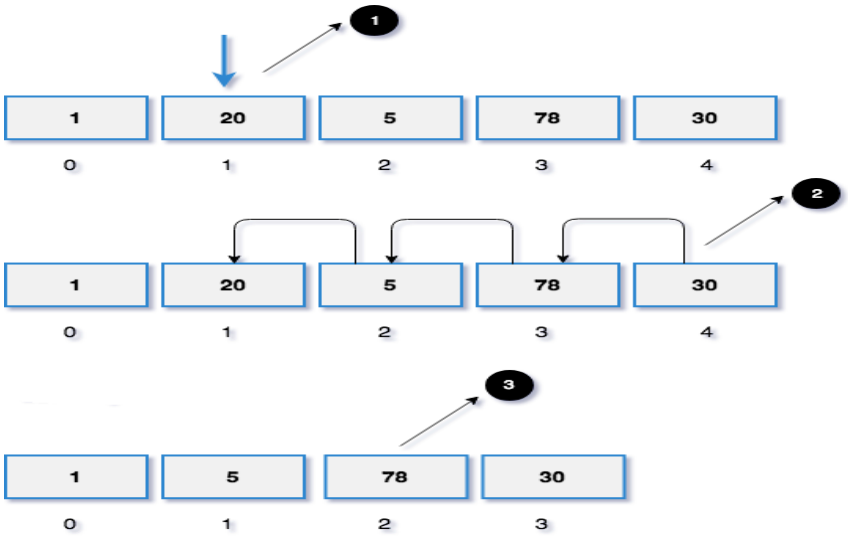
Algorithm

1. Find the given element in the given array and note the index.
2. If the element found,
   Shift all the elements from index + 1 by 1 position to the left.
   Reduce the array size by 1.
3. Otherwise, print "Element Not Found"

Input

Array : {1, 20, 5, 78, 30}

Element : 78



Implementation

```
#include<stdio.h>

#define size 5

int main()

{

int arr[size] = {1, 20, 5, 78, 30};

int key, i, index = -1;

printf("Enter element to delete\n");

scanf("%d",&key);

for(i = 0; i < size; i++)

{

if(arr[i] == key)

{

index = i;

break;

}

}
```

```
if(index != -1)
{
//shift all the element from index+1 by one position to the left
for(i = index; i < size - 1; i++)
arr[i] = arr[i+1];
printf("New Array : ");
for(i = 0; i < size - 1; i++)
printf("%d ",arr[i]);
}
else
printf("Element Not Found\n");
return 0;
}
```

Search operation

Search whether the given key is present or not in the array.

Input

arr[5] = {10, 30, 5, 100, 4};

key = 30

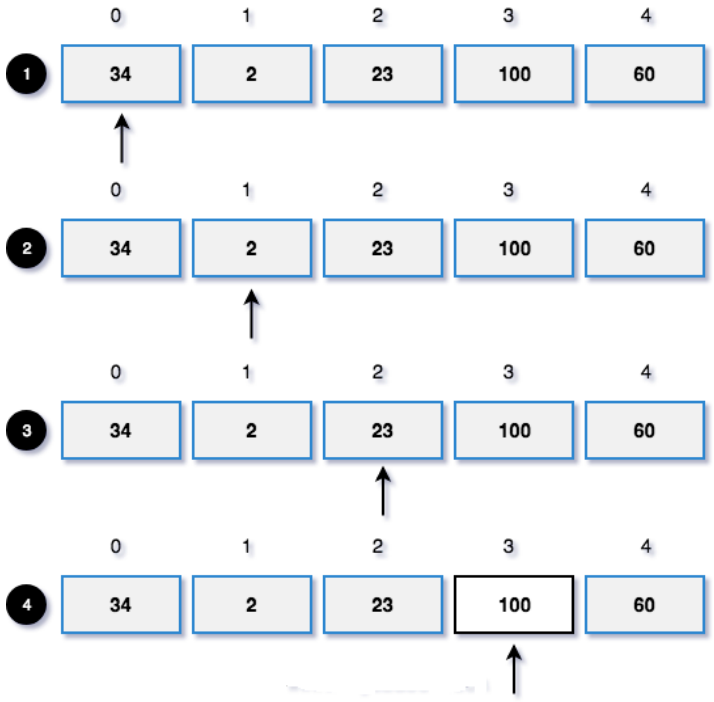
Algorithm

1. Iterate the array using the loop.
2. Check whether the given key present in the array i.e. arr[i] == key.
3. If yes,

   print "Search Found".

4. Else

print "Search Not Found".



Implementation

```
#include<stdio.h>

#define size 5

int main()

{

int arr[size] = {34, 2, 23, 100, 60};

int key,i,flag = 0;

printf("Enter element to search\n");

scanf("%d",&key);

/*

* iterate the array elements using loop

* if any element matches the key, set flag as 1 and break the loop

* flag = 1 indicates that the key present in the array
```

```
* if execution comes out of loop and the flag remains 0, print search not found
*/
for(i = 0; i < size; i++)
{
if(arr[i] == key)
{
flag = 1;
break;
}
}
if(flag == 1)
printf("Search Found\n");
else
printf("Search Not Found\n");
return 0;
}
```

Advantages

- There is no wasted space for an individual element (do not need space for pointers)

Disadvantages

- Lacking efficiency for insertion/deletion operations and memory allocation.

Application

- Arrays are used to implement data structures like a stack, queue, etc.
- Arrays are used for matrices and other mathematical implementations.
- Arrays are used in lookup tables in computers.
- Arrays can be used for CPU scheduling

1.4 LINKED LISTS

A linked list is a linear data structure, in which the elements are not stored at contiguous memory locations

The linked list is a linear data structure where each node has two parts.

1. Data

2. Reference to the next node

Data: we can store the required information. It can be any data type such as int,float,double.

int age; char name[20];

Reference to the next node: It will hold the next nodes address. Hence it is a type pointer

Here, we need to group two different data types (heterogeneous).

We can use **structure** data type to group the different data types. So, every node in a linked list is a structure data type.

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

Types of Linked List

Following are the various types of linked list.

Simple or Singly Linked List − Item navigation is forward only.

Doubly Linked List − Items can be navigated forward and backward.

Circular Linked List − Last item contains link of the first element as next and the first element has a link to the last element as previous.

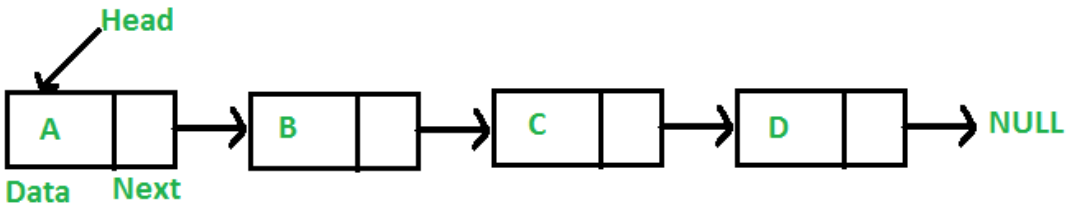
1.4.1 SINGLY LINKED LIST

Singly Linked List in C is one of the simplest linear data structures, that we use for storing our data in an easy and efficient way. Linked List in C comprises nodes like structures, which can further be divided into 2 parts in the case of a singly linked list. These two parts are-:

Node – for storing the data.

Pointer – for storing the address of the next node.

In a Singly linked list there is only one pointer type variable that contains the address of the next node.

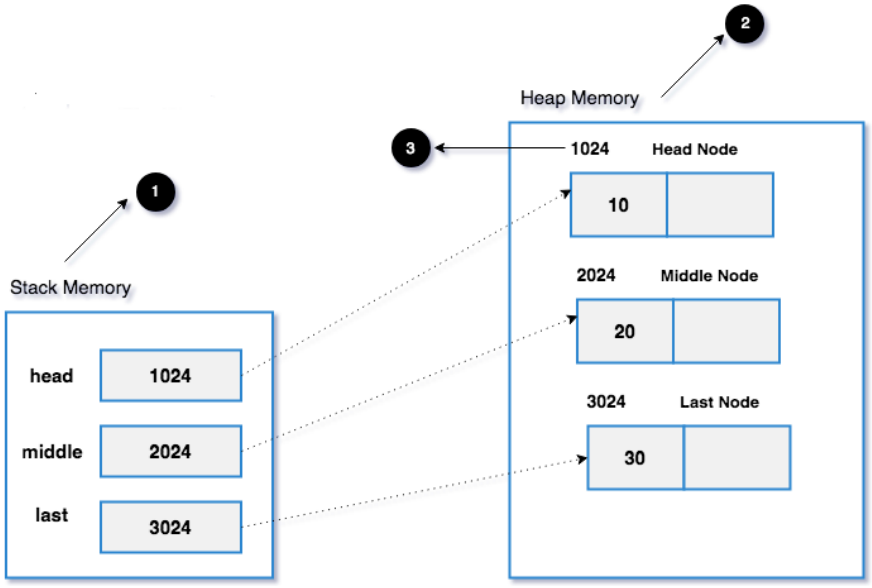


Let's create and allocate memory for 3 nodes

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

struct node *head,*middle,*last;

head   = malloc(sizeof(struct node));
middle = malloc(sizeof(struct node));
last   = malloc(sizeof(struct node));
```

Assign values to each node

```
head->data   = 10;
middle->data = 20;
last->data   = 30;
```



1. Stack memory stores all the local variables and function calls (static memory).

Example: int a = 10;

2. Heap memory stores all the dynamically allocated variables.

Example: int *ptr = malloc(sizeof(int)); Here, memory will be allocated in the heap section. And the ptr resides in the stack section and receives the heap section memory address on successful memory allocation.

3. Address of the dynamic memory which will be assigned to the corresponding variable.

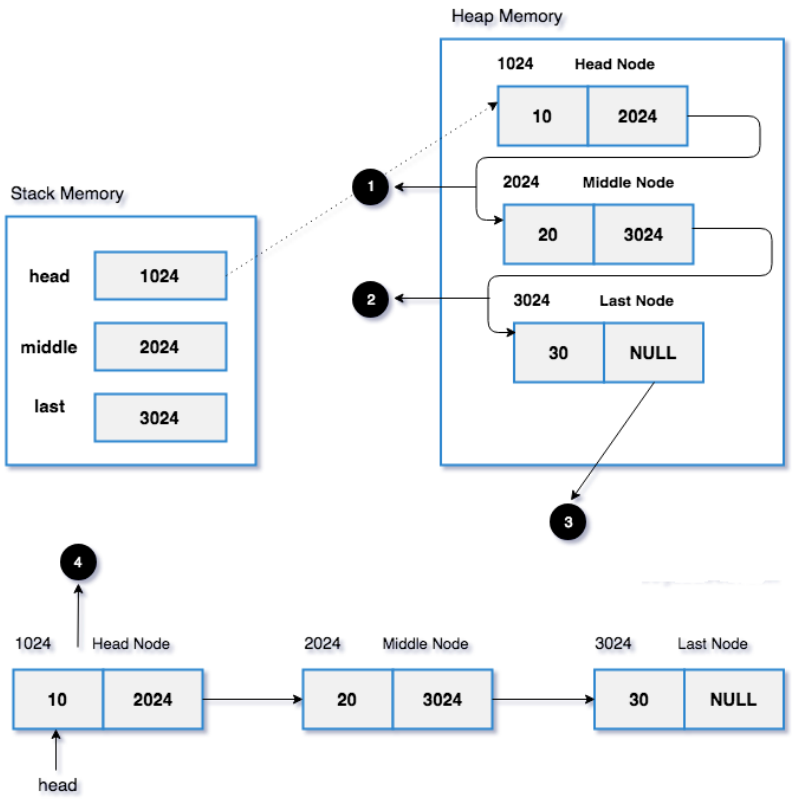
Linking each nodes

headnode -> middlenode-> lastnode-> NULL

head->next = middle;

middle->next = last;

last->next = NULL; //NULL indicates the end of the linked list



1. head => next = middle. Hence head => next holds the memory address of the middle node (2024).

2. middle => next = last. Hence middle => next holds the memory address of the last node (3024).

3. last => next = NULL which indicates it is the last node in the linked list.

4. The simplified version of the heap memory section.

Printing each node data in a linked list

To print each node's data, we have to traverse the linked list till the end.

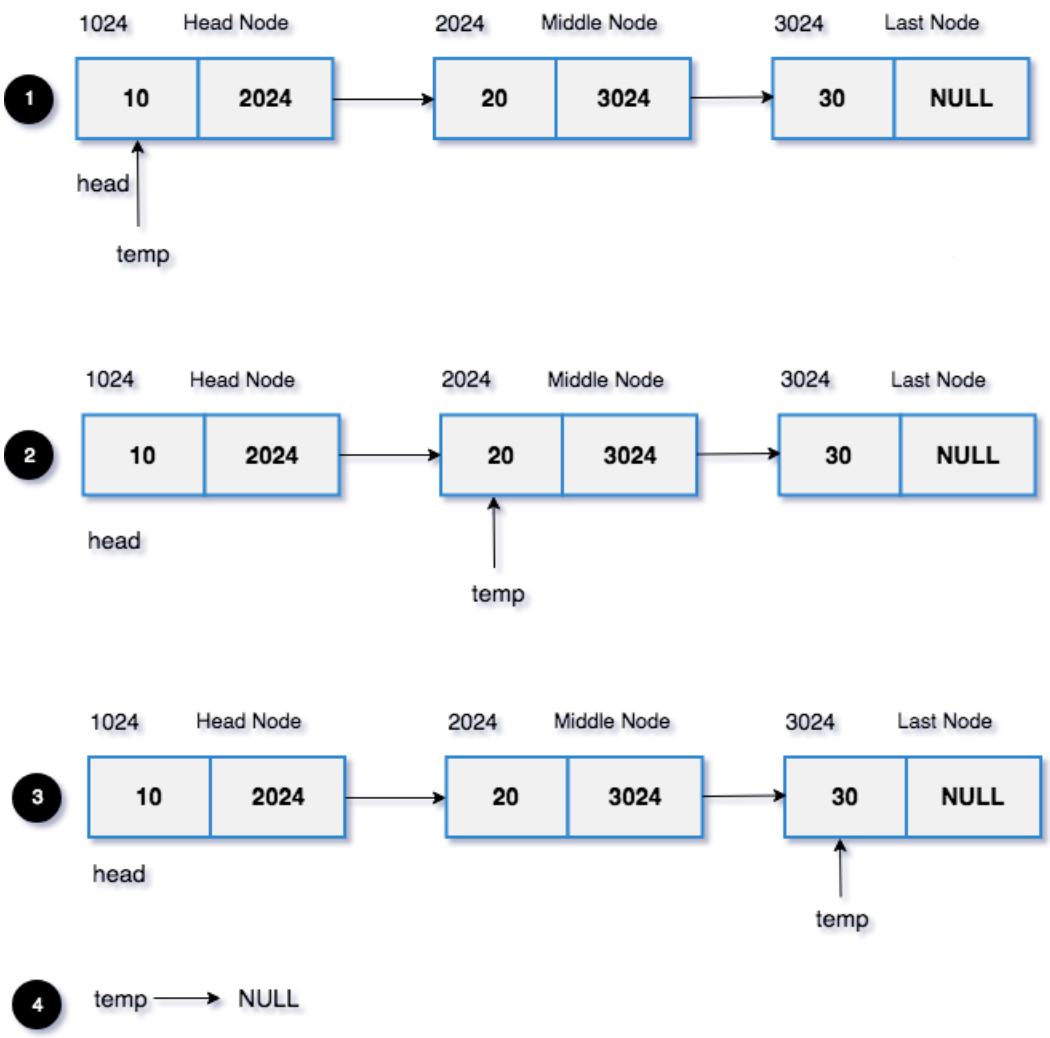
Algorithm

1. Create a temporary node (temp) and assign the head node's address.

2. Print the data which present in the temp node.

3. After printing the data, move the temp pointer to the next node.

4. Do the above process until we reach the end.



1. temp points to the head node. temp => data = 10 will be printed. temp will point to the next node (Middle Node).

2. temp != NULL. temp => data = 20 will be printed. Again temp will point to the next node (Last Node).

3. temp != NULL. temp => data = 30 will be printed. Again temp will point to the next node which is NULL.

4. temp == NULL. Stop the process we have printed the whole linked list.

Code

```
struct node *temp = head;

while(temp != NULL)
{
   printf("%d ",temp->data);
   temp = temp->next;
}
```

Why do we need to use the temp node instead head?

If we use the head pointer instead of the temp while printing the linked list, we will miss the track of the starting node. (After printing the data head node will point the NULL).

To avoid that, we should not change the head node's address while processing the linked list. We should always use a temporary node to manipulate the linked list.

Sample Linked List Implementation

Example

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

int main()
{
  //node structure
  struct node
  {
     int data;
     struct node *next;
  };

  //declaring nodes
  struct node *head,*middle,*last;

  //allocating memory for each node
  head   = malloc(sizeof(struct node));
  middle = malloc(sizeof(struct node));
  last   = malloc(sizeof(struct node));

  //assigning values to each node
  head->data   = 10;
  middle->data = 20;
  last->data   = 30;
```

```
  //connecting each nodes. head->middle->last
  head->next   = middle;
  middle->next = last;
  last->next   = NULL;

  //temp is a reference for head pointer.
  struct node *temp = head;

  //till the node becomes null, printing each nodes data
  while(temp != NULL)
  {
     printf("%d->",temp->data);
     temp = temp->next;
  }
  printf("NULL");

  return 0;
}
```

Inserting a node at the beginning of a linked list

The new node will be added at the beginning of a linked list.

Example

Assume that the linked list has elements: 20 30 40 NULL
If we insert 100, it will be added at the beginning of a linked list.
After insertion, the new linked list will be
100 20 30 40 NULL

Algorithm

1. Declare a head pointer and make it as NULL.
2. Create a new node with the given data.
3. Make the new node points to the head node.
4. Finally, make the new node as the head node.

1. Declare a head pointer and make it as NULL

struct node

```
{
   int data;
   struct node *next;
};
struct node *head = NULL;
```

2.Create a new node with the given data.

```
void addFirst(struct node **head, int val)
{
     //create a new node

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

     newNode->data = val;

     }
```

3.Make the new node points to the head node

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

     newNode->next = *head;
    }
```

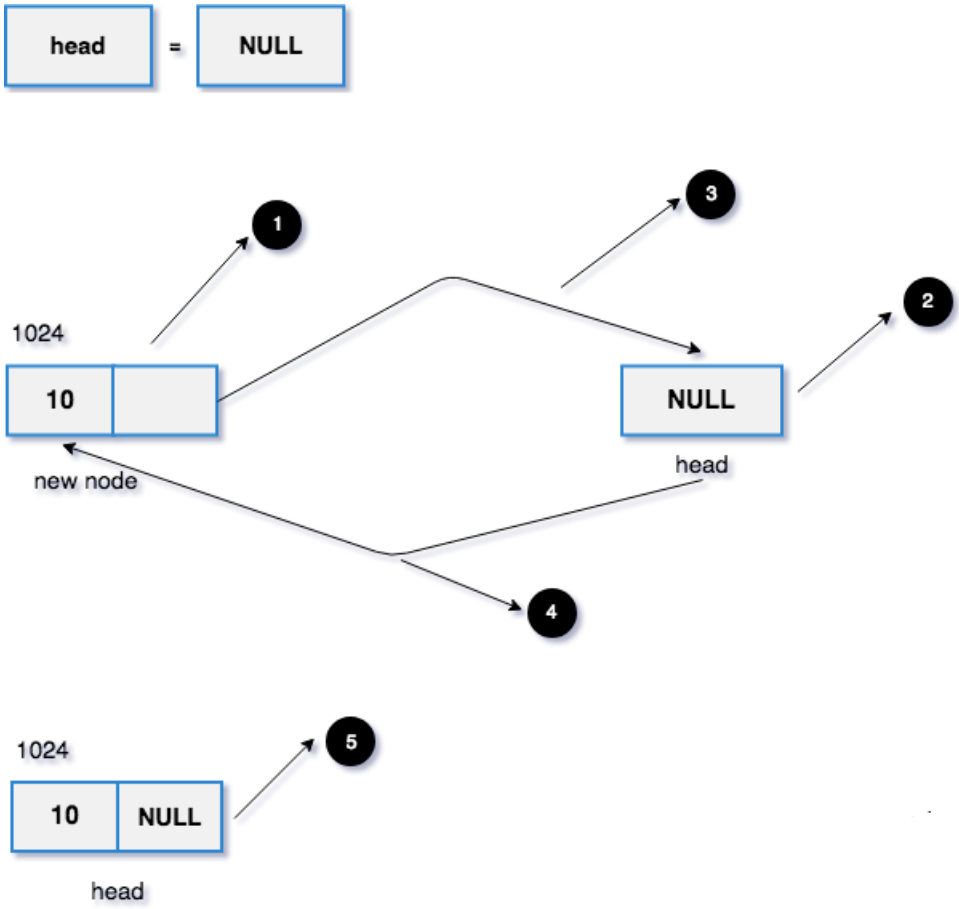
4.Make the new node as the head node

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

     *head = newNode;

}
```

Example

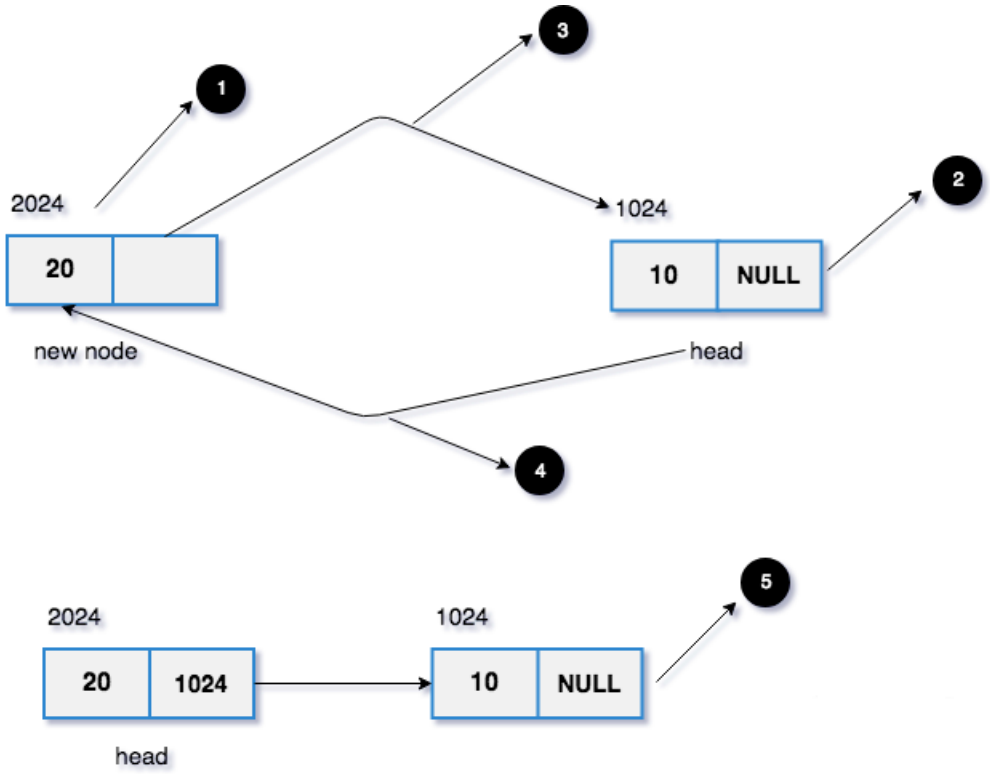
Insert 10

Steps
1. A newly allocated node with data as 10.
2. Head points to NULL.
3. New node -> next points to the head which is NULL. So newnode->next = NULL.
4. Make the head points to the new node. Now, the head will hold the address of the new node w hich is 1024.
5. Finally, the new linked list.



Insert 20

Steps
1. A newly allocated node with data as 20.
2. Head points to the memory address 1024 (It has only one node. 10->NULL).
3. New node -> next points to the head which is 1024. So newnode->next = 1024 (10->NULL) w ill be added back to the new node.
4. Make the head points to the new node. Now, the head will hold the address of the new node w hich is 2024.
5. Finally, the new linked list.



Program to insert a node at the beginning of linked list

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

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


void addFirst(struct node **head, int val)
{
     //create a new node
     struct node *newNode = malloc(sizeof(struct node));
     newNode->data = val;

     newNode->next = *head;

     *head = newNode;
```

```
}

void printList(struct node *head)
{
   struct node *temp = head;

   //iterate the entire linked list and print the data
   while(temp != NULL)
   {
       printf("%d->", temp->data);
       temp = temp->next;
   }
   printf("NULL\n");
}

int main()
{
    struct node *head = NULL;

    addFirst(&head,10);
    addFirst(&head,20);
    printList(head);

    return 0;
}
```

Inserting a node at the end of a linked list

The new node will be added at the end of the linked list.

Example
Input
Linked List : 10 20 30 40 NULL.

50

Output
Linked List : 10 20 30 40 50 NULL.

Algorithm

1. Declare head pointer and make it as NULL.

2. Create a new node with the given data. And make the new node => next as NULL. (Because the new node is going to be the last node.)

3. If the head node is NULL (Empty Linked List), make the new node as the head.

4. If the head node is not null, (Linked list already has some elements),

find the last node.

make the last node => next as the new node.

1. Declare head pointer and make it as NULL.

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

struct node *head = NULL;
```

2. Create a new node

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

3. If the head node is NULL, make the new node as head

```
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;
```

}

4. Otherwise, find the last node and set last node => new node

The last node of a linked list has the reference pointer as NULL. i.e. node=>next = NULL.

To find the last node, we have to iterate the linked till the node=>next != NULL

```
while(node->next != NULL)
{
   node = node->next;
}
```

After that, we have to make the last node-> next as the new node. i.e. last node->next = new node;

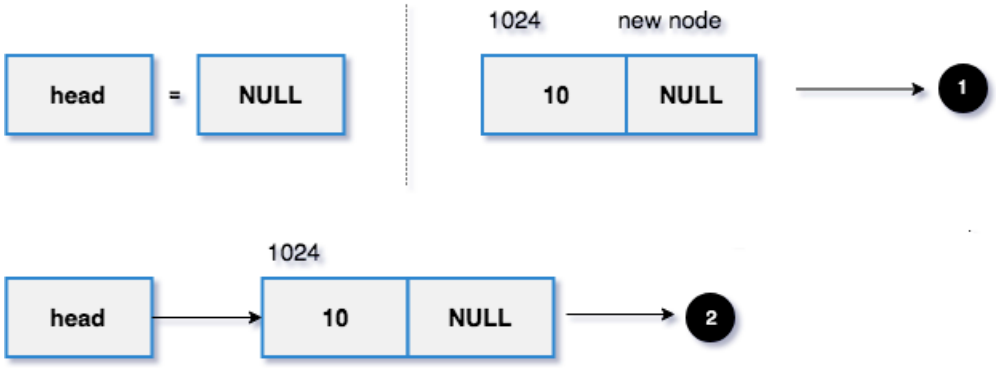
Example

Insert data 10.

The head is NULL initially.

1. The new node with data as 10 and reference is NULL (address 1024).

2. Since it is the first node, make the head node points to the newly allocated node.



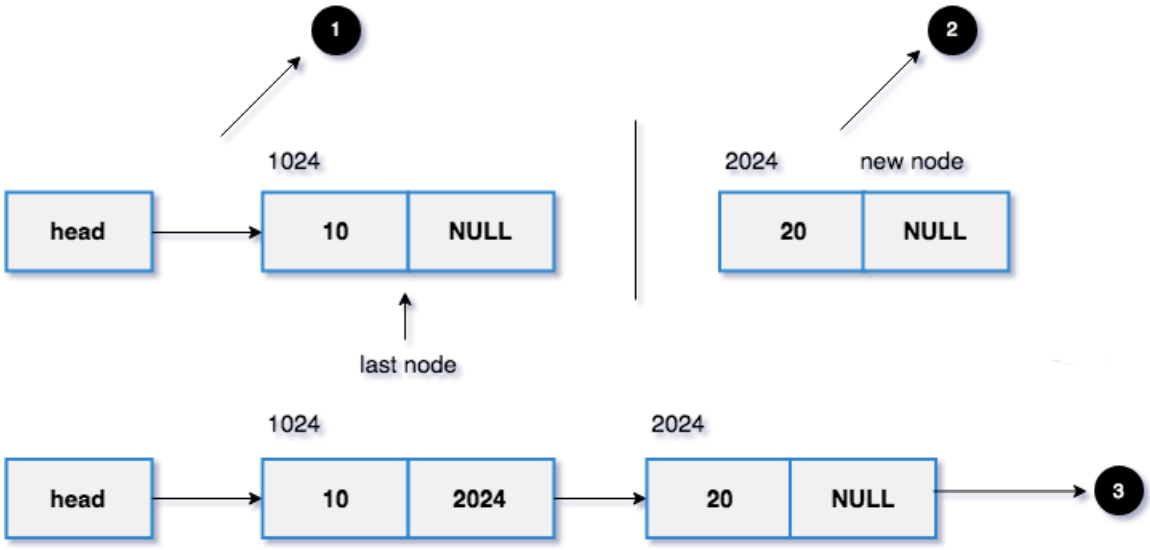
Insert data 20.

1. The head points to the memory address 1024 and it is the last node.

2. The new node with data as 20 and reference is NULL (address 2024).

set last node =>next = new node. The new node added at the end of the linked list.

3. Finally, the new linked list.



Implementation of inserting a node at the end of a linked list

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
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 newNode
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;
}}
void printList(struct node *head)
{
struct node *temp = head;


//iterate the entire linked list and print the data
while(temp != NULL)
{
```

```
printf("%d->", temp->data);
temp = temp->next;
}
printf("NULL\n");
}
int main()
{
struct node *head = NULL;

addLast(&head,10);
addLast(&head,20);
printList(head);
return 0;
}
```

Searching a node in singly linked list

Check whether the given key is present or not in the linked list.

Example

Linked List : 10 20 30 40 NULL.

Input

20

Output

Search Found

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.

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;

   }



}
```

2. 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;
   }
}
```

3. 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;
}
```

Example

Linked List : 10 20 30 NULL

Key : 100

Steps:

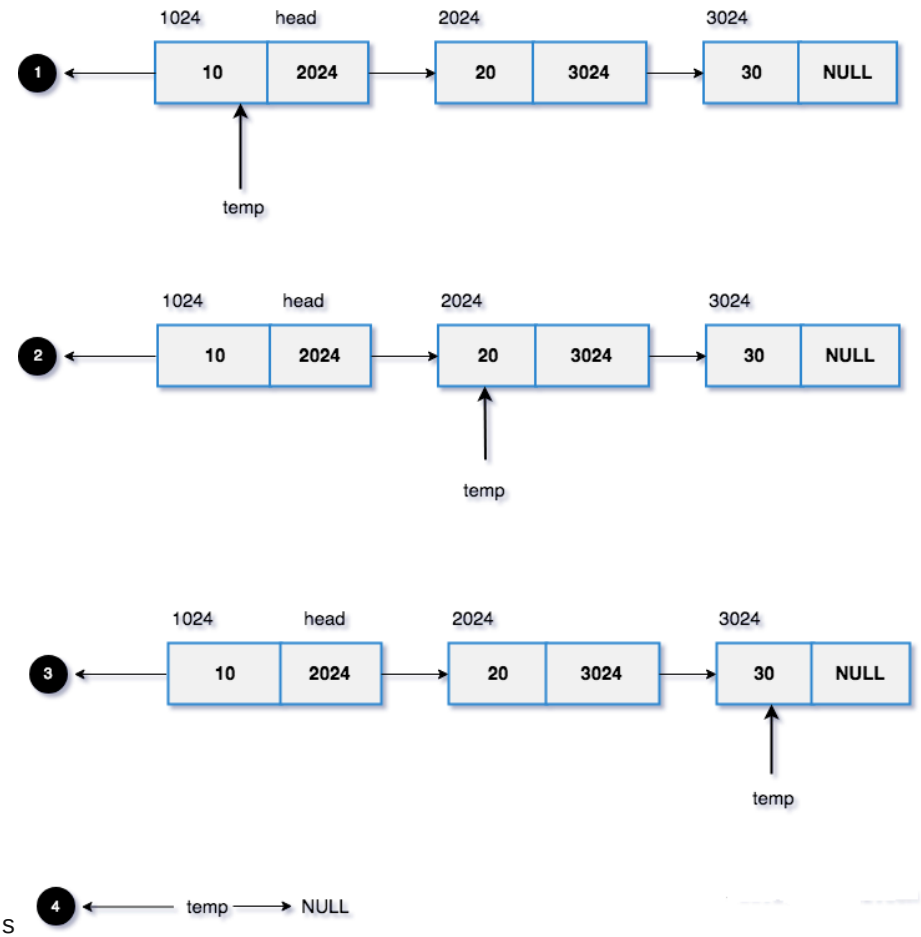
1. temp->data = 10. key = 100. temp->data != key. Hence move the temp variable to the next node.

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

3. temp->data = 30. key = 100. temp->data != key. Hence move the temp variable to the next node which is NULL.

4. Finally, the program execution will come out of the loop. So, it will return -1.

"Search Not Found".



s

Implementation of searching a node in singly linked list

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

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

```
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 newNode
   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 data
   while(temp != NULL)
   {
       //key found return 1.
       if(temp->data == key)
          return 1;
       temp = temp->next;
   }
   //key not found
   return -1;
}
int main()
{
    struct node *head = NULL;
    addLast(&head,10);
    addLast(&head,20);
    addLast(&head,30);
    //change the key and try it yourself.
```

```
    if(searchNode(head,20) == 1)
        printf("Search Found\n");
    else
        printf("Search Not Found\n");
    return 0;
}
```

Deleting a node in linked list

Delete a given node from the linked list.

Example

Linked List : 10 20 30 NULL

Input

20

Output

10 30 NULL

Algorithm

1. If the head node has the given key, make the head node points to the second node and free its memory.

2. Otherwise,

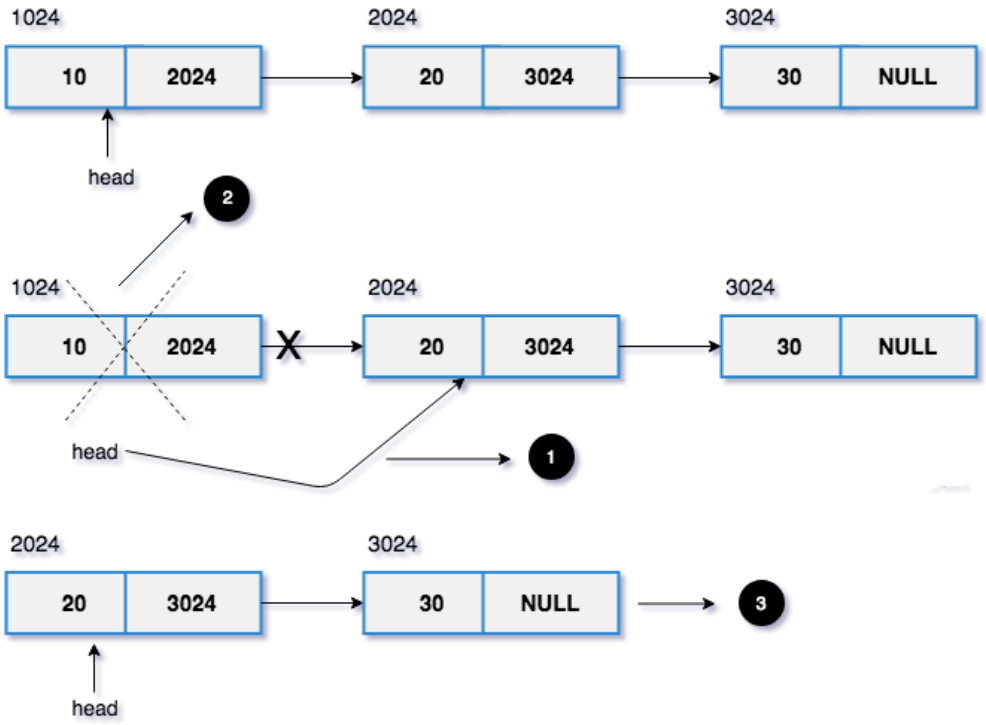
From the current node, check whether the next node has the given key

If yes, make the current->next = current->next->next and free the memory. Else, update the current node to the next and do the above process (from step 2) till the last node.

1. Head node has the given key

```
void deleteNode(struct node **head, int key)
{
    //temp is used to freeing the memory
    struct node *temp;
    //key found on the head node.
```

```
//move to head node to the next and free the head.
if(*head->data == key)
{
   temp = *head;    //backup the head to free its memory
   *head = (*head)->next;
   free(temp);
} }
```



1. Make the head points to the next node.

2. Free the head node's memory.

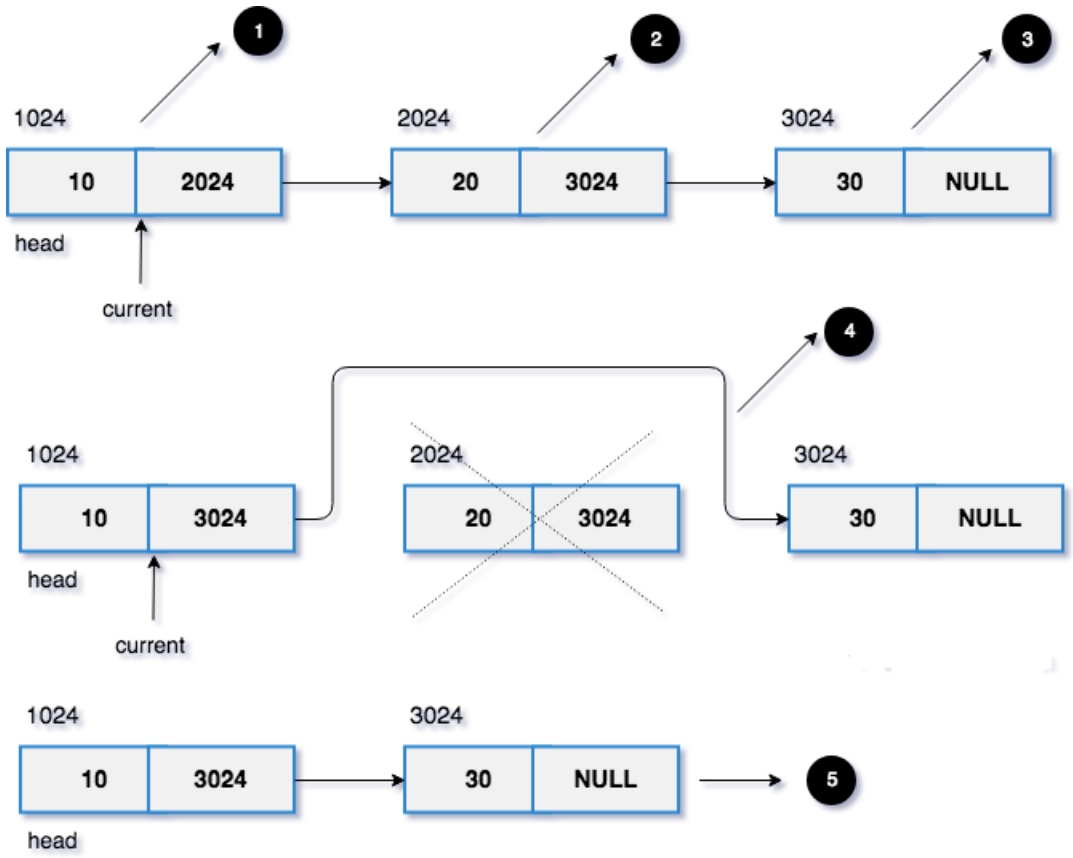
3. Finally, the new linked list.

2. For other nodes (Non-Head)

```
void deleteNode(struct node **head, int key)
{     //temp is used to freeing the memory
      struct node *temp;
      //key found on the head node.
      //move to head node to the next and free the head.
      if((*head)->data == key)
      {
         temp = *head;    //backup to free the memory
         *head = (*head)->next;
         free(temp);
      }
      else
      {
         struct node *current  = *head;
         while(current->next != NULL)
         {
            //if yes, we need to delete the current->next node
            if(current->next->data == key)
            {
               temp = current->next;
               //node will be disconnected from the linked list.
               current->next = current->next->next;
               free(temp);
               break;
            }
```

```
        //Otherwise, move the current node and proceed
        else
            current = current->next;
    }     }}
```

Let's delete data 20.



1. Make the current node points to the head node. (current => data = 10).

2. current => next. (current=>next=>data = 20).

3. current => next => next. (current=>next=>next=>data = 30).

4. We have to remove the node 20. Since current => next = 20 we can remove the node by setting current => next = current =>next => next. And then free the node.

5. Finally, the new linked list.

Implementation of deleting a node in linked list

Example

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
   int data;
   struct node *next;
};
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 newNode
   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;
    }}
void deleteNode(struct node **head, int key)
{
    //temp is used to freeing the memory
    struct node *temp;
    //key found on the head node.
    //move to head node to the next and free the head.
    if((*head)->data == key)
    {
        temp = *head;    //backup to free the memory
        *head = (*head)->next;
        free(temp);
    }
    else
    {
        struct node *current  = *head;
        while(current->next != NULL)
        {
            //if yes, we need to delete the current->next node
            if(current->next->data == key)
            {
                temp = current->next;
```

```
            //node will be disconnected from the linked list.
            current->next = current->next->next;
            free(temp);
            break;
        }
        //Otherwise, move the current node and proceed
        else
            current = current->next;
    }}}

void printList(struct node *head)
{
    struct node *temp = head;
    //iterate the entire linked list and print the data
    while(temp != NULL)
    {
        printf("%d ->", temp->data);
        temp = temp->next;
    }
    printf("NULL\n");
}
int main()
{    struct node *head = NULL;
     addLast(&head,10);
     addLast(&head,20);
```

```
    addLast(&head,30);
    printf("Linked List Elements:\n");
    printList(head);
    //delete first node
    deleteNode(&head,10);
    printf("Deleted 10. The New Linked List:\n");
    printList(head);
    //delete last node
    deleteNode(&head,30);
    printf("Deleted 30. The New Linked List:\n");
    printList(head);
    //delete 20
    deleteNode(&head,20);
    printf("Deleted 20. The New Linked List:\n");
    printList(head);
    return 0;
}
```

Advantage of Singly Linked list:-

1) Insertions and Deletions can be done easily.

2) It does not need movement of elements for insertion and deletion.

3) It space is not wasted as we can get space according to our requirements.

4) Its size is not fixed.

5) It can be extended or reduced according to requirements.

6) Elements may or may not be stored in consecutive memory available; even then we can store the data in computer.

7) It is less expensive.

Disadvantage of Singly Linked list:-

1) It requires more space as pointers are also stored with information.

2) Different amount of time is required to access each element.

3) If we have to go to a particular element then we have to go through all those elements that come before that element.

4) We cannot traverse it from last & only from the beginning.

5) It is not easy to sort the elements stored in the linear linked list.

Applications of Singly Linked list:-

- One of the applications of singly linked list is in applications like a photo viewer, for watching similar photos in a continuous manner in the form of a slide show.
- Strategy for file allocation schemes by Operating System. Singly Linked List can be used to keep track of free space in the secondary disk. All the free spaces can be linked together

1.4.2 DOUBLY LINKED LIST

A Doubly Linked List is a unique type of Data Structure where there are a chain of nodes, that are connected to one another using pointers, where any individual node has 3 components –

Data

Previous Pointer

Next Pointer

For any node, its previous pointer contains the address of the previous node and the next pointer contains the address of the next node in the chain of nodes.

Components in a Doubly Linked List Program in C

For writing the Doubly Linked List Program in C we need to know that Doubly Linked List generally has the following components –

- Node – A single unit in Doubly Linked List (DLL) contains – Data, previous and next pointers.
- Next Pointer – Contains the Address to the next node

- Previous Pointer – Contains Addresses to the previous node
- Data – Stores the data value
- Head – The first node in DLL

Some variations of DLL also have a tail node pointer, which signifies that this node is the end node in DLL.

The next pointer of the last node points to NULL and previous pointer of the first node points to NULL as well

Why doubly linked list?

For Doubly Linked list in Data Structure in C, unlike singly Linked List, which only traverses in one direction, Doubly Linked List can traverse both in forwards and backwards direction.

As for any given node, we have both previous and next node addresses information available.

Syntax for creating a node

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

<u>Insertion operation in doubly linked list</u>

Following insertion operation can be performed on a doubly linked list.

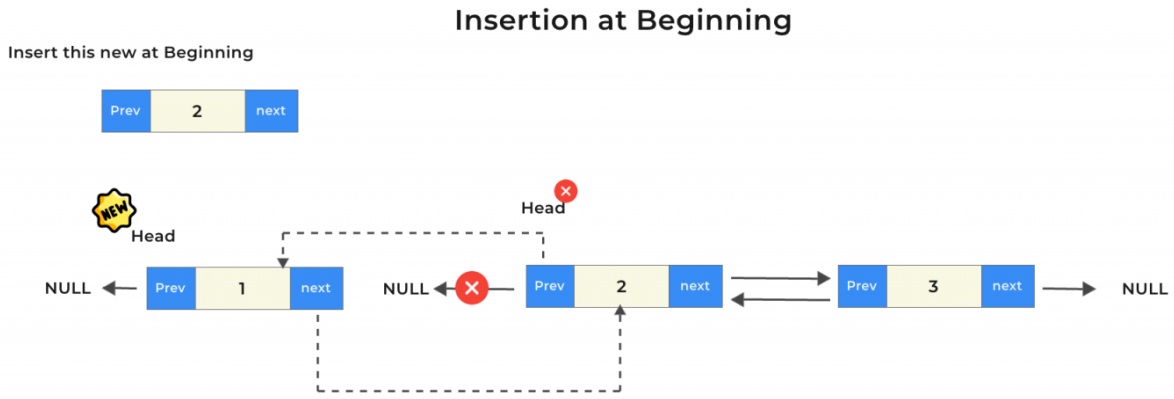
1. Insertion at beginning
2. Insertion at end

3. Insertion in between of nodes

Doubly Linked List Insertion at Beginning

Algorithm of insertion at the beginning

- Create a new node
- Assign its data value
- Assign newly created node's next ptr to current head reference. So, it points to the previous start node of the linked list address
- Assign newly created node's previous node to NULL
- Assign the current head's previous node to this new node
- Change the head reference to the new node's address.



Code (Insertion at Beginning)

```
#include<stdio.h>

#include<stdlib.h>


struct Node {

   int data;

   struct Node *next;

   struct Node *prev;

};
```

```
void insertStart(struct Node** head, int data){
    // creating memory for newNode
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    // assigning newNode's next as the current head
    // Assign data & and make newNode's prev as NULL
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;
    // if list already had item(s)
    // We need to make current head previous node as this new node
    if(*head != NULL)
        (*head)->prev = newNode;
    // change head to this newNode
    *head = newNode;
}
void display(struct Node* node)
{
    struct Node* end;
    printf("\nIn Forward Direction\n");
    while (node != NULL) {
        printf(" %d ", node->data);
        end = node;
        node = node->next;
    }
    printf("\nIn Backward direction \n");
```

```
    while (end != NULL) {
        printf(" %d ", end->data);
        end = end->prev;
    }
}
int main()
{
    struct Node* head = NULL;
    // Need '&' i.e. address as we need to change head
    insertStart(&head,1);
    insertStart(&head,2);
    insertStart(&head,3);
    // no need for '&' as head need not be changed
    // only doing traversal
    display(head);
    return 0;
}
```

Output

In Forward Direction

3 2 1

In Backward direction

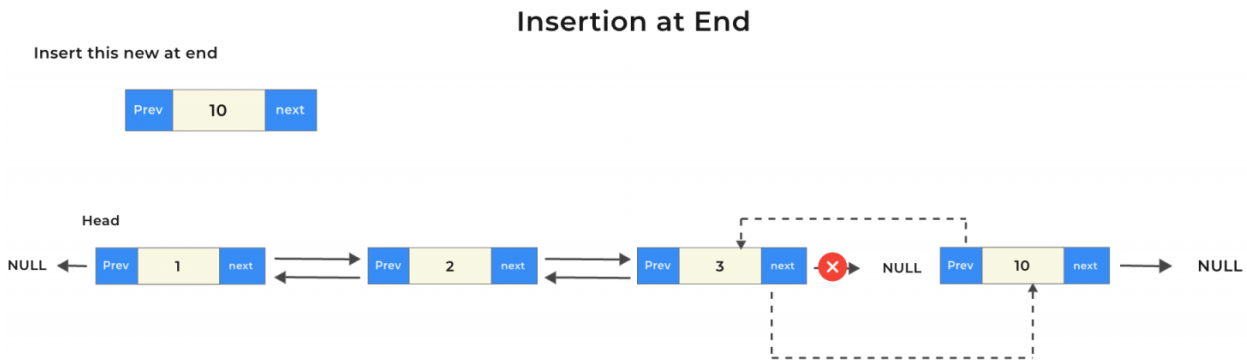
1 2 3

Doubly Linked List Insertion at the end

Algorithm of insertion at the beginning

- Create a new node

- Assign its data value
- Traverse till the end of the Linked List call this node **temp**
- Assign newly created node's next node to NULL
- Assign newly created node's previous node to temp
- Assign Temp's next node to this newly created node.



Code in C (Insertion at the End)

```
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node *next;
    struct Node *prev;
};
void insertStart(struct Node** head, int data){

    // creating memory for newNode
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    // assigning newNode's next as the current head
    // Assign data & and make newNode's prev as NULL
```

```
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;
    // if list already had item(s)
    // We need to make current head previous node as this new node
    if(*head != NULL)
        (*head)->prev = newNode;
    // change head to this newNode
    *head = newNode;
}
void insertLast(struct Node** head, int data){
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    //need this if there is no node present in linked list at all
    if(*head==NULL){
        *head = newNode;
        newNode->prev = NULL;
        return;
    }

    struct Node* temp = *head;
    // traverse till the last node
    while(temp->next!=NULL)
        temp = temp->next;
```

```
    // assign last node's next to this new Node
    temp->next = newNode;
    // assign this new Node's previous to last node(temp)
    newNode->prev = temp;
}
void display(struct Node* node)
{
    struct Node* end;
    printf("\nIn Forward Direction\n");
    while (node != NULL) {
        printf(" %d ", node->data);
        end = node;
        node = node->next;
    }
    printf("\nIn Backward direction \n");
    while (end != NULL) {
        printf(" %d ", end->data);
        end = end->prev;
    }
}
int main()
{
    struct Node* head = NULL;
    // Need '&' i.e. address as we need to change head
    insertStart(&head,1);
```

```
    insertStart(&head,2);
    insertStart(&head,3);
    insertLast(&head,10);
    insertLast(&head,20);
    // no need for '&' as head need not be changed
    // only doing traversal
    display(head);
    return 0;
}
```

Output

In Forward Direction

3 2 1 10 20

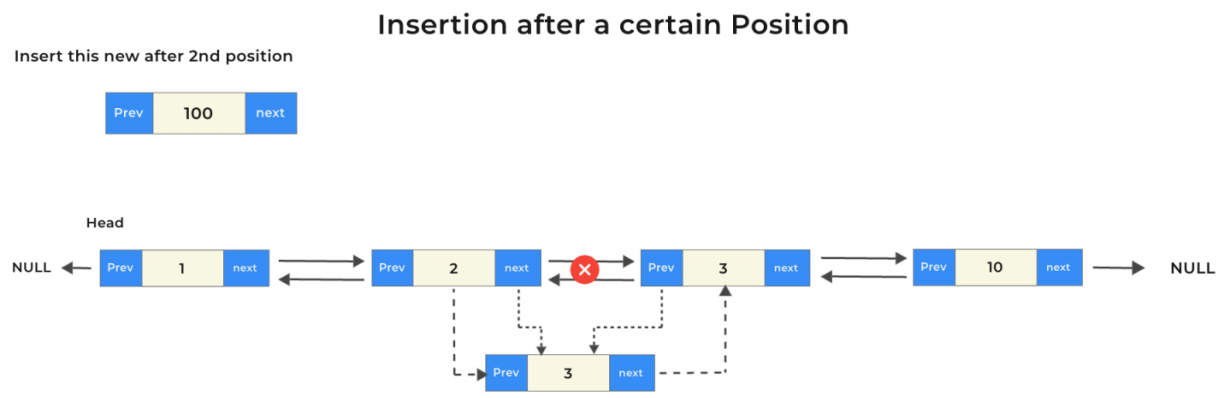
In Backward direction

20 10 1 2 3

Doubly Linked List Insertion after a position

Algorithm of insertion at the beginning

- Create a new node
- Assign its data value
- Traverse till nth(pos) node let's call this temp
- Assign newly created node's next node to temp's next node
- Assign newly created node's previous node to temp
- Assign Temp's next node to this newly created node.



Code in C (Insertion After a certain Position)

Run

```
#include<stdio.h>
#include<stdlib.h>
struct Node {
    int data;
    struct Node *next;
    struct Node *prev;
};
void insertStart(struct Node** head, int data){
    // creating memory for newNode
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    // assigning newNode's next as the current head
    // Assign data & and make newNode's prev as NULL
    newNode->data = data;
    newNode->next = *head;
    newNode->prev = NULL;
    // if list already had item(s)
```

```
    // We need to make current head previous node as this new node
    if(*head != NULL)
        (*head)->prev = newNode;
    // change head to this newNode
    *head = newNode;
}

void insertLast(struct Node** head, int data){
    struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));
    newNode->data = data;
    newNode->next = NULL;
    //need this if there is no node present in linked list at all
    if(*head==NULL){
        *head = newNode;
        newNode->prev = NULL;
        return;
    }
    struct Node* temp = *head;
    // traverse till the last node
    while(temp->next!=NULL)
        temp = temp->next;

    // assign last node's next to this new Node
    temp->next = newNode;
    // assign this new Node's previous to last node(temp)
```

```
    newNode->prev = temp;
}
int calcSize(struct Node* node){
    int size=0;
    while(node!=NULL){
        node = node->next;
        size++;
    }
    return size;
}
void insertPosition(int pos, int data, struct Node** head){
        int size = calcSize(*head);
    //If pos is 0 then should use insertStart method
    //If pos is less than 0 then can't enter at all
    //If pos is greater than size then bufferbound issue
    if(pos<1 || size < pos)
    {
        printf("Can't insert, %d is not a valid position\n",pos);
    }
    else{
        struct Node* temp = *head;
        struct Node* newNode = (struct Node*) malloc(sizeof(struct Node));

        newNode->data = data;
        newNode->next = NULL;
```

```
        // traverse till pos
        while(--pos)
        {
           temp=temp->next;
        }
        // assign prev/next of this new node
        newNode->next = temp->next;
        newNode->prev = temp;
        // change next of temp node
        temp->next = newNode;
    }
}
void display(struct Node* node)
{
    struct Node* end;
    printf("\nIn Forward Direction\n");
    while (node != NULL) {
        printf("%d  ", node->data);
        end = node;
        node = node->next;
    }

    printf("\n\nIn Backward direction \n");
    while (end != NULL) {
        printf("%d  ", end->data);
```

```
        end = end->prev;
    }
}
int main()
{
    struct Node* head = NULL;
    // Need '&' i.e. address as we need to change head
    insertStart(&head,1);
    insertStart(&head,2);
    insertStart(&head,3);
    insertLast(&head,10);
    insertLast(&head,20);

    insertPosition(2, 100, &head);
    // no need for '&' as head need not be changed
    // only doing traversal
    display(head);
    return 0;
}
```

Output

```
In Forward Direction
3  2  100  1  10  20

In Backward direction
20  10  1  2  3
```

Deletion in a Doubly Linked List

In a doubly linked list, we need to delete a node from the linked list. we just need to copy the head pointer to pointer ptr and shift the head pointer to its next.

So when we want to delete the node in the doubly linked list we have three ways to delete the node in another position.

- Deletion at beginning
- Deletion at middle
- Deletion at last

In Disadvantages, Doubly linked list occupy more space and often more operations are required for the similar tasks as compared to singly linked lists.

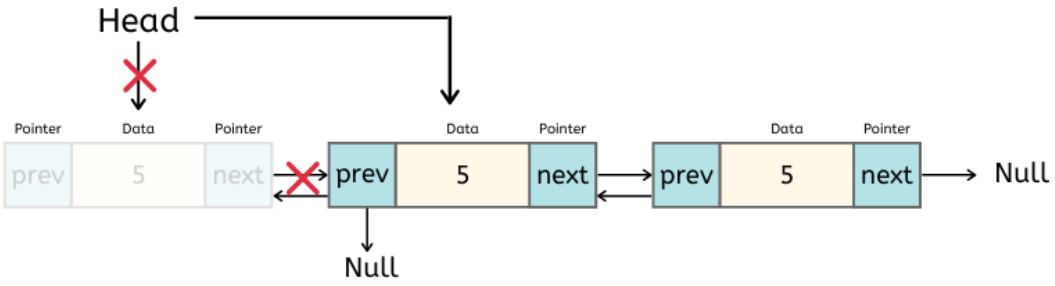
It is easy to reverse the linked list.

If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Deletion at Beginning

Algorithm

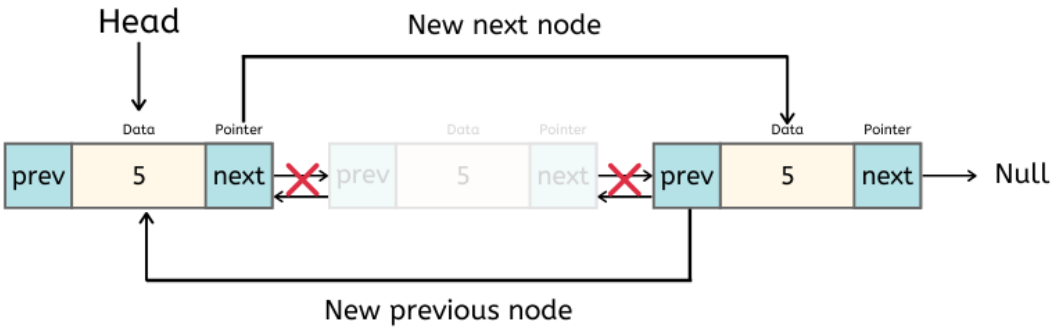
- Check if there is only 1 node or not
- If there is one node
    - Assign head to NULL
    - Free memory
- Else
    - Assign head to next node in the list
    - Assign head->prev to NULL
    - Free memory



Deletion at middle

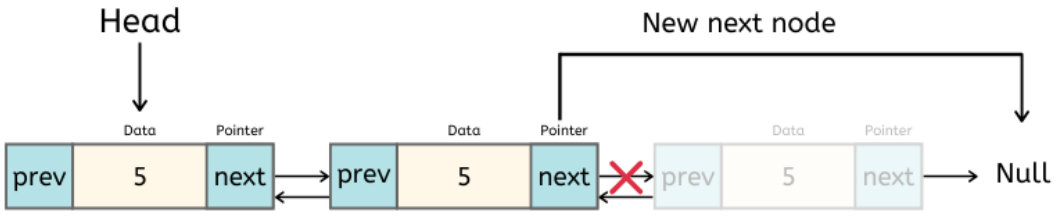
Algorithm

- Traverse till the target node
- create a node called the previous storing previous node of the target node
- Assign previous node's next pointer to the next node of the target node
- For the next node of the target node, its previous pointer is assigned to the targets node's previous node's address
- Free memory of target node



Deletion at last

- Traverse till the target node
- Check if this is the last node i.e. if node->next = NULL, then its last node
- Assign last node's previous node's next pointer to the last node's next node's address, which basically is NULL in this case
- Free the memory



Code for Deletion in a Doubly Linked List

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


struct Node{

  int data;

  struct Node *next;

  struct Node *prev;

};

int getLength(struct Node* node);


void insert(struct Node** head, int data){


    struct Node* freshNode = (struct Node*) malloc(sizeof(struct Node));


    freshNode->data = data;

    freshNode->next = *head;

    freshNode->prev = NULL;


    // If the linked list already had atleast 1 node
```

```
    if(*head !=NULL)
        (*head)->prev = freshNode;

    // freshNode will become head
    *head = freshNode;
}

void deleteFront(struct Node** head)
{
    struct Node* tempNode = *head;

    // if DLL is empty
    if(*head == NULL){
        printf("Linked List Empty, nothing to delete\n");
        return;
    }

    // if Linked List has only 1 node
    if(tempNode->next == NULL){
        printf("%d deleted\n", tempNode->data);
        *head = NULL;
        return;
    }

    // move head to next node
```

```
    *head = (*head)->next;
    // assign head node's previous to NULL
    (*head)->prev = NULL;

    printf("%d deleted\n", tempNode->data);
    free(tempNode);
}

void deleteEnd(struct Node** head){
    struct Node* tempNode = *head;

    // if DLL is empty
    if(*head == NULL){
        printf("Linked List Empty, nothing to delete\n");
        return;
    }

    // if Linked List has only 1 node
    if(tempNode->next == NULL){
        printf("%d deleted\n", tempNode->data);
        *head = NULL;
        return;
    }

    // else traverse to the last node
```

```
    while (tempNode->next != NULL)
        tempNode = tempNode->next;

    struct Node* secondLast = tempNode->prev;

    // Curr assign 2nd last node's next to Null
    secondLast->next = NULL;

    printf("%d deleted\n", tempNode->data);
    free(tempNode);
}

void deleteNthNode(struct Node** head, int n){
    //if the head node itself needs to be deleted
    int len = getLength(*head);

    // not valid
    if(n < 1 || n > len){
        printf("Enter valid position\n");
        return;
    }

    // delete the first node
    if(n == 1){
        deleteFront(head);
```

```
    return;
}

if(n == len){
    deleteEnd(head);
    return;
}

struct Node* tempNode = *head;

// traverse to the nth node
while (--n){
    tempNode = tempNode->next;
}

struct Node* previousNode = tempNode->prev; // (n-1)th node
struct Node* nextNode = tempNode->next; // (n+1)th node

// assigning (n-1)th node's next pointer to (n+1)th node
previousNode->next = tempNode->next;

// assigning (n+1)th node's previous pointer to (n-1)th node
nextNode->prev = tempNode->prev;

// deleting nth node
```

```
    printf("%d deleted \n", tempNode->data);
    free(tempNode);
}
// required for deleteNthNode
int getLength(struct Node* node){
    int len = 0;



    while(node!=NULL){
        node = node->next;
        len++;
    }

    return len;
}

//function to print the doubly linked list
void display(struct Node* node)
{
    struct Node *end = NULL;

    printf("List in Forward direction: ");
    while (node != NULL) {
        printf(" %d ", node->data);
        end = node;
```

```
        node = node->next;
    }

    printf("\nList in backward direction:");

    while (end != NULL) {
        printf(" %d ", end->data);
        end = end->prev;
    }
    printf("\n\n");
}

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

    insert(&head,7);
    insert(&head,8);
    insert(&head,9);
    insert(&head,10);
    insert(&head,11);
    insert(&head,12);

    display(head);
```

```
    deleteFront(&head);
    display(head);

    deleteEnd(&head);
    display(head);

    // delete 3rd node
    deleteNthNode(&head, 3);
    display(head);

    // delete 1st node
    deleteNthNode(&head, 1);
    display(head);

    // delete 1st node
    deleteEnd(&head);
    display(head);

    return 0;
}
```

Output

```
List in Forward direction:  12  11  10  9  8  7
List in backward direction: 7  8  9  10  11  12

12 deleted
```

List in Forward direction: 11 10 9 8 7

List in backward direction: 7 8 9 10 11

7 deleted

List in Forward direction: 11 10 9 8

List in backward direction: 8 9 10 11

9 deleted

List in Forward direction: 11 10 8

List in backward direction: 8 10 11

11 deleted

List in Forward direction: 10 8

List in backward direction: 8 10

8 deleted

List in Forward direction: 10

List in backward direction: 10

Advantages of using doubly linked list in C

- It saves time as we can traverse in both directions.
- It utilizes memory as we can construct and delete nodes according to our needs.
- Insertion and deletion of the node become efficient if the position is given.

Disadvantages of using doubly linked list in C

- Uses more memory per node.
- Insertion and deletion take more time because extra pointers need to be maintained

Applications

- It is used in the navigation systems where front and back navigation is required.
- It is used by the browser to implement backward and forward navigation of visited web pages that is a back and forward button.
- It is also used to represent a classic game deck of cards.

1.4.3 Circular Linked Lists

A Circular Linked List is a collection of elements connected together with the help of pointers. Each node contains a data value and addresses to the next node in the link where the last link of the circular Linked list has the address of the first node.

Why circular linked list?

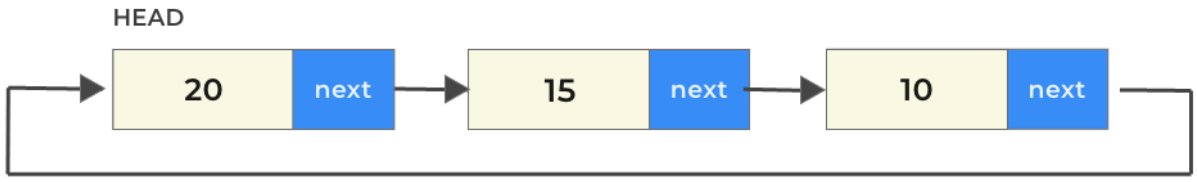
Circular Linked List is used when we continuously access the same items in a loop. Whenever we reach the last node we can restart operations again by directly reaching the first node from the last node itself.

Types:

The circular linked list is also two types.

- Singly linked list

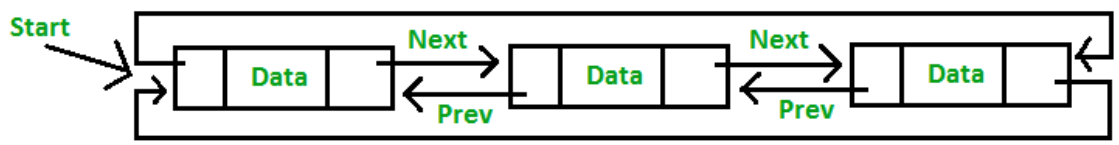
  In a singly circular linked list, the address of the last node's next pointer rather than being NULL is pointed towards the address of the head node.



- Doubly linked list

  Similarly, in a doubly linked list, in addition to the address of the last node's next pointer being the address of head node. The previous pointer of the head node is provided to the address of the last node.

O



In a circular linked list, we perform the following operations...

1. Insertion
2. Deletion
3. Display

Before we implement actual operations, first we need to setup empty list. First perform the following steps before implementing actual operations.

- Step 1 - Include all the **header files** which are used in the program.
- Step 2 - Declare all the **user defined** functions.
- Step 3 - Define a **Node** structure with two members **data** and **next**
- Step 4 - Define a Node pointer '**head**' and set it to **NULL**.
- Step 5 - Implement the **main** method by displaying operations menu and make suitable function calls in the main method to perform user selected operation.

Insertion

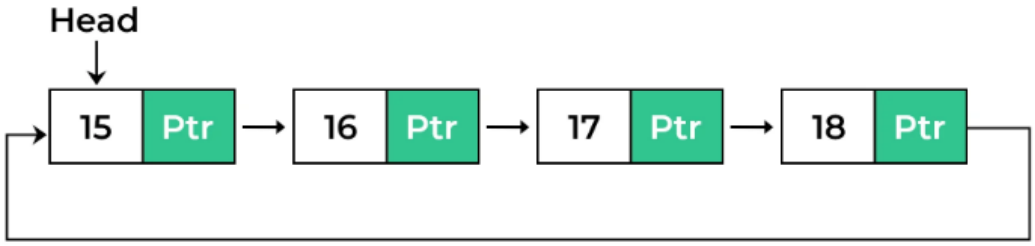
In a circular linked list, the insertion operation can be performed in three ways. They are as follows...

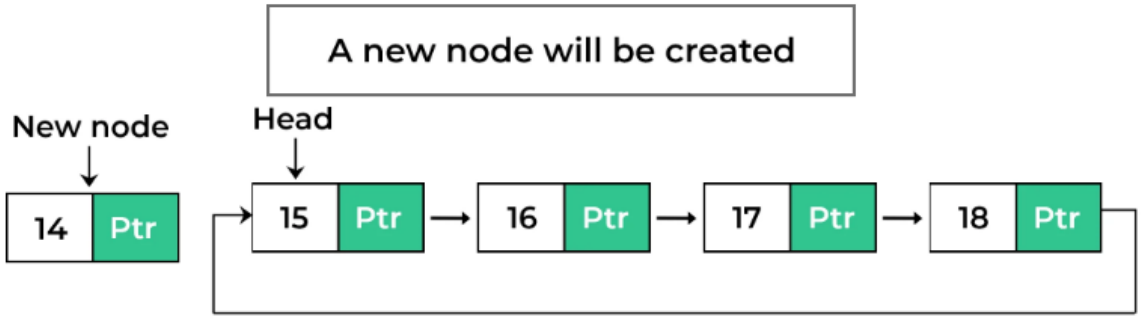
1. Inserting At Beginning of the list
2. Inserting At End of the list
3. Inserting At Specific location in the list

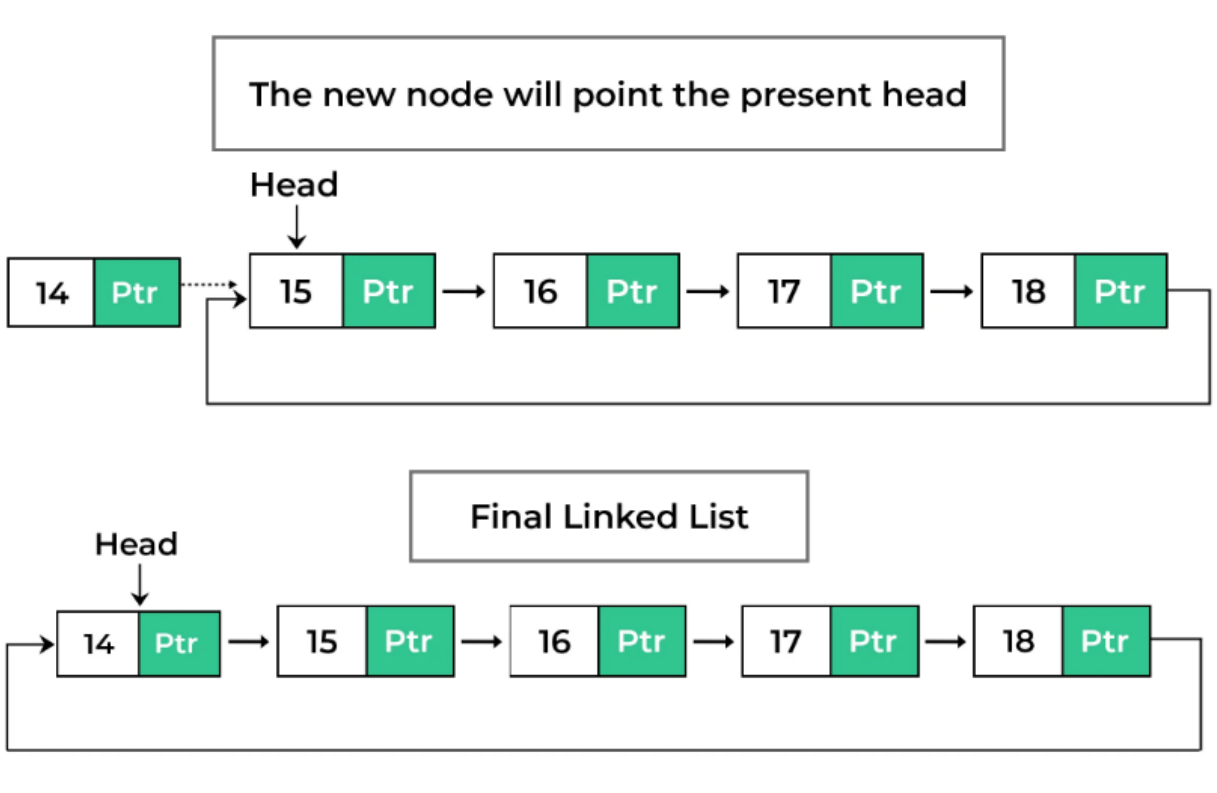
Inserting At Beginning of the list

We can use the following steps to insert a new node at beginning of the circular linked list...

- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode→next** = **head**.
- Step 4 - If it is **Not Empty** then, define a Node pointer '**temp**' and initialize with '**head**'.
- Step 5 - Keep moving the '**temp**' to its next node until it reaches to the last node (until '**temp → next** == **head**').
- Step 6 - Set '**newNode → next** =**head**', '**head** = **newNode**' and '**temp → next** = **head**'.





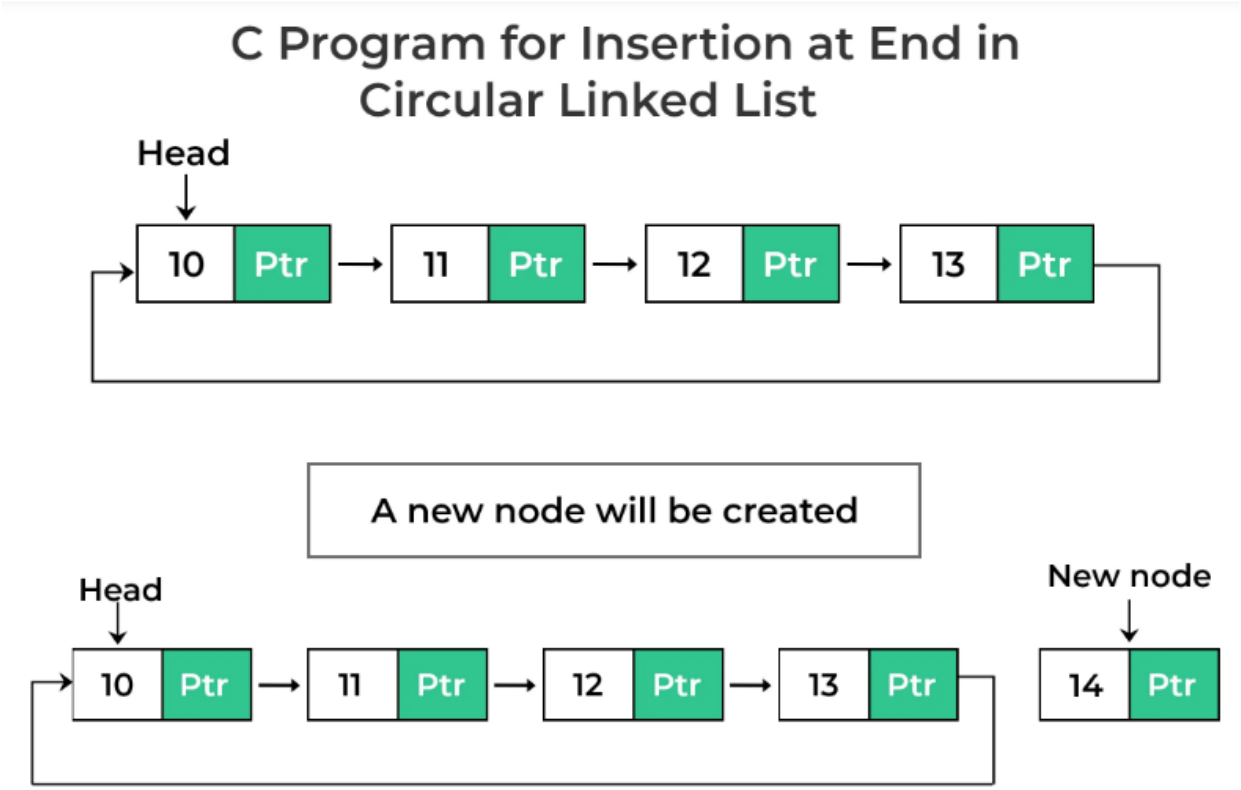


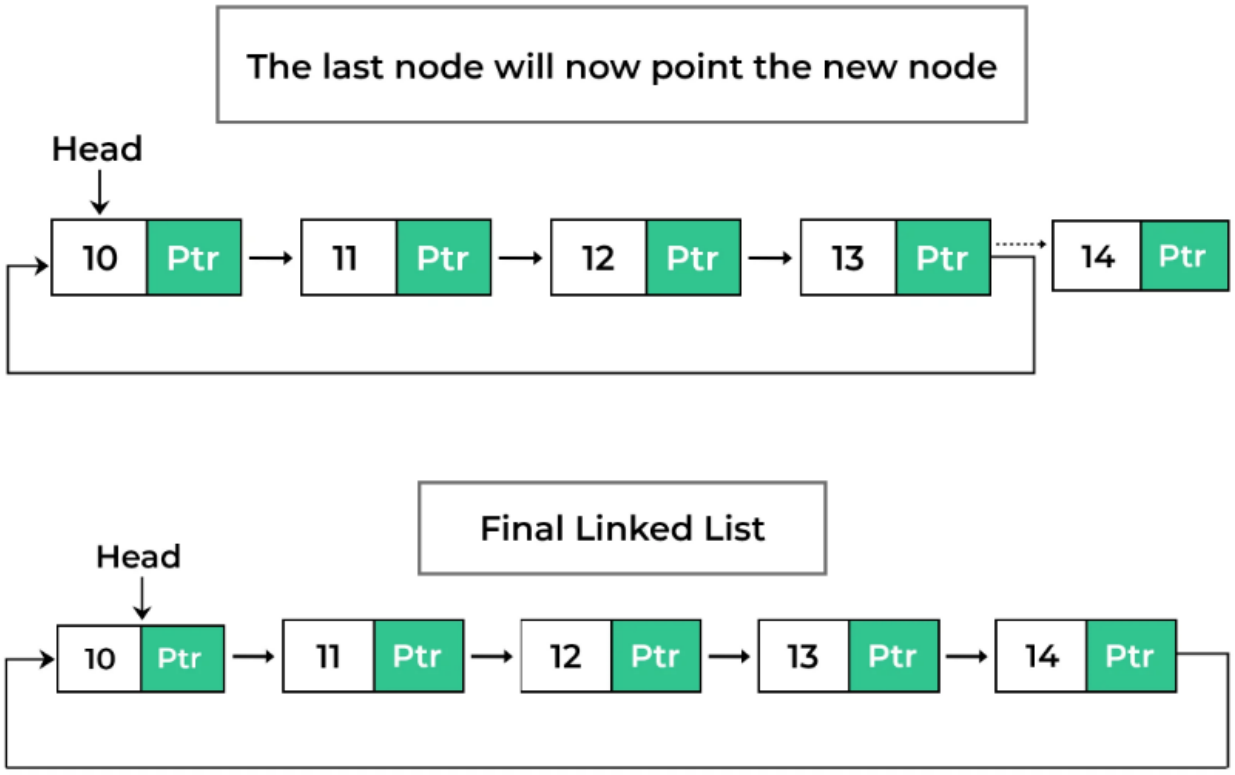
Inserting At End of the list

We can use the following steps to insert a new node at end of the circular linked list...

- Step 1 - Create a **newNode** with given value.

- Step 2 - Check whether list is **Empty** (**head** == **NULL**).
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the last node in the list (until **temp** → **next** == **head**).
- Step 6 - Set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.

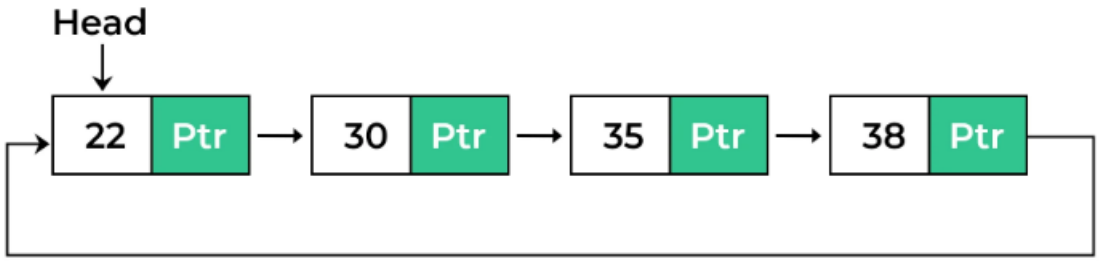




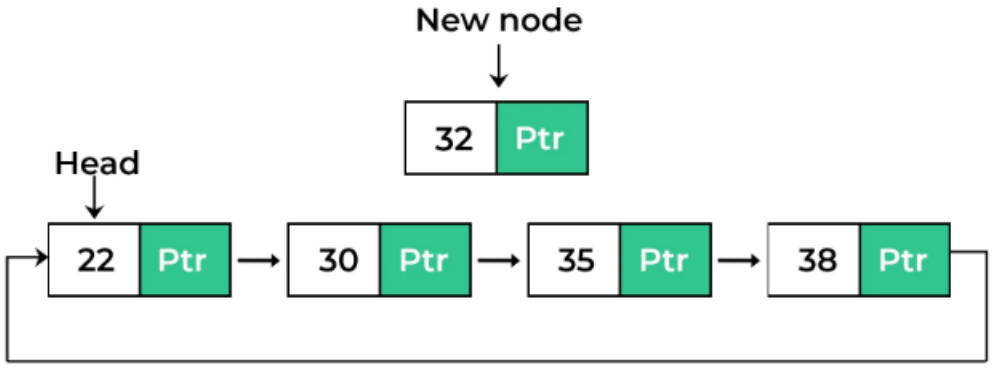
Inserting At Specific location in the list (After a Node)

We can use the following steps to insert a new node after a node in the circular linked list...

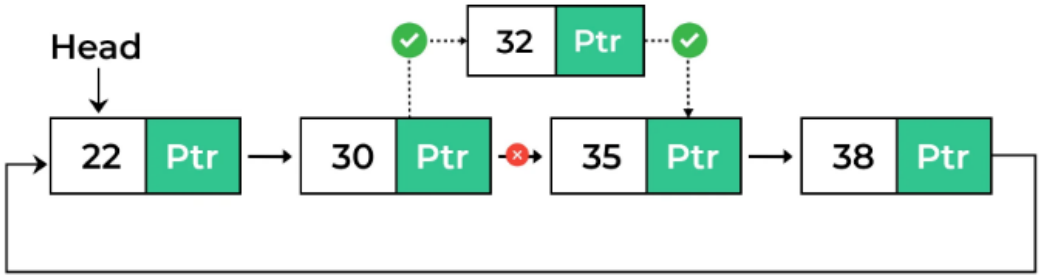
- Step 1 - Create a **newNode** with given value.
- Step 2 - Check whether list is **Empty** (**head** == **NULL**)
- Step 3 - If it is **Empty** then, set **head** = **newNode** and **newNode** → **next** = **head**.
- Step 4 - If it is **Not Empty** then, define a node pointer **temp** and initialize with **head**.
- Step 5 - Keep moving the **temp** to its next node until it reaches to the node after which we want to insert the newNode (until **temp1** → **data** is equal to **location**, here location is the node value after which we want to insert the newNode).
- Step 6 - Every time check whether **temp** is reached to the last node or not. If it is reached to last node then display **'Given node is not found in the list!!! Insertion not possible!!!'** and terminate the function. Otherwise move the **temp** to next node.
- Step 7 - If **temp** is reached to the exact node after which we want to insert the newNode then check whether it is last node (temp → next == head).
- Step 8 - If **temp** is last node then set **temp** → **next** = **newNode** and **newNode** → **next** = **head**.
- Step 8 - If **temp** is not last node then set **newNode** → **next** = **temp** → **next** and **temp** → **next** = **newNode**.



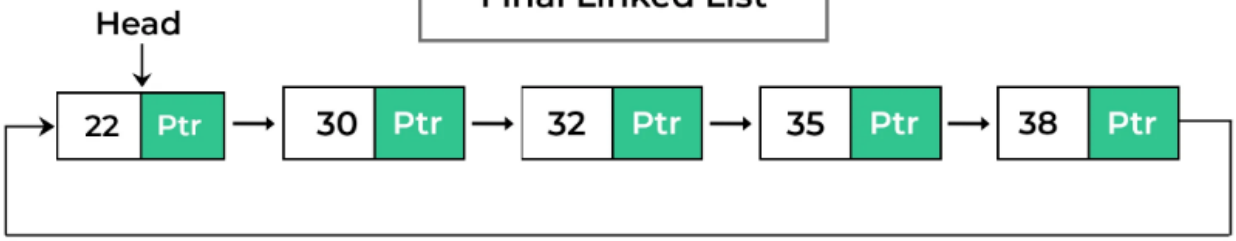
Let's suppose we have to insert the node at 3rd position



Now the links will be altered



Final Linked List



Deletion

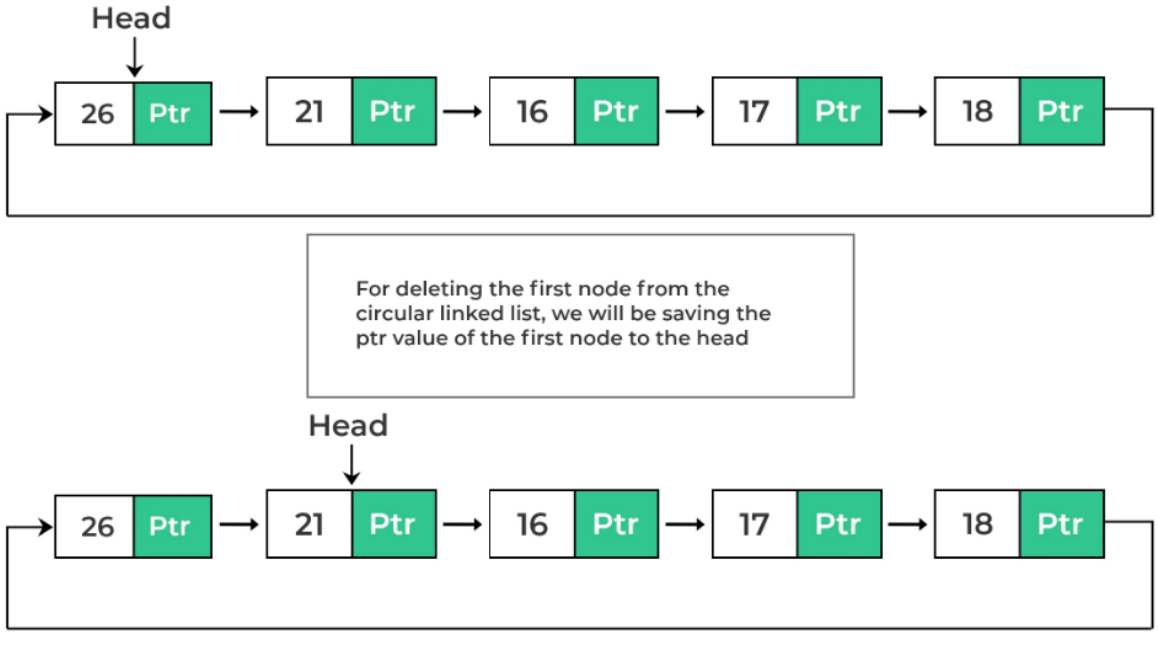
In a circular linked list, the deletion operation can be performed in three ways those are as follows.

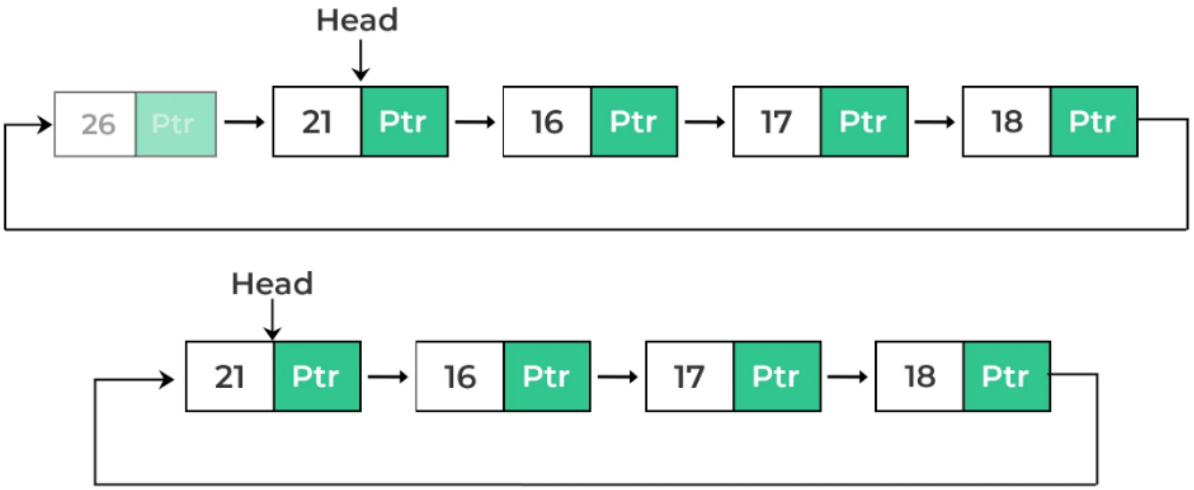
1. Deleting from Beginning of the list
2. Deleting from End of the list
3. Deleting a Specific Node

Deleting from Beginning of the list

We can use the following steps to delete a node from beginning of the circular linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- Step 3 - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize both '**temp1**' and '**temp2**' with **head**.
- Step 4 - Check whether list is having only one node (**temp1** → **next** == **head**)
- Step 5 - If it is **TRUE** then set **head** = **NULL** and delete **temp1** (Setting **Empty** list conditions)
- Step 6 - If it is **FALSE** move the **temp1** until it reaches to the last node. (until **temp1** → **next** == **head**)
- Step 7 - Then set **head** = **temp2** → **next**, **temp1** → **next** = **head** and delete **temp2**.

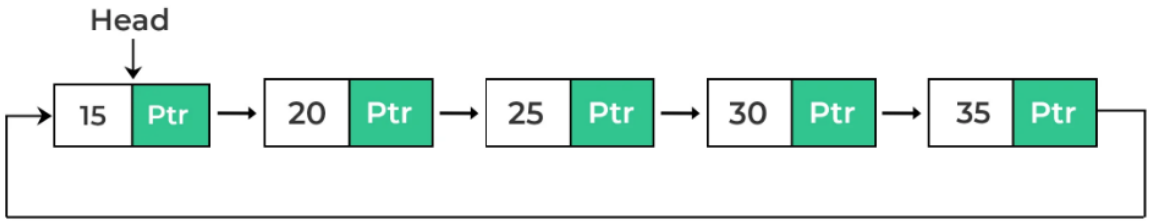




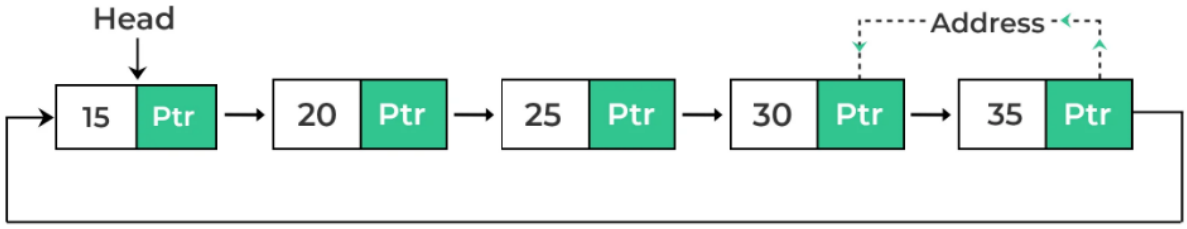
Deleting from End of the list

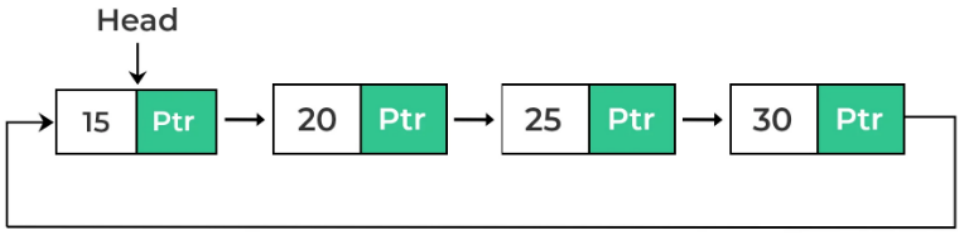
We can use the following steps to delete a node from end of the circular linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display '**List is Empty!!! Deletion is not possible**' and terminate the function.
- Step 3 - If it is **Not Empty** then, define two Node pointers '**temp1**' and '**temp2**' and initialize '**temp1**' with **head**.
- Step 4 - Check whether list has only one Node (**temp1** → **next** == **head**)
- Step 5 - If it is **TRUE**. Then, set **head** = **NULL** and delete **temp1**. And terminate from the function. (Setting **Empty** list condition)
- Step 6 - If it is **FALSE**. Then, set '**temp2 = temp1** ' and move **temp1** to its next node. Repeat the same until **temp1** reaches to the last node in the list. (until **temp1** → **next** == **head**)
- Step 7 - Set **temp2** → **next** = **head** and delete **temp1**.



For deleting the last node from the circular linked list, we will be saving the ptr value of the n^{th} node into $(n-1)^{th}$ node to the head



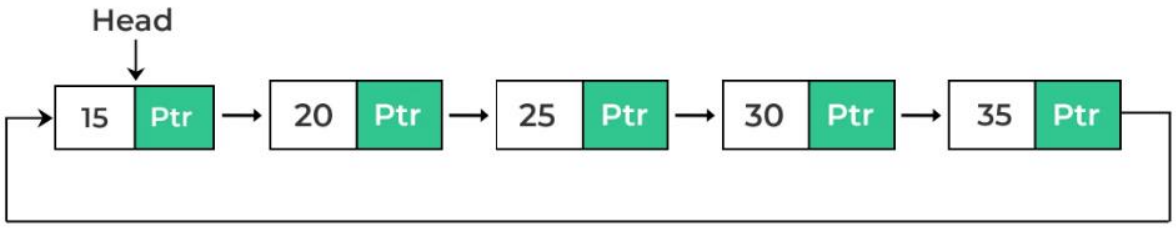


Deleting a Specific Node from the list

We can use the following steps to delete a specific node from the circular linked list...

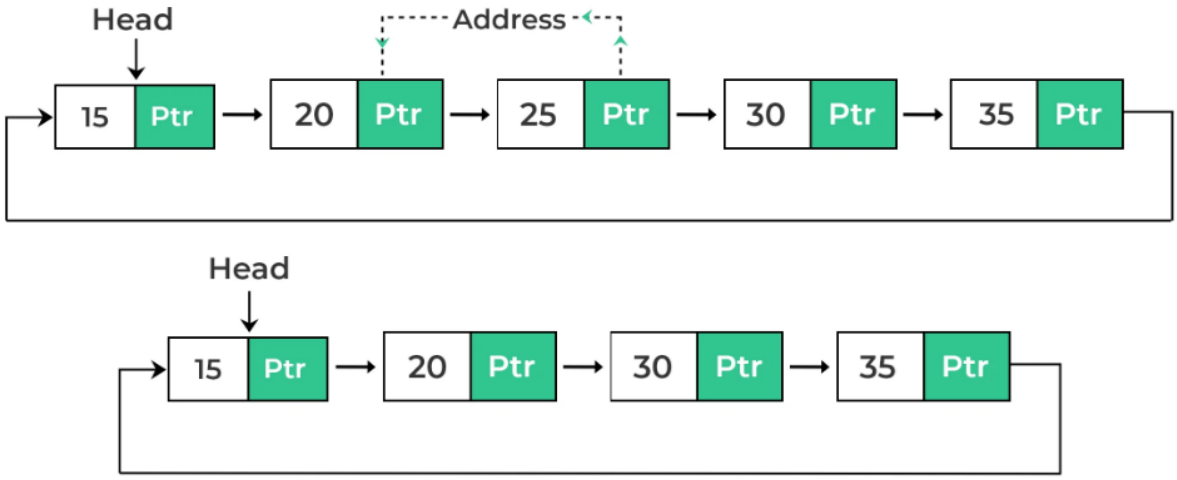
- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty** then, display **'List is Empty!!! Deletion is not possible'** and terminate the function.
- Step 3 - If it is **Not Empty** then, define two Node pointers **'temp1'** and '**temp2**' and initialize '**temp1**' with **head**.
- Step 4 - Keep moving the **temp1** until it reaches to the exact node to be deleted or to the last node. And every time set '**temp2 = temp1**' before moving the '**temp1**' to its next node.

- Step 5 - If it is reached to the last node then display **'Given node not found in the list! Deletion not possible!!!'**. And terminate the function.
- Step 6 - If it is reached to the exact node which we want to delete, then check whether list is having only one node (**temp1 → next == head**)
- Step 7 - If list has only one node and that is the node to be deleted then set **head** = **NULL** and delete **temp1** (**free(temp1)**).
- Step 8 - If list contains multiple nodes then check whether **temp1** is the first node in the list (**temp1 == head**).
- Step 9 - If **temp1** is the first node then set **temp2** = **head** and keep moving **temp2** to its next node until **temp2** reaches to the last node. Then set **head = head → next**, **temp2 → nex**t = **head** and delete **temp1**.
- Step 10 - If **temp1** is not first node then check whether it is last node in the list (**temp1 → next == head**).
- Step 1 1- If **temp1** is last node then set **temp2 → next** = **head** and delete **temp1** (**free(temp1)**).
- Step 12 - If **temp1** is not first node and not last node then set **temp2 → next** = **temp1 → next** and delete **temp1** (**free(temp1)**).



For deleting the node from a specific position
we will interchange the ptr values of n^{th} and $(n-1)^{th}$node

Here n=3



Displaying a circular Linked List

We can use the following steps to display the elements of a circular linked list...

- Step 1 - Check whether list is **Empty** (**head** == **NULL**)
- Step 2 - If it is **Empty**, then display **'List is Empty!!!'** and terminate the function.
- Step 3 - If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **head**.
- Step 4 - Keep displaying **temp → data** with an arrow (--->) until **temp** reaches to the last node
- Step 5 - Finally display **temp → data** with arrow pointing to **head → data**.

Implementation of Circular Linked List using C Programming

```
#include<stdio.h>
#include<conio.h>
void insertAtBeginning(int);
void insertAtEnd(int);
void insertAtAfter(int,int);
void deleteBeginning();
void deleteEnd();
void deleteSpecific(int);
void display();

struct Node
{
  int data;
```

```
  struct Node *next;
}*head = NULL;

void main()
{
  int choice1, choice2, value, location;
  clrscr();
  while(1)
  {
    printf("\n********** MENU **********\n");
    printf("1. Insert\n2. Delete\n3. Display\n4. Exit\nEnter your choice: ");
    scanf("%d",&choice1);
    switch()
    {
      case 1: printf("Enter the value to be inserted: ");
              scanf("%d",&value);
          while(1)
          {
              printf("\nSelect from the following Inserting options\n");
              printf("1. At Beginning\n2. At End\n3. After a Node\n4. Cancel\nEnter your c
hoice: ");
            scanf("%d",&choice2);
            switch(choice2)
            {
              case 1:     insertAtBeginning(value);
                          break;
              case 2:     insertAtEnd(value);
                          break;
              case 3:     printf("Enter the location after which you want to insert: ");
                          scanf("%d",&location);
                          insertAfter(value,location);
                          break;
              case 4:     goto EndSwitch;
```

```
                default: printf("\nPlease select correct Inserting option!!!\n");
              }
            }
        case 2: while(1)
            {
                    printf("\nSelect from the following Deleting options\n");
                    printf("1. At Beginning\n2. At End\n3. Specific Node\n4. Cancel\nEnter your
choice: ");
              scanf("%d",&choice2);
              switch(choice2)
              {
                case 1:         deleteBeginning();
                                break;
                case 2:         deleteEnd();
                                break;
                case 3:         printf("Enter the Node value to be deleted: ");
                                scanf("%d",&location);
                                deleteSpecic(location);
                                break;
                case 4:         goto EndSwitch;
                default: printf("\nPlease select correct Deleting option!!!\n");
              }
            }
            EndSwitch: break;
        case 3: display();
              break;
        case 4: exit(0);
        default: printf("\nPlease select correct option!!!");
     }
   }
}

void insertAtBeginning(int value)
```

```
{
   struct Node *newNode;
   newNode = (struct Node*)malloc(sizeof(struct Node));
   newNode -> data = value;
   if(head == NULL)
   {
     head = newNode;
     newNode -> next = head;
   }
   else
   {
     struct Node *temp = head;
     while(temp -> next != head)
       temp = temp -> next;
     newNode -> next = head;
     head = newNode;
     temp -> next = head;
   }
   printf("\nInsertion success!!!");
}
void insertAtEnd(int value)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    head = newNode;
    newNode -> next = head;
  }
  else
  {
    struct Node *temp = head;
```

```
    while(temp -> next != head)
      temp = temp -> next;
    temp -> next = newNode;
    newNode -> next = head;
  }
  printf("\nInsertion success!!!");
}
void insertAfter(int value, int location)
{
  struct Node *newNode;
  newNode = (struct Node*)malloc(sizeof(struct Node));
  newNode -> data = value;
  if(head == NULL)
  {
    head = newNode;
    newNode -> next = head;
  }
  else
  {
    struct Node *temp = head;
    while(temp -> data != location)
    {
      if(temp -> next == head)
      {
        printf("Given node is not found in the list!!!");
        goto EndFunction;
      }
      else
      {
        temp = temp -> next;
      }
    }
    newNode -> next = temp -> next;
```

```
    temp -> next = newNode;
    printf("\nInsertion success!!!");
  }
  EndFunction:
}
void deleteBeginning()
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp = head;
    if(temp -> next == head)
    {
      head = NULL;
      free(temp);
    }
    else{
      head = head -> next;
      free(temp);
    }
    printf("\nDeletion success!!!");
  }
}
void deleteEnd()
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp1 = head, temp2;
    if(temp1 -> next == head)
    {
```

```
      head = NULL;
      free(temp1);
    }
    else{
      while(temp1 -> next != head){
        temp2 = temp1;
        temp1 = temp1 -> next;
      }
      temp2 -> next = head;
      free(temp1);
    }
    printf("\nDeletion success!!!");
  }
}
void deleteSpecific(int delValue)
{
  if(head == NULL)
    printf("List is Empty!!! Deletion not possible!!!");
  else
  {
    struct Node *temp1 = head, temp2;
    while(temp1 -> data != delValue)
    {
      if(temp1 -> next == head)
      {
        printf("\nGiven node is not found in the list!!!");
        goto FuctionEnd;
      }
      else
      {
        temp2 = temp1;
        temp1 = temp1 -> next;
      }
```

```
    }
    if(temp1 -> next == head){
      head = NULL;
      free(temp1);
    }
    else{
      if(temp1 == head)
      {
        temp2 = head;
        while(temp2 -> next != head)
          temp2 = temp2 -> next;
        head = head -> next;
        temp2 -> next = head;
        free(temp1);
      }
      else
      {
        if(temp1 -> next == head)
        {
          temp2 -> next = head;
        }
        else
        {
          temp2 -> next = temp1 -> next;
        }
        free(temp1);
      }
    }
    printf("\nDeletion success!!!");
  }
  FuctionEnd:
}
void display()
```

```
{
  if(head == NULL)
    printf("\nList is Empty!!!");
  else
  {
    struct Node *temp = head;
    printf("\nList elements are: \n");
    while(temp -> next != head)
    {
      printf("%d ---> ",temp -> data);
    }
    printf("%d ---> %d", temp -> data, head -> data);
  }
}
```

Advantage of circular linked list

- Entire list can be traversed from any node of the list.
- It saves time when we have to go to the first node from the last node.
- Its is used for the implementation of queue.
- Reference to previous node can easily be found.
- When we want a list to be accessed in a circle or loops then circular linked list are used.

Disadvantage of circular linked list

- Circular list are complex as compared to singly linked lists.
- Reversing of circular list is a complex as compared to singly or doubly lists.
- If not traversed carefully, then we could end up in an infinite loop.
- Like singly and doubly lists circular linked lists also doesn't support direct accessing of elements.

Applications of Circular Linked List

A Circular Linked List can be used for the following –

- Circular lists are used in applications where the entire list is accessed one-by-one in a loop.
- It is also used by the Operating system to share time for different users, generally uses a Round-Robin time-sharing mechanism.
- Multiplayer games use a circular list to swap between players in a loop.
- Implementation of Advanced data structures like Fibonacci Heap
- The browser cache which allows you to hit the BACK button
- Undo functionality in Photoshop or Word
- Circular linked lists are used in Round Robin Scheduling
- Circular linked list used Most recent list (MRU LIST)

What is the benefit of a circularly linked list over singly linked list in search applications?

- If your program wants to access items of Linked List over and over again in a loop. Such processes do exist in OS, where some list of system processes must be implemented in loop one by one
- These are round-robin type of OS scheduling algorithms
- Any node can be the starting point since, we can access all nodes from anywhere, unlike a singly linked list where we must only start from the head node if we want to access all nodes.
- The fibonacci heap can be implemented via Circular Linked List

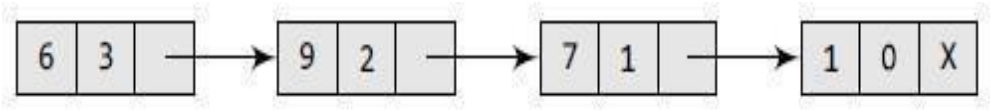
1.5 Polynomial Manipulation

Linked lists can be used to represent polynomials and the different operations that can be performed on them

Polynomial Representation

Consider a polynomial 6x3 + 9x2 + 7x + 1. Every individual term in a polynomial consists of two parts, a coefficient and a power. Here, 6, 9, 7, and 1 are the coefficients of the terms that have 3, 2, 1, and 0 as their powers respectively. Every term of a polynomial can be represented as a node of the linked list

Linked representation of a polynomial



Polynomial manipulations such as addition, subtraction & differentiation etc.. can be performed using linked list Declaration for Linked list implementation of Polynomial ADT

```
struct poly
{
int coeff;
int power;
struct poly * Next;
}
*list1,*list2,*list3;
```

Creation of the Polynomial

```
poly create(poly *head, poly *newnode)
{
poly*ptr;
 if(head==NULL)
{
head=newnode;
 return(head);
}
else
{
ptr=head;
```

```
while(ptr-> next!=NULL)
 ptr=ptr->next;
ptr->next=newnode;
}
return(head);
}
```

Addition of Polynomials:

To add two polynomials, we need to scan them once. If we find terms with the same exponent in the two polynomials, then we add the coefficients; otherwise, we copy the term of larger exponent into the sum and go on. When we reach at the end of one of the polynomial, then remaining part of the other is copied into the sum.

To add two polynomials, follow the following steps:

Read two polynomials.

Add them.

Display the resultant polynomial.

Addition of Polynomials:

```
void add()
{
poly *ptr1, *ptr2, *newnode;
 ptr1=list1;
ptr2=list2;
while(ptr1!=NULL && ptr2!= NULL)
{
newnode=malloc(sizeof(struct poly));
 if(ptr1->power==ptr2->power)
{
```

```
newnode->coeff = ptr1->coeff + ptr2->coeff;
 newnode->power=ptr1->power;
newnode->next=NULL;
list3=create(list3,newnode);
ptr1=ptr->next;
ptr2=ptr2->next;
}
else
{
if(ptr1->power > ptr2->power)
{
newnode->coeff = ptr1->coeff;
 newnode->power=ptr1->power;
newnode->next=NULL;
 list3=create(list3,newnode);
 ptr1=ptr1->next;
}
else
{
newnode->coeff = ptr2->coeff
newnode->power=ptr2->power;
newnode->next=NULL;
list3=create(list3,newnode);
 ptr2=ptr2->next;
}
```

}

}

FOR SUBTRACTION OF POLYNOMIALS,

Add this statement in the above program newnode->coeff = ptr1->coeff - ptr2->coeff

Multiplication of Polynomials:

You are given the head pointers two linked lists representing two polynomials, say head1 and head2. Perform the multiplication of the two polynomials and return the output as a linked list. The structure of the linked list is provided below.

Now, let's take a look at an example of the multiplication of two polynomials.
Let say we have,
P(x) = $x^2 - 5x + 9$ and
Q(x) = x3 − 10x2 + 9x + 1

We need to find the product **P(x)*Q(x)**.

P(x)*Q(x) = (**x^2 − 5x + 9**) * (**x3 − 10x2 + 9x + 1**)

Every term of **P(x)** will be multiplied to **Q(x)** in this way -
P(x)*Q(x) = **x^2** (**x3 − 10x2 + 9x + 1**) - **5x**(**x3 − 10x2 + 9x + 1**) + **9**(**x3−10x2+ 9x + 1**)
To multiply any two terms, we multiply their coefficients and add up the exponents of the variable.
So, we get -
P(x)*Q(x) =**x5 - 10x4 + 9x3 + x2 - 5x4+50x3 -45x2 - 5x + 9x3 - 90x2 + 81x + 9**
=x5 -15x4 +68x3 -134x2 +76x +9
In this example, we found the product using the pen-paper method. Now, let's learn how to do the same using a linked list.`
Representation of Polynomials as Linked List
This section will see how to represent polynomials using linked lists.
Each node in the **linked list** denotes one term of the polynomial.
Every node stores -

- Value of exponent
- Value of coefficient
- Pointer to the next node

So, the structure of the node looks like this -

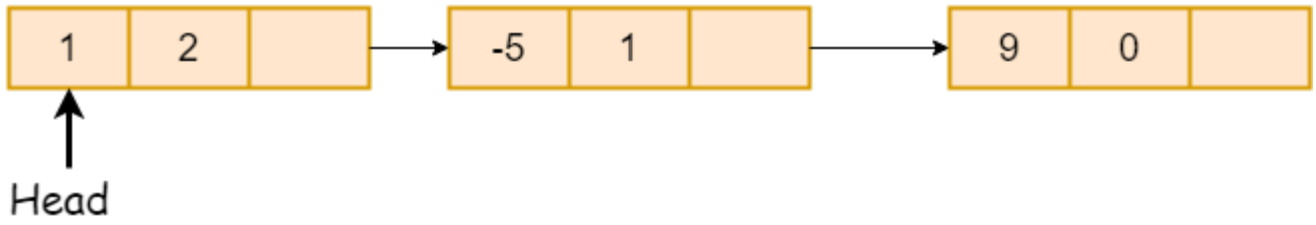
Coefficient	Exponent	Next Pointer

Structure of Node of a linked list to represent polynomials

Example - P(x) = $x^2 - 5x + 9$ is represented by a linked list containing 3 nodes as it has 3 terms which are as follows -

1. Exponent = 2 and Coefficient=1
2. Exponent = 1 and Coefficient = -5
3. Exponent = 0 and Coefficient=9

$P(x) = x^2 - 5x + 9$



Linked list to represent polynomials

Algorithm

From the previous sections, we know now how to represent a polynomial as a linked list.
So, now according to the problem statement, we have two pointers pointing to the head nodes of the given polynomials.
The algorithm is quite simple. We only need to iterate over the two linked lists, multiply the corresponding coefficients, and add the exponents. Let's see all the steps one by one below -

1. Define two pointers **ptr1** and **ptr2**, which point to **head1** and **head2**, respectively. These pointers will be used to iterate over the two lists.

2. Define a new node **head3** which points to the head node of the resulting product polynomial.

3. Multiply the terms pointed by **ptr1** and **ptr2**.

4. Declare two variables **coefficient** and **exponent** where coefficient = **ptr1->coefficient*ptr2->coefficient** and exponent = **ptr1->exponent** + **ptr2->exponent**.

5. Create a new node with the coefficient and exponent found above. And append it to the list pointed by **head3**.

6. Update **ptr2** to point to the next node in the second polynomial and repeat the above steps till **ptr2** becomes **NULL**.

7. Similarly, after each complete pass over the second polynomial, reset **ptr2** to point to **head2** and update **ptr1** to point to the next node in the first polynomial.