ICT-2101
Data Structure

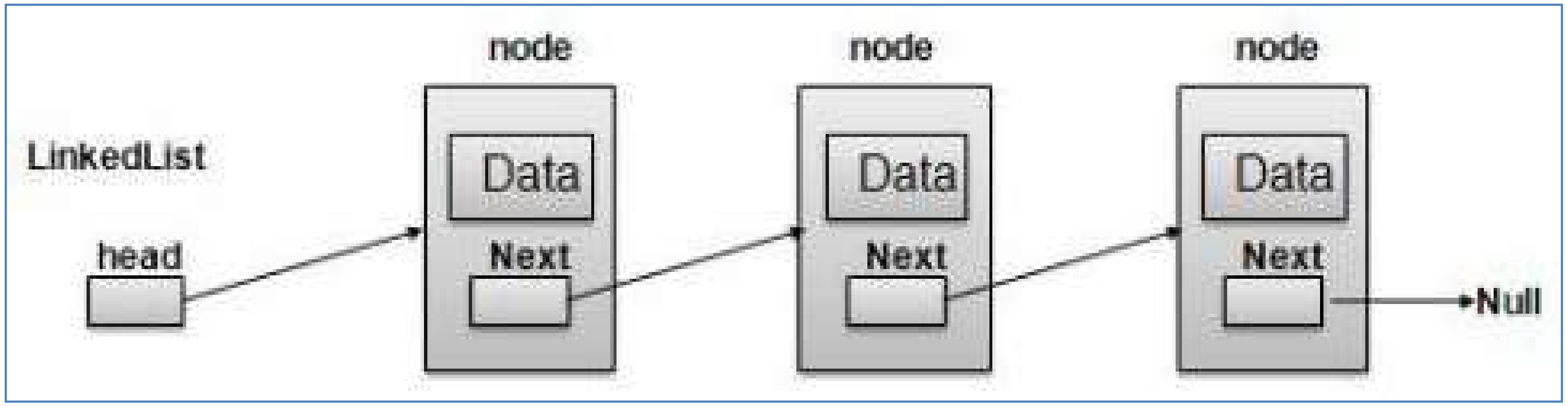
Lecture 05

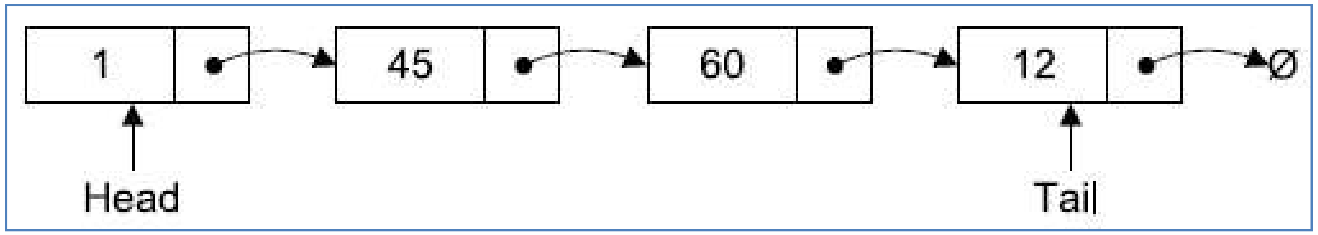
Linked List

Linked list

- A linked-list is a sequence of data structures which are connected together via links.
- Linked List is a sequence of links which contains items. Each link contains a connection to another link. Linked list the second most used data structure after array. Following are important terms to understand the concepts of Linked List.
  - **Link – Each Link of a linked list can store a data called an element.**
  - **Next – Each Link of a linked list contain a link to next link called Next.**
  - **LinkedList – A LinkedList contains the connection link to the first Link called First.**

Linked List Representation





- As per above shown illustration, following are the important points to be considered.
  - Linked List contains an link element called first.
  - Each Link carries a data field(s) and a Link Field called next.
  - Each Link is linked with its next link using its next link.
  - Last Link carries a Link as null to mark the end of the list.

Types of Linked List

Following are the various types of linked list.

- **Simple Linked List** – Item Navigation is forward only.

- **Doubly Linked List** – Items can be navigated forward and backward way.

- **Circular Linked List** – Last item contains link of the first element as next and first element has link to last element as prev.

Simple Linked List : Basic Operations

Following are the basic operations supported by a list:

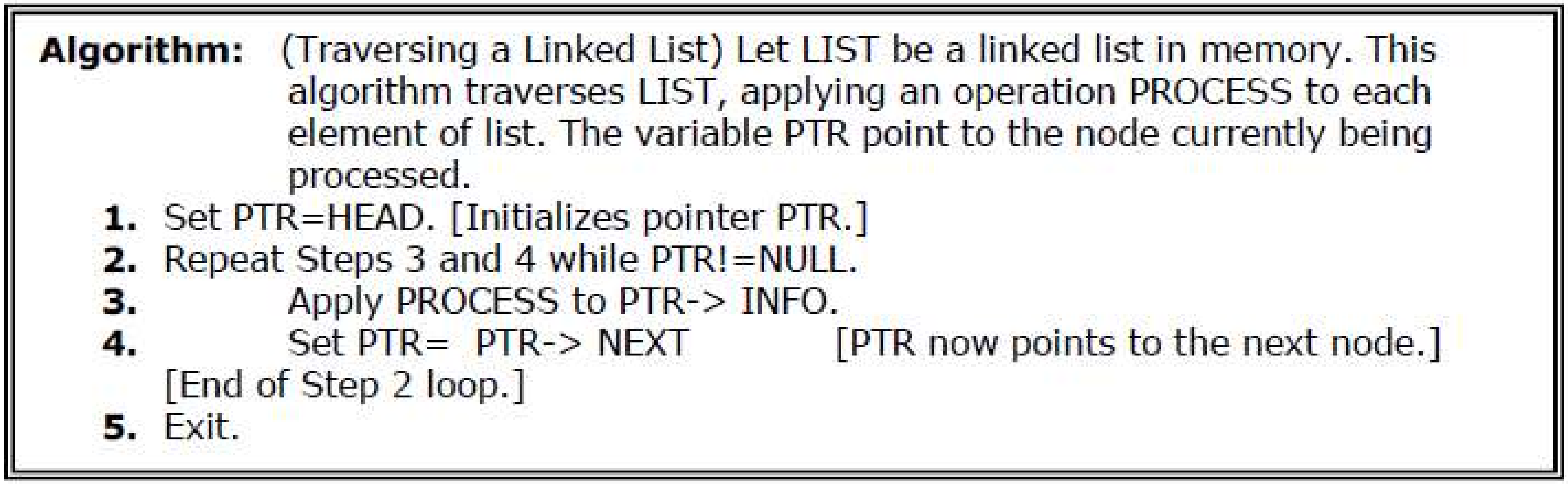
- **Insertion** – add an element at the beginning of the list.
- **Deletion** – delete an element at the beginning of the list.
- **Traverse** – Traversing complete list.
- **Search** – search an element using given key.
- **Delete** – delete an element using given key.

Simple Linked List : Basic Operations

- **Traversing a Linked List:**
- Suppose we want to traverse LIST in order to process each node exactly once.
- The traversing algorithm uses a pointer variable PTR which points to the node that is currently being processed. Accordingly, PTR->NEXT points to the next node to be processed

Simple Linked List : Basic Operations

- **Traversing a Linked List:**



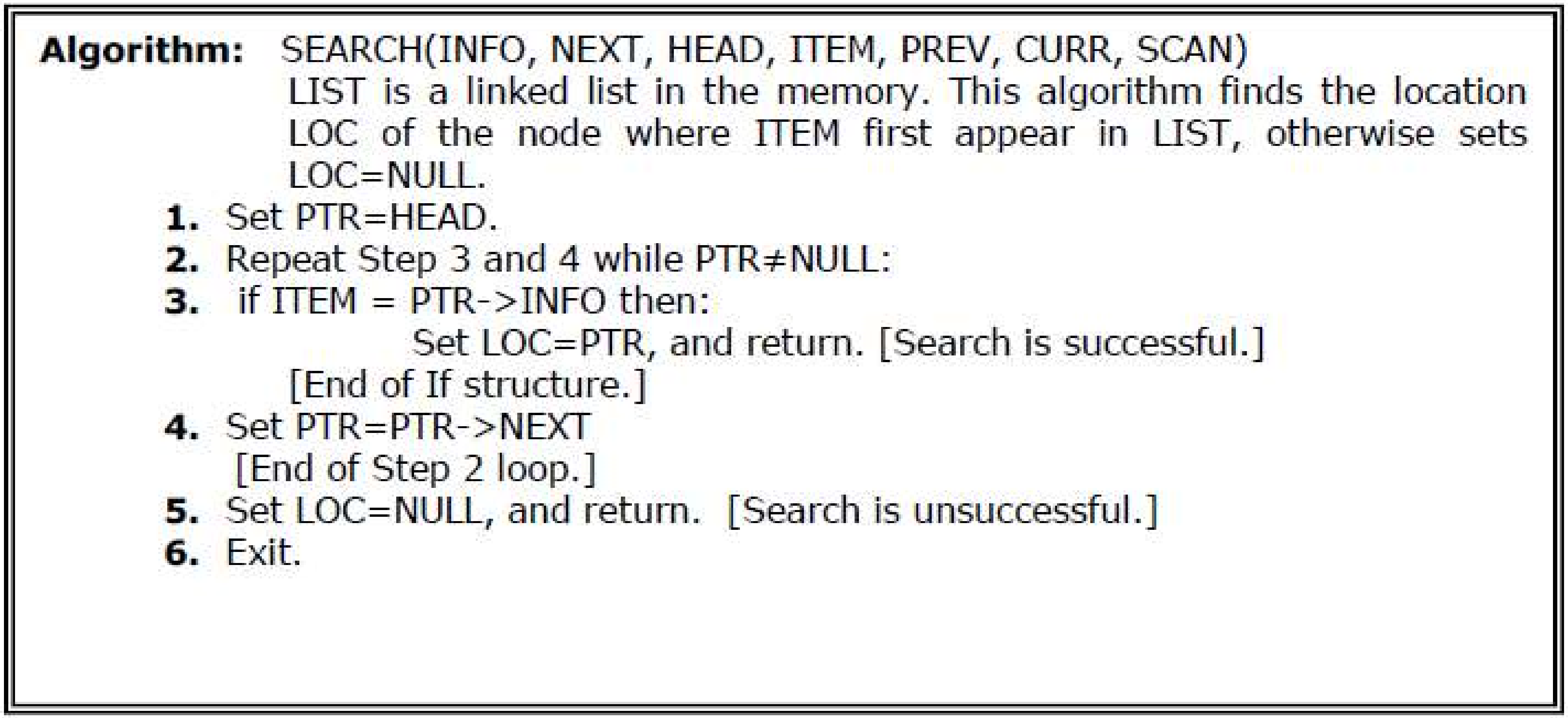
```
Algorithm:   (Traversing a Linked List) Let LIST be a linked list in memory. This
             algorithm traverses LIST, applying an operation PROCESS to each
             element of list. The variable PTR point to the node currently being
             processed.
  1.  Set PTR=HEAD. [Initializes pointer PTR.]
  2.  Repeat Steps 3 and 4 while PTR!=NULL.
  3.          Apply PROCESS to PTR-> INFO.
  4.          Set PTR=  PTR-> NEXT          [PTR now points to the next node.]
      [End of Step 2 loop.]
  5.  Exit.
```

Simple Linked List : Basic Operations

- **Searching a Linked List:**
- Let list be a linked list in the memory and a specific ITEM of information is given to search.
- Search for wanted ITEM in List can be performed by traversing the list using a pointer variable PTR and comparing ITEM with the contents PTR->INFO of each node, one by one of list.

Simple Linked List : Basic Operations

- **Searching a Linked List:**



```
Algorithm:  SEARCH(INFO, NEXT, HEAD, ITEM, PREV, CURR, SCAN)
            LIST is a linked list in the memory. This algorithm finds the location
            LOC of the node where ITEM first appear in LIST, otherwise sets
            LOC=NULL.
    1.  Set PTR=HEAD.
    2.  Repeat Step 3 and 4 while PTR≠NULL:
    3.   if ITEM = PTR->INFO then:
                  Set LOC=PTR, and return. [Search is successful.]
            [End of If structure.]
    4.  Set PTR=PTR->NEXT
         [End of Step 2 loop.]
    5.  Set LOC=NULL, and return.  [Search is unsuccessful.]
    6.  Exit.
```

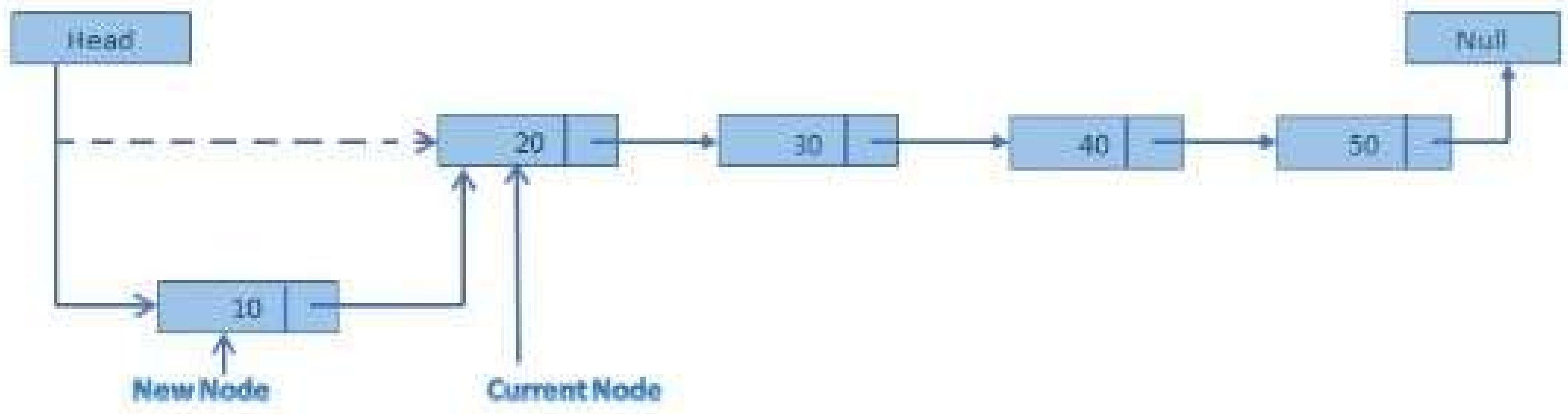
Simple Linked List : Basic Operations (Insertion)

- To add a new node, we must identify the logical predecessor (address of previous node) where the new node is to be inserting.

- There are three situation for inserting element in list.
  - Insertion at the front of list.
  - Insertion in the middle of the list.
  - Insertion at the end of the list.

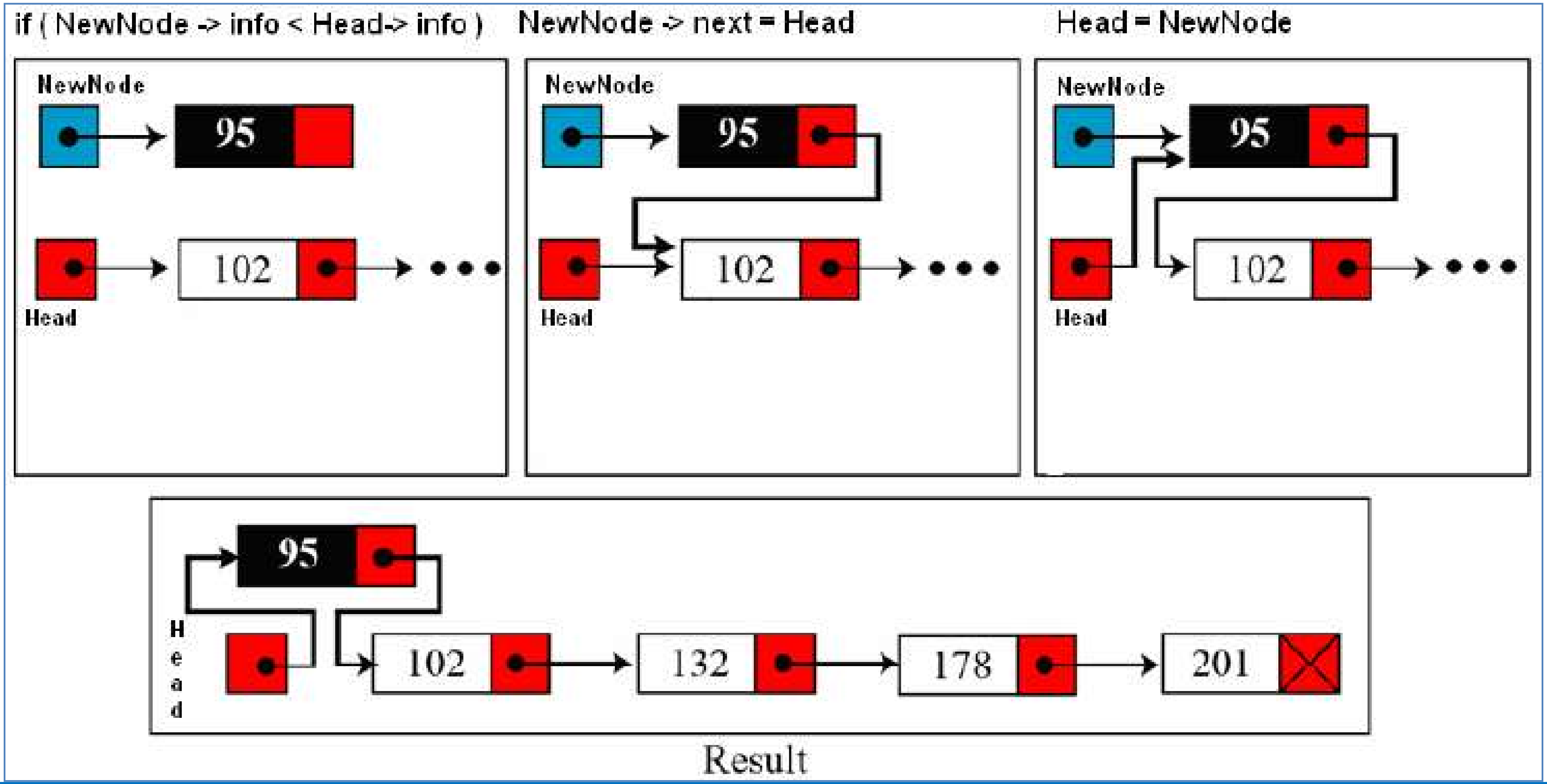
Simple Linked List : Basic Operations (Insertion)

Insertion at the front of list



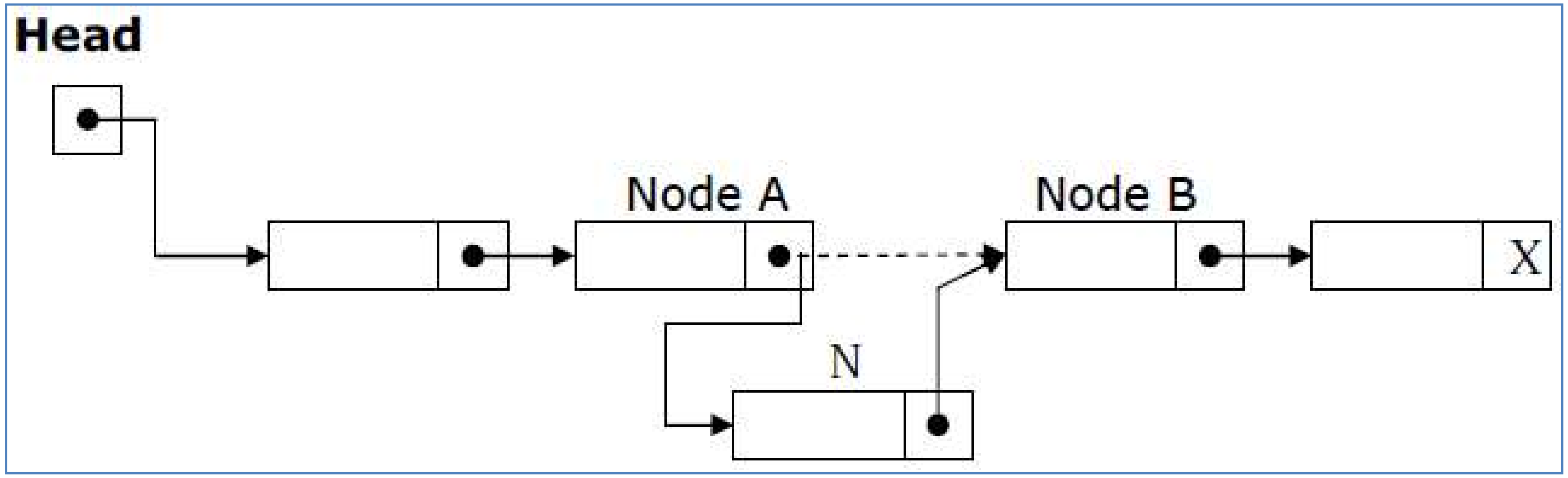
Inserting at the Beginning of a List:

- If the linked list is sorted list and new node has the least low value already stored in the list i.e. (***if New->info < Head->info)*** ***then new node is inserted at*** the beginning / Top of the list.



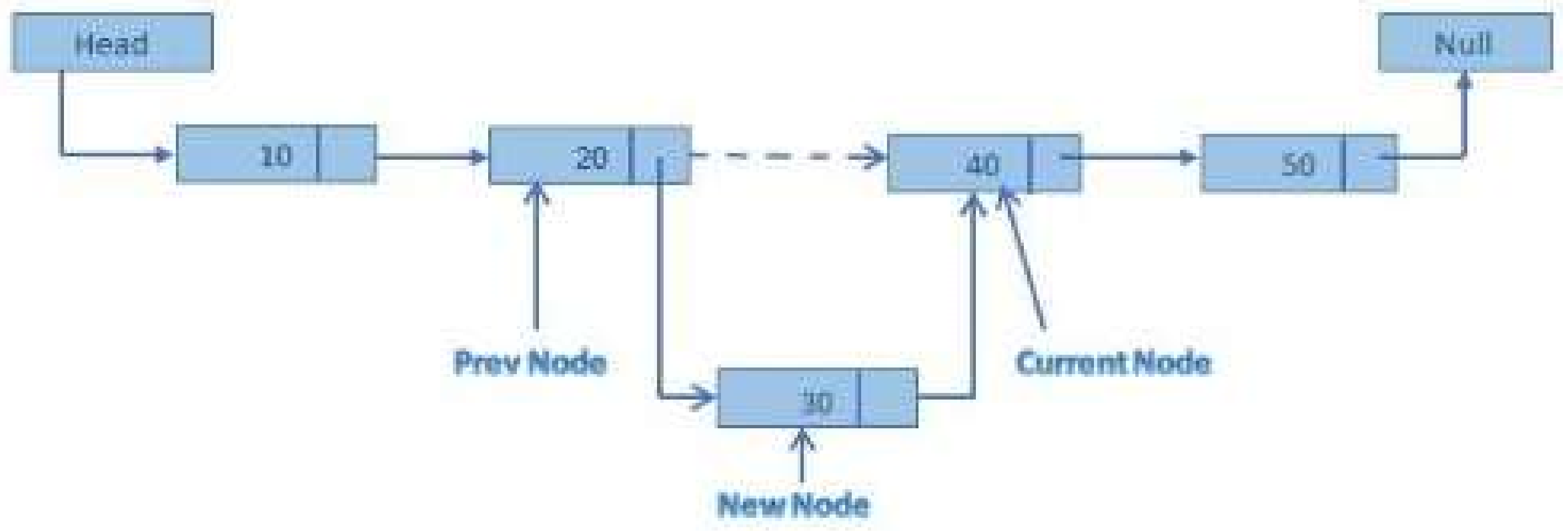
Simple Linked List : Basic Operations (Insertion)

- If a node N is to be inserted into the list between nodes **A and B in a linked list** named LIST. Its schematic diagram would be;



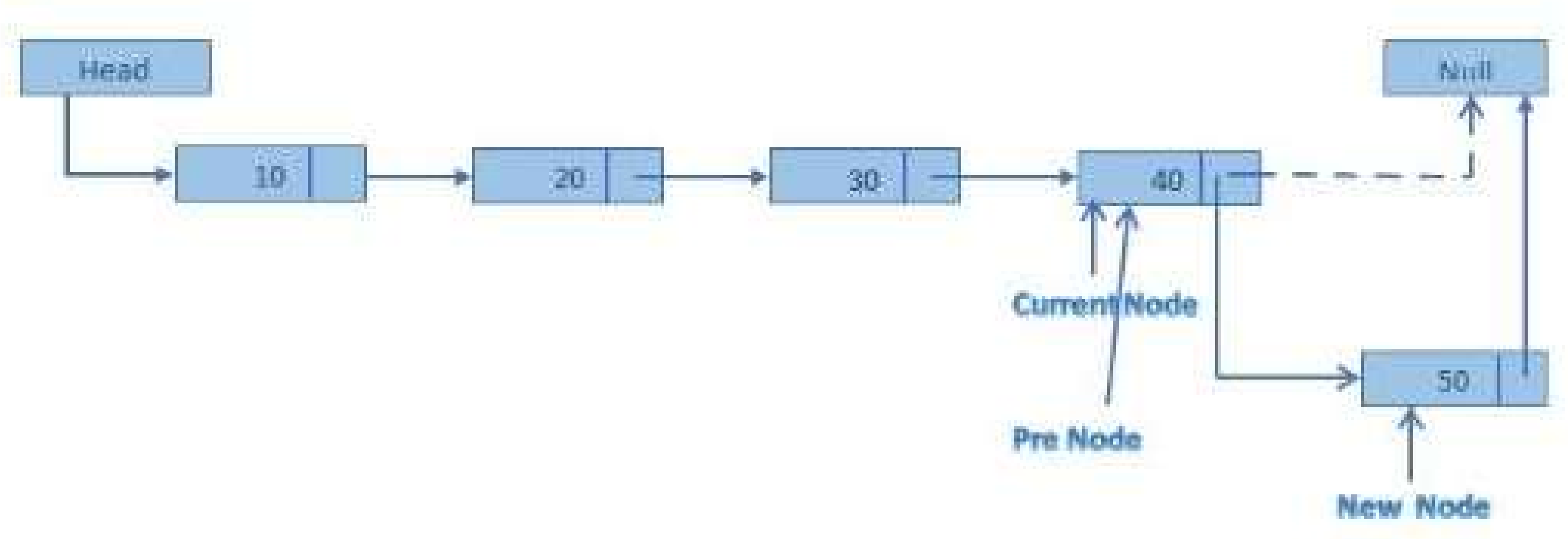
Simple Linked List : Basic Operations (Insertion)

Insertion Node in given location Linked List



Simple Linked List : Basic Operations (Insertion)

Insertion at the end of the list.



Simple Linked List : Basic Operations

- **Insertion is a three step process –**
  - Create a new Link with provided data.
  - Point New Link to old First Link.
  - Point First Link to this New Link.

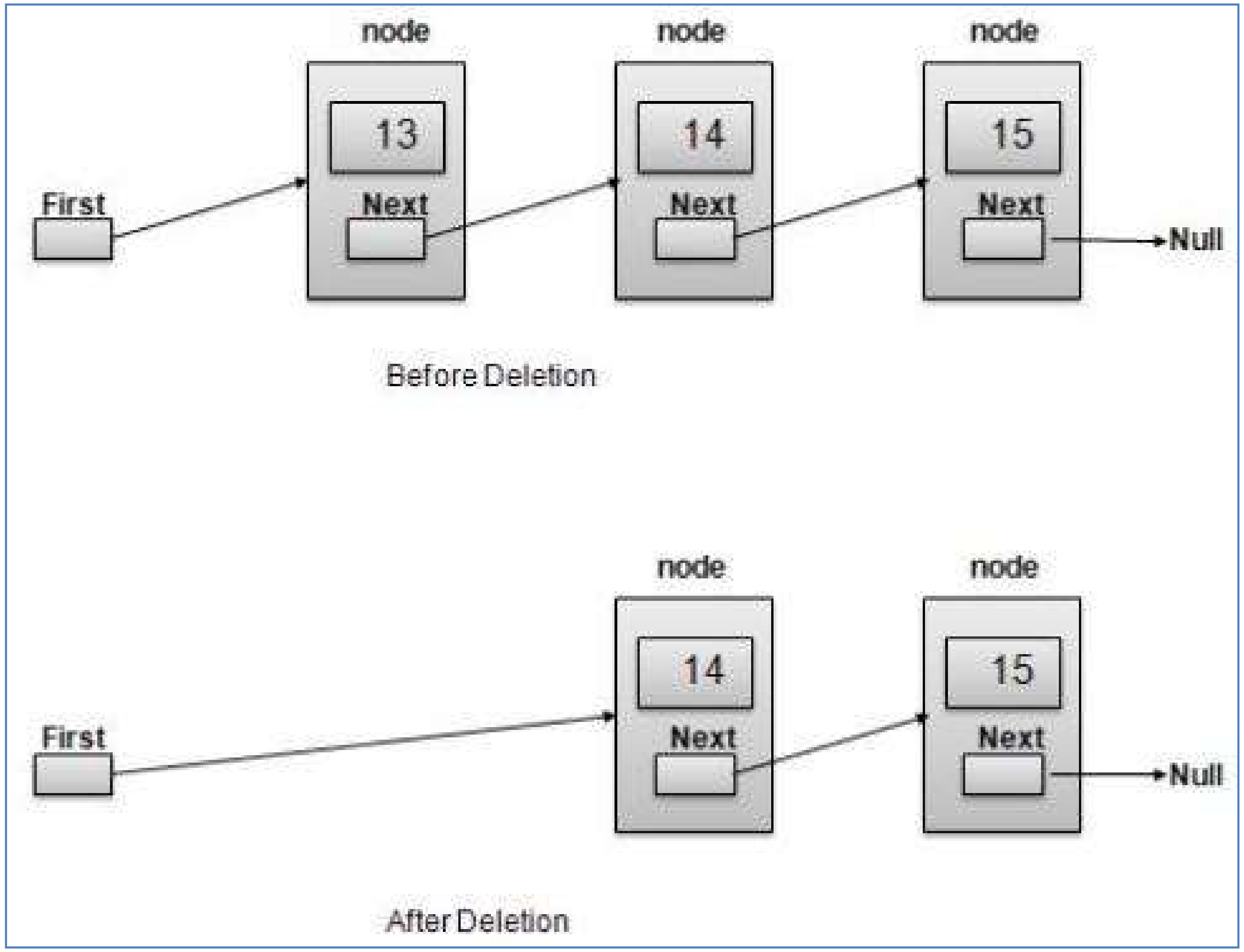
Simple Linked List : Basic Operations

```
Algorithm:   INSERT( ITEM)
             [This algorithm add newnodes at any position (Top, in Middle and at
             End) in the List ]
    1.  Create a NewNode node in memory
    2.  Set NewNode -> INFO =ITEM. [Copies new data into INFO of new node.]
    3.  Set NewNode -> NEXT = NULL. [Copies NULL in NEXT of new node.]
    4.  If HEAD=NULL, then HEAD=NewNode  and return. [Add first node in list]
    5.  if NewNode-> INFO < HEAD->INFO
        then  Set NewNode->NEXT=HEAD and HEAD=NewNode and return
        [Add node on top of existing list]

    6.  PrevNode = NULL, CurrNode=NULL;
    7.   for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
             { if(NewNode->INFO <= CurrNode ->INFO)
                 {
                       break the loop
                 }
              PrevNode = CurrNode;
            } [ end of loop ]
       [Insert after PREV node (in middle or at end) of the list]
    8.  Set NewNode->NEXT = PrevNode->NEXT  and
    9.  Set PrevNode->NEXT= NewNode.

    10.Exit.
```

Simple Linked List : Basic Operations

Deletion Operation: Deletion is a two step process –

- Get the Link pointed by First Link as Temp Link.
- Point First Link to Temp Link's Next Link.

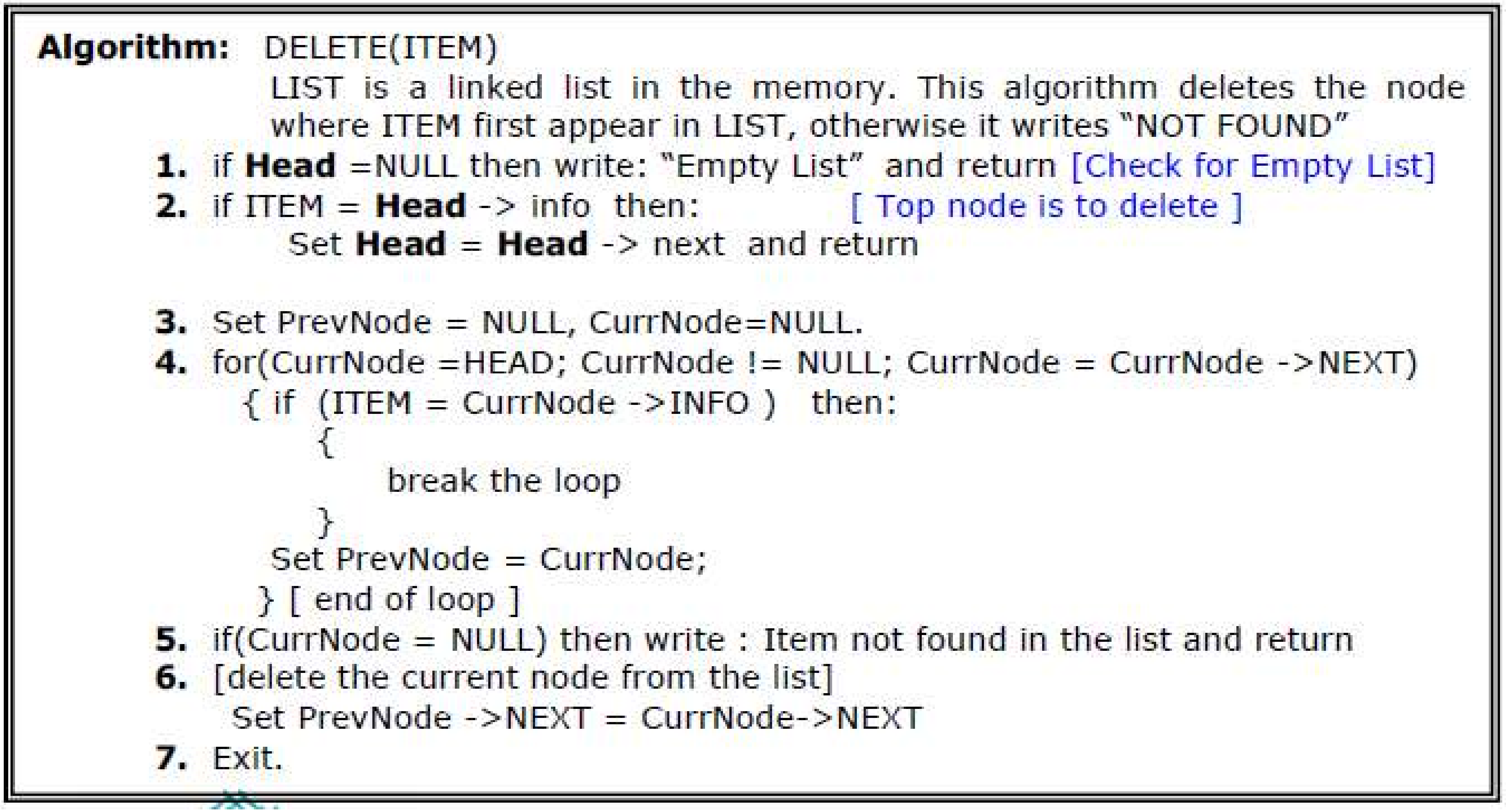


Simple Linked List : Basic Operations

- Deleting a node from a linked list is straightforward but there are a few cases we need to account for:
  - 1. the list is empty; or
  - 2. the node to remove is the only node in the linked list; or
  - 3. we are removing the head node; or
  - 4. we are removing the tail node; or
  - 5. the node to remove is somewhere in between the head and tail; or
  - 6. the item to remove doesn't exist in the linked list

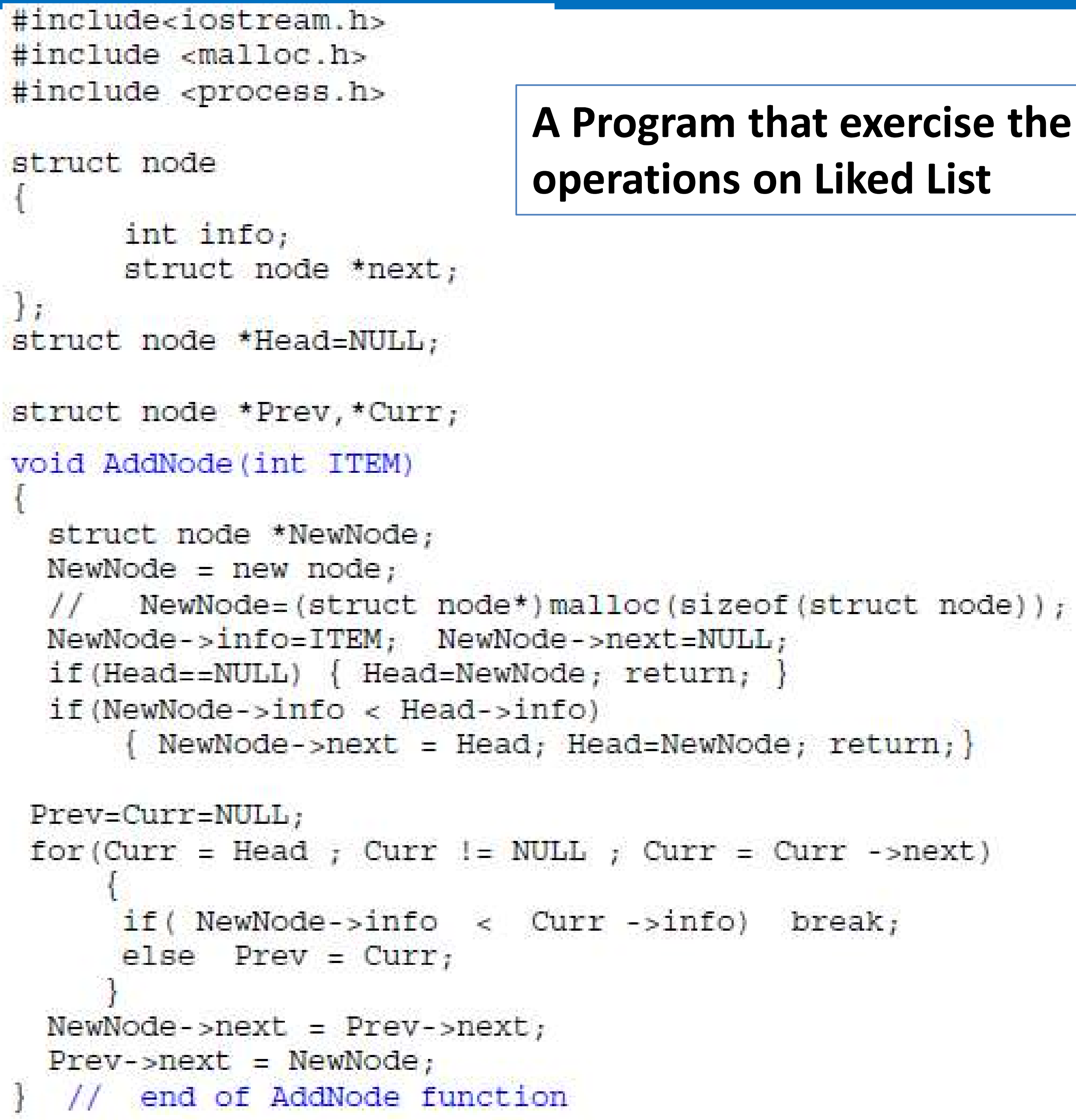
Simple Linked List : Basic Operations

- The following algorithm deletes a node from any position in the LIST.



```
Algorithm:   DELETE(ITEM)
             LIST is a linked list in the memory. This algorithm deletes the node
             where ITEM first appear in LIST, otherwise it writes "NOT FOUND"
    1.  if Head =NULL then write: "Empty List"  and return [Check for Empty List]
    2.  if ITEM = Head -> info  then:               [ Top node is to delete ]
              Set Head = Head -> next  and return

    3.  Set PrevNode = NULL, CurrNode=NULL.
    4.  for(CurrNode =HEAD; CurrNode != NULL; CurrNode = CurrNode ->NEXT)
          { if  (ITEM = CurrNode ->INFO )   then:
                {
                      break the loop
                }
             Set PrevNode = CurrNode;
           } [ end of loop ]
    5.  if(CurrNode = NULL) then write : Item not found in the list and return
    6.  [delete the current node from the list]
           Set PrevNode ->NEXT = CurrNode->NEXT
    7.  Exit.
```

A Program that exercise the operations on Liked List



```
#include<iostream.h>
#include <malloc.h>
#include <process.h>

struct node
{
      int info;
      struct node *next;
};
struct node *Head=NULL;

struct node *Prev,*Curr;
void AddNode(int ITEM)
{
  struct node *NewNode;
  NewNode = new node;
  //   NewNode=(struct node*)malloc(sizeof(struct node));
  NewNode->info=ITEM;  NewNode->next=NULL;
  if(Head==NULL) { Head=NewNode; return; }
  if(NewNode->info < Head->info)
      { NewNode->next = Head; Head=NewNode; return;}

 Prev=Curr=NULL;
 for(Curr = Head ; Curr != NULL ; Curr = Curr ->next)
     {
      if( NewNode->info  <  Curr ->info)  break;
      else  Prev = Curr;
     }
  NewNode->next = Prev->next;
  Prev->next = NewNode;
}  //  end of AddNode function
```

```
void DeleteNode()
{       int inf;
        if(Head==NULL){ cout<< "\n\n empty linked list\n"; return;}
        cout<< "\n Put the info to delete: ";
        cin>>inf;

        if(inf == Head->info)   // First / top node to delete
                { Head = Head->next;  return;}

        Prev = Curr = NULL;
        for(Curr = Head ; Curr != NULL ; Curr = Curr ->next )
             {
                if(Curr ->info == inf)  break;
                Prev = Curr;
             }
             if( Curr == NULL)
                   cout<<inf<< " not found in list \n";
                     else
                     { Prev->next = Curr->next; }

 }//  end of DeleteNode function
```

```
void Traverse()
{
        for(Curr = Head; Curr != NULL ; Curr = Curr ->next )

                cout<< Curr ->info<<"\t";
} //  end of Traverse function

int main()
{ int inf, ch;
  while(1)
  { cout<< " \n\n\n\n Linked List Operations\n\n";
    cout<< "  1- Add Node  \n  2- Delete Node \n";
    cout<< "  3- Traverse List \n  4- exit\n";
    cout<< "\n\n  Your Choice: "; cin>>ch;
      switch(ch)
      { case 1: cout<< "\n  Put info/value to Add: ";
                        cin>>inf);
                        AddNode(inf);
                        break;
        case 2:  DeleteNode();  break;
        case 3:  cout<< "\n Linked List Values:\n";
                 Traverse(); break;
        case 4: exit(0);
      }  // end of switch
  } //  end of while loop
 return 0;
} // end of main ( ) function
```

Insertion in Circular Linked List

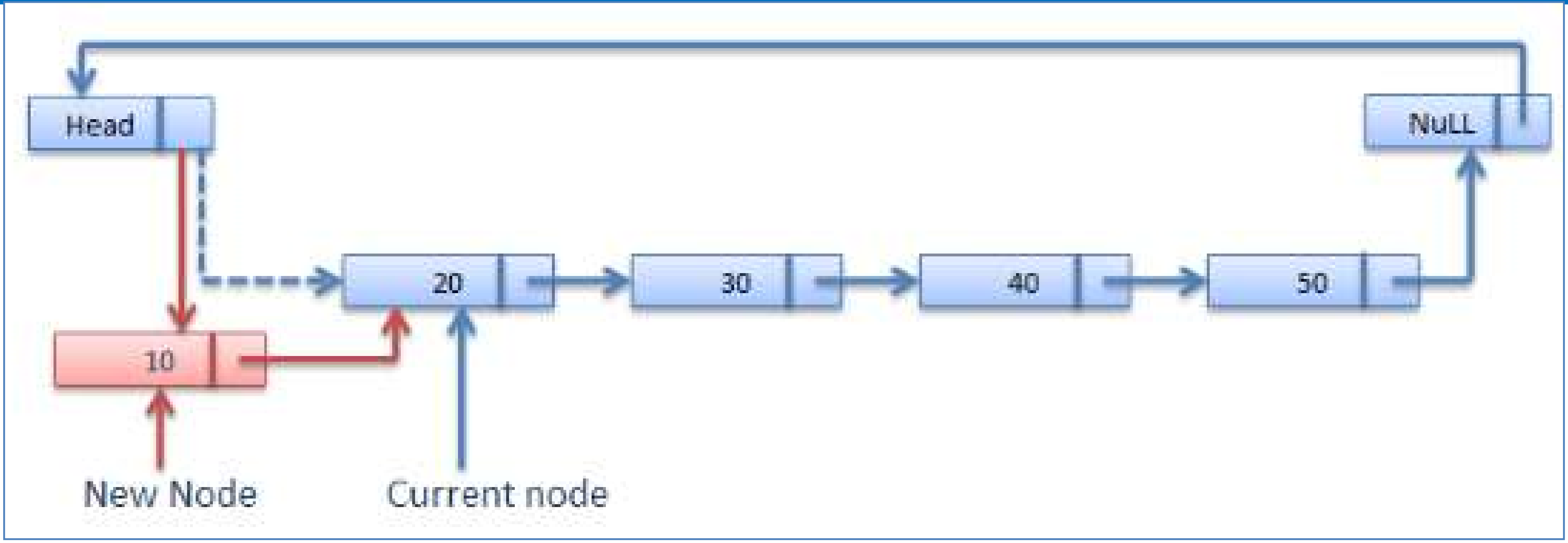
- There are three situation for inserting element in Circular linked list.

1.Insertion at the front of Circular linked list.
2.Insertion in the middle of the Circular linked list.
3.Insertion at the end of the Circular linked list.

Insertion In Circular Linked List

Insertion at the front of Circular linked list :

- Step1. Create the new node
- Step2. Set the new node’s next to itself (circular!)
- Step3. If the list is empty, return new node.
- Step4. Set our new node’s next to the front.
- Step5. Set tail’s next to our new node.
- Step6. Return the end of the list.

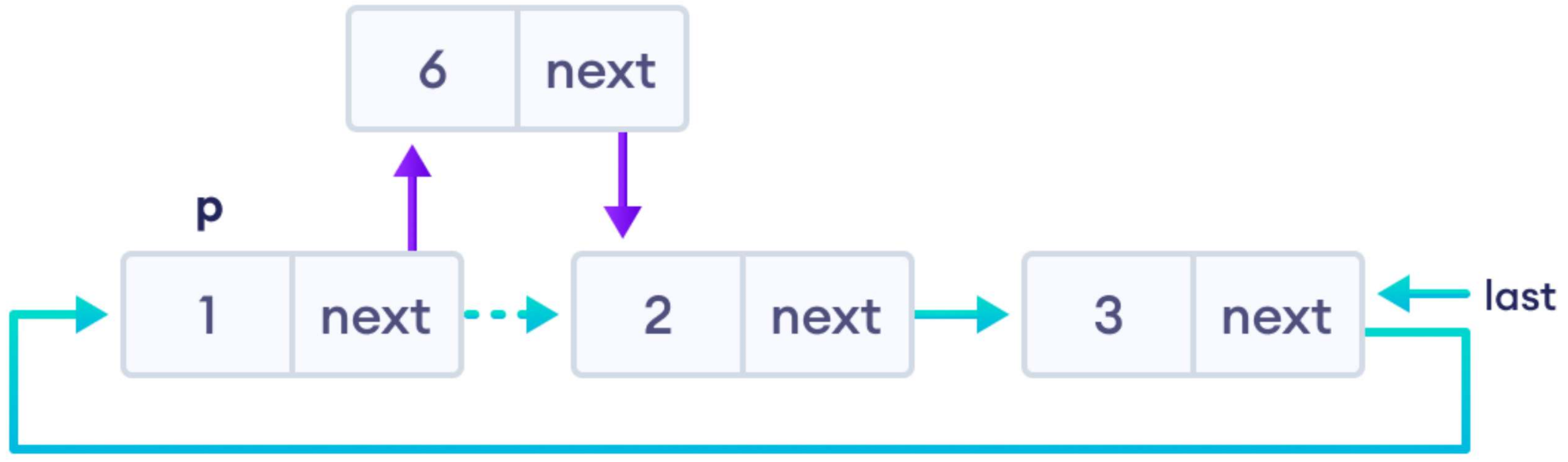


```
node* AddFront(node* tail, int num)
{
        node *temp = (node*)malloc(sizeof(node));
        temp->data = num;
        temp->next = temp;
        if (tail == NULL)
                return temp;
        temp->next = tail->next;
        tail->next = temp;
        return tail;
}
```

Insertion in between two nodes

Let's insert **newNode** after the first node.

- travel to the node given (let this node be **p**)
- point the **next** of **newNode** to the node next to **p**
- store the address of **newNode** at **next** of **p**

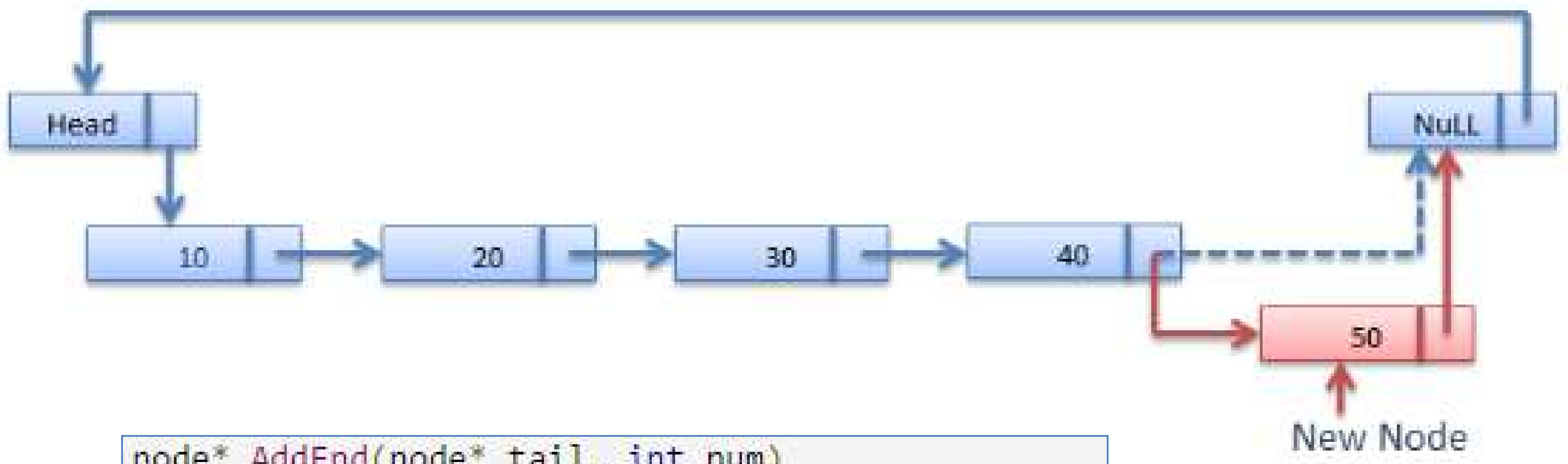


Insertion in between of two nodes

```
void insertPosition (int data, int pos, struct Node **head)
//function to insert element at specific position
{
  struct Node *newnode, *curNode;
  int i;

  if (*head == NULL)
    {
      printf ("List is empty");
    }
  if (pos == 1)
    {
      insertStart (head, data);
      return;
    }
  else
    {
      newnode = (struct Node *) malloc (sizeof (struct Node));
      newnode->data = data;
      curNode = *head;
      while (--pos > 1)
                {
                  curNode = curNode->next;
                }
      newnode->next = curNode->next;
      curNode->next = newnode;
    }
}
```

Insertion at the end of Circular linked list :

- Step1. Create the new node
- Step2. Set the new node’s next to itself (circular!)
- Step3. If the list is empty, return new node.
- Step4. Set our new node’s next to the front.
- Step5. Set tail’s next to our new node.
- Step6. Return the end of the list.



```
node* AddEnd(node* tail, int num)
{
        node *temp = (node*)malloc(sizeof(node));
        temp->data = num;
        temp->next = temp;
        if (tail == NULL)
                return temp;
        temp->next = tail->next;
        tail->next = temp;
        return temp;
}
```

Thank you