Linked Lists

Linked List

Lists, Array, List using arrays and pointers, Sparse matrices and its representation.

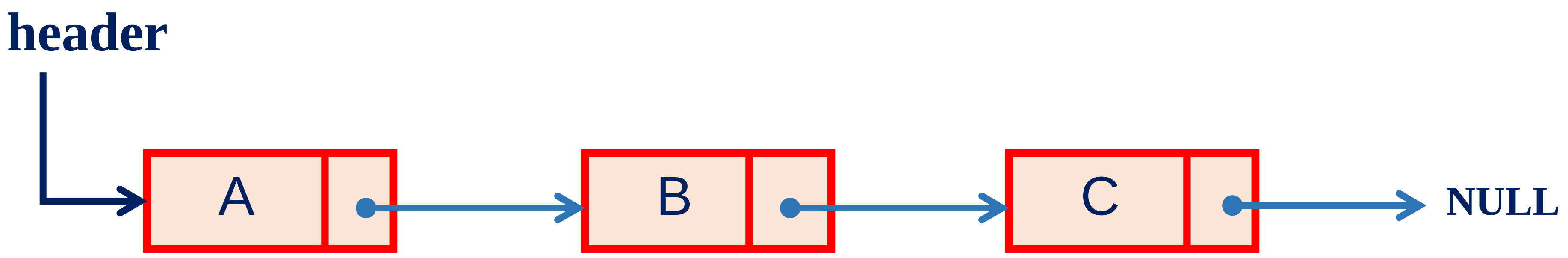
Stacks: Stack, operations, implementation using arrays, linked list and Stack Applications: Infix to postfix expression conversion, Evaluation of Postfix expressions, balancing the symbols.

Queue: Queues, operations, implementation using arrays, linked list & its applications. Circular queue: definition &its operations, De queue: definition & its types, applications.

Linked lists; Singly Linked, Doubly Linked, Circular Linked Lists, Polynomial ADT.

Linked List

- A linked list is a data structure which allows to store data dynamically and manage data efficiently.
- Typically, a linked list, in its simplest form looks like the following

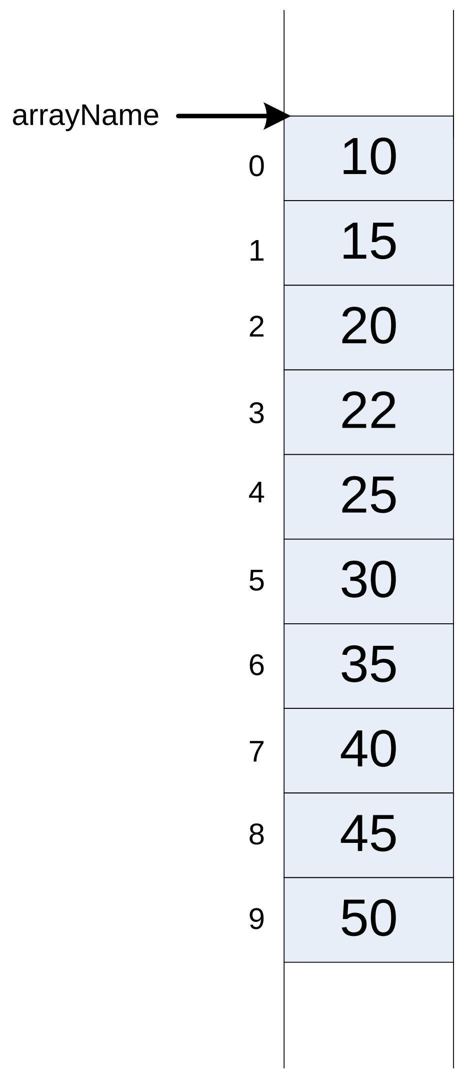


Linked List

- Few salient features
  - There is a pointer (called header) points the first element (also called node)
  - Successive nodes are connected by pointers.
  - Last element points to NULL.
  - It can grow or shrink in size during execution of a program.
  - It can be made just as long as required.
  - It does not waste memory space, consume exactly what it needs.

Arrays versus Linked Lists

Array: Contagious Storage



Array versus Linked Lists

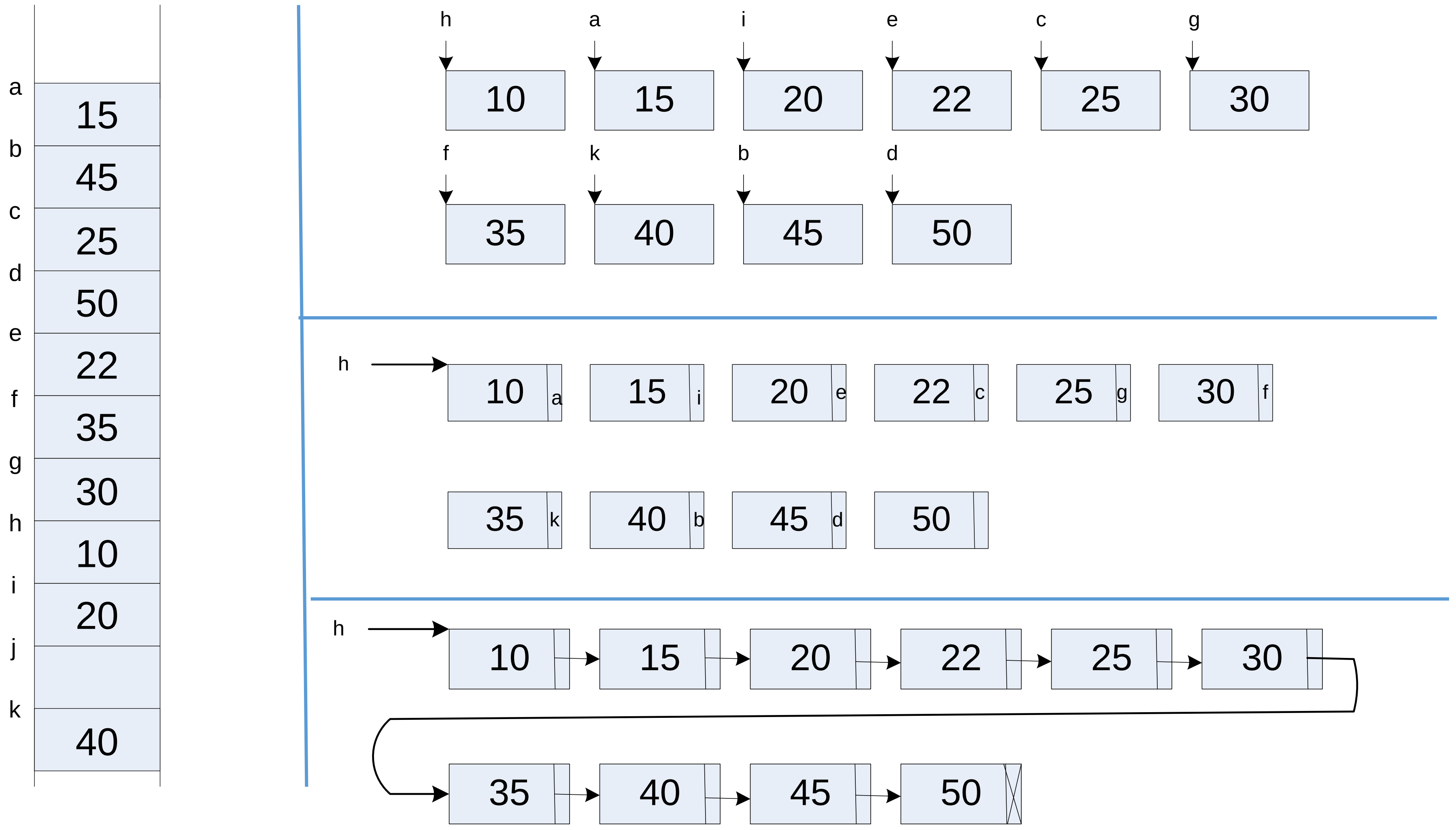
In arrays

- Elements are stored in a contagious memory locations
- Arrays are static data structure unless we use dynamic memory allocation

Arrays are suitable for

- Inserting/deleting an element **at the end**.
- Randomly accessing any element.
- **Searching** the list for a particular value.

Linked List: Non-Contagious Storage



Array versus Linked Lists

In Linked lists

- adjacency between any two elements are maintained by means of links or pointers

It is essentially a dynamic data structure

Linked lists are suitable for

- Inserting an element **at any position**.
- Deleting an element from **any where**.
- Applications where sequential access is required.
- In situations, where the number of elements cannot be predicted beforehand.

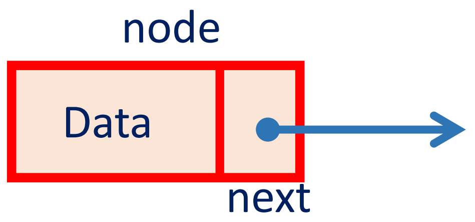
Defining a Node of a Linked List

Each structure of the list is called a node, and consists of two fields:

- Item (or) data
- Address of the next item in the list (or) pointer to the next node in the list

How to define a node of a linked list?

```
struct node
{
   int data;             /* Data */
   struct node *next; /* pointer*/
} ;
```



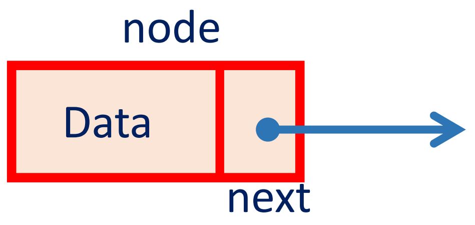
Note:

Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Defining a Node of a Linked List

Python

```
# Node class
class Node:

    # Function to initialize the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None # Initialize
        # next as null

# Linked List class


class LinkedList:

    # Function to initialize the Linked
    # List object
    def __init__(self):
        self.head = None
```



Note:

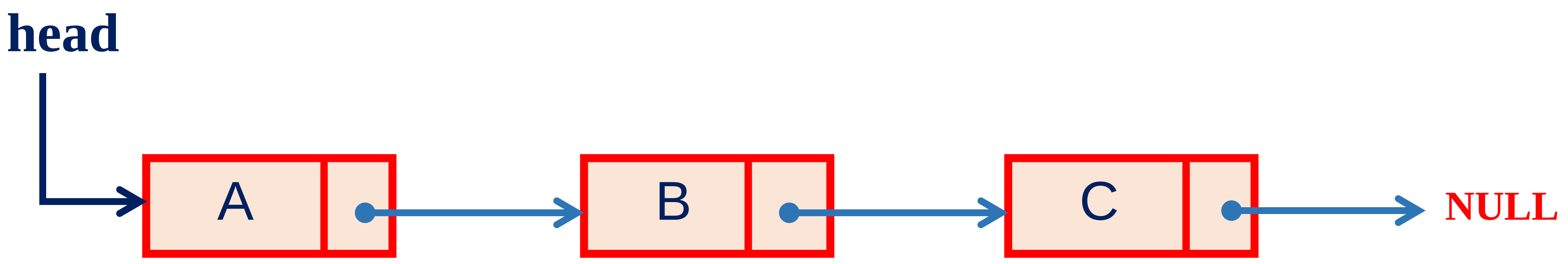
Such structures which contain a member field pointing to the same structure type are called self-referential structures.

Types of Lists: Single Linked List

Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.

Single linked list (or simply **linked list)**

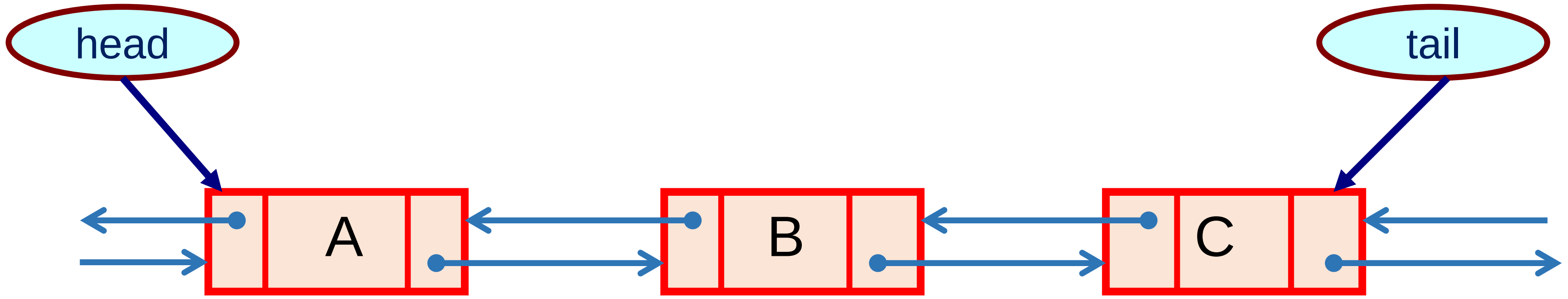
- A head pointer addresses the first element of the list.
- Each element points at a successor element.
- The last element has a link value NULL.



Types of Lists: Double Linked List

Double linked list

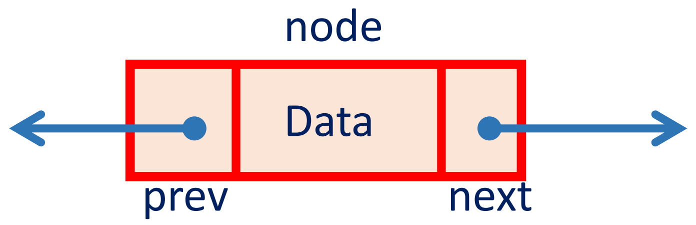
- Pointers exist between adjacent nodes in both directions.
- The list can be traversed either forward or backward.
- Usually, two pointers are maintained to keep track of the list, *head* and *tail*.



Defining a Node of a Double Linked List

Each node of doubly linked list (DLL) consists of three fields:

- Item (or) Data
- Pointer of the next node in DLL
- Pointer of the previous node in DLL



How to define a node of a doubly linked list (DLL)?

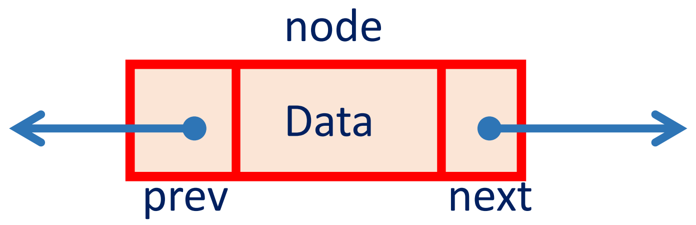
```
struct node
{
  int data;
  struct node *next; // Pointer to next node in DLL
  struct node *prev; // Pointer to previous node in DLL
};
```

Defining a Node of a Double Linked List

Python

Each node of doubly linked list (DLL) consists of three fields:

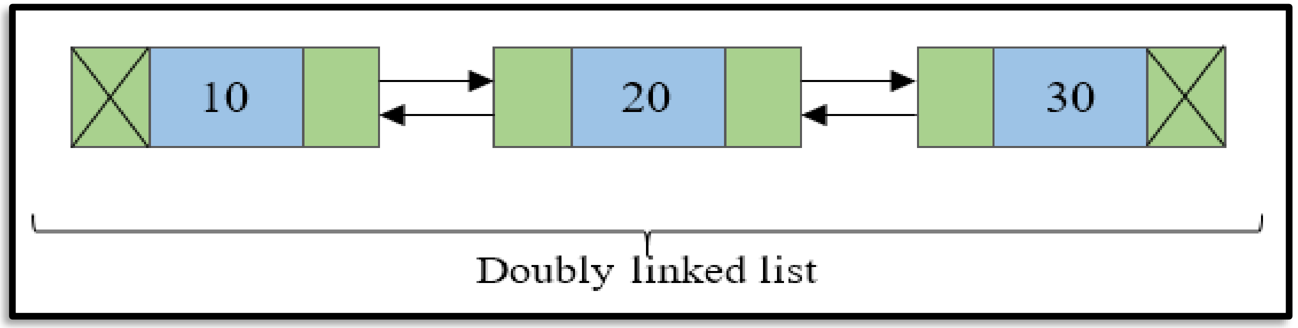
- Item (or) Data
- Pointer of the next node in DLL
- Pointer of the previous node in DLL



```
# Node of a doubly linked list


class Node:
    def __init__(self, next=None, prev=None, data=None):

        # reference to next node in DLL
        self.next = next

        # reference to previous node in DLL
        self.prev = prev
        self.data = data
```

Double Linked List

- Doubly linked list is a collection of nodes linked together in a sequential way.
- Doubly linked list is almost similar to singly linked list except it contains two address or reference fields, where one of the address field contains reference of the next node and other contains reference of the previous node.
- First and last node of a linked list contains a terminator generally a NULL value, that determines the start and end of the list.
- Doubly linked list is sometimes also referred as bi-directional linked list since it allows traversal of nodes in both direction.
- Since doubly linked list allows the traversal of nodes in both direction, we can keep track of both first and last nodes.



Double versus Single Linked List

Advantages over singly linked list

1) A DLL can be traversed in both forward and backward direction.
2) The delete operation in DLL is more efficient if pointer to the node to be deleted is given.

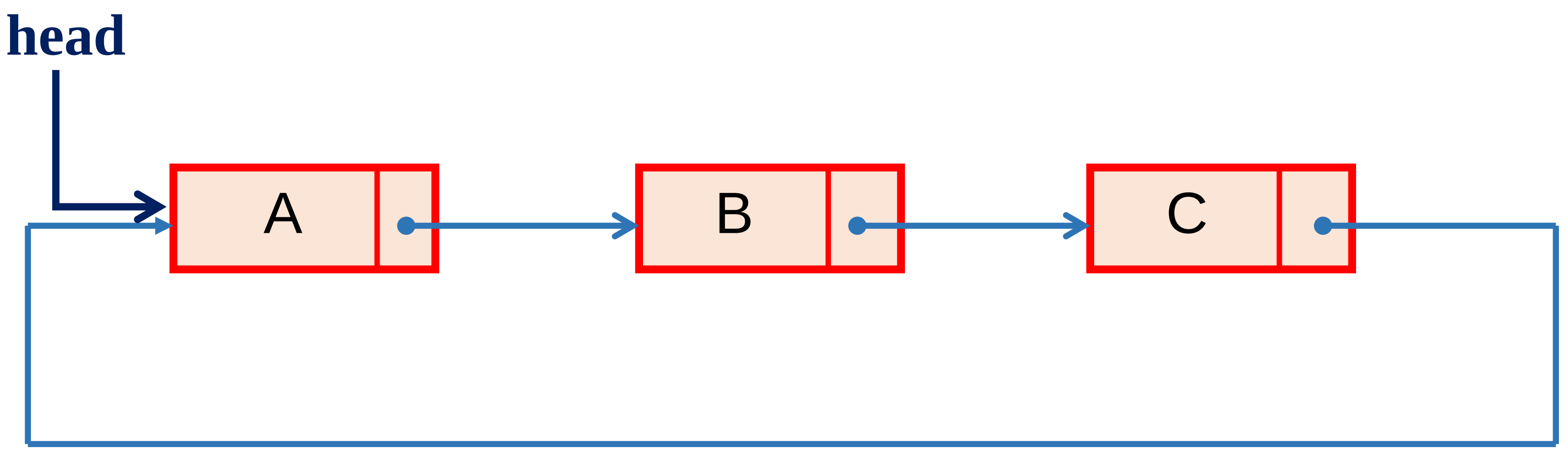
Disadvantages over singly linked list

1) Every node of DLL Require extra space for an previous pointer.
2) All operations require an extra pointer previous to be maintained.

Types of Lists: Circular Linked List

Circular linked list

- The pointer from the last element in the list points back to the first element.



Circular Linked List

- A circular linked list is basically a linear linked list that may be single- or double-linked.
- The only difference is that there is no any NULL value terminating the list.
- In fact in the list every node points to the next node and last node points to the first node, thus forming a circle. Since it forms a circle with no end to stop it is called as **circular linked list**.
- In circular linked list there can be no starting or ending node, whole node can be traversed from any node.
- In order to traverse the circular linked list, only once we need to traverse entire list until the starting node is not traversed again.
- A circular linked list can be implemented using both singly linked list and doubly linked list.

Example 1: Creating a Single Linked List

Linked list to store and print roll number, name and age of 3 students.

```
#include <stdio.h>
struct stud
{
   int roll;
   char name[30];
   int age;
   struct stud *next;
};
main()
{
   struct stud n1, n2, n3;
   struct stud *p;
   scanf ("%d %s %d", &n1.roll, n1.name, &n1.age);
   scanf ("%d %s %d", &n2.roll,n2.name, &n2.age);
   scanf ("%d %s %d", &n3.roll,n3.name, &n3.age);
```

Example 1: Creating a Single Linked List

```
    n1.next = &n2 ;
    n2.next = &n3 ;
    n3.next = NULL ;
/* Now traverse the list and print the elements */
    p = &n1 ;         /* point to 1st element */
    while (p != NULL)
    {
        printf (“\n %d %s %d”, p->roll, p->name, p->age);
        p = p->next;
    }
}
```

Example 1: Illustration

The structure:

```
struct stud
{
  int roll;
  char name[30];
  int age;
  struct stud *next;
};
```

Also assume the list with three nodes n1, n2 and n3 for 3 students.

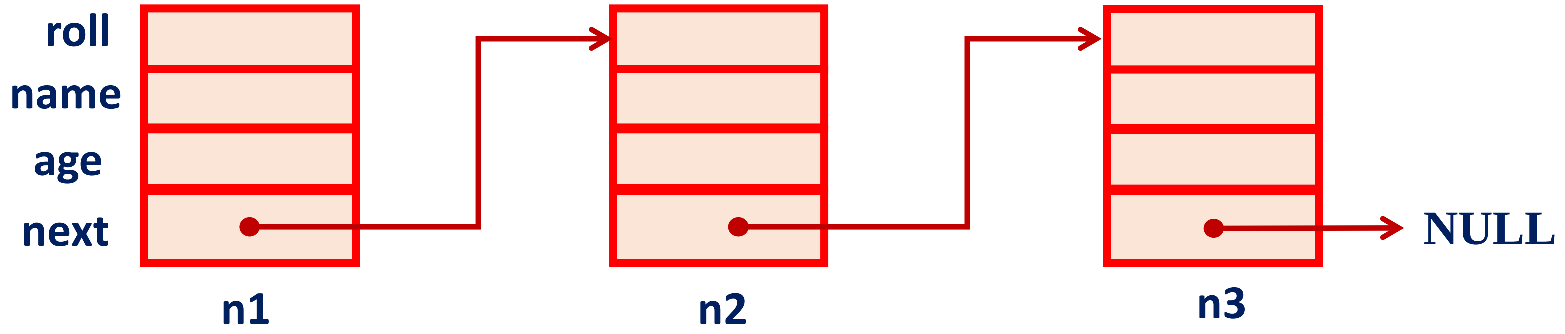
```
struct stud n1, n2, n3;
```

Example 1: Illustration

To create the links between nodes, it is written as:

```
n1.next = &n2 ;
n2.next = &n3 ;
n3.next = NULL ;   /* No more nodes follow */
```

- Now the list looks like:



Example 2: Creating a Single Linked List

C-program to store 10 values on a linked list reading the data from keyboard.

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

struct node {
    int data;                       //Data part
    struct node *next;              //Address part
}*header;

void createList(int n);               /* Functions to create a list*/

int main()
{
    int n;
    printf("Enter the total number of nodes: ");
    scanf("%d", &n);
    createList(n);
    return 0;
}
```

Example 2: Creating a Single Linked List

```
void createList(int n)
{
    struct node *newNode, *temp;
    int data, i;

    /* A node is created by allocating memory to a structure */
    newNode = (struct node *)malloc(sizeof(struct node));

    /* If unable to allocate memory for head node */
    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        printf("Enter the data of node 1: ");
        scanf("%d", &data);

        newNode->data = data; //Links the data field with data
        newNode->next = NULL; //Links the address field to NULL
        header = newNode;     //Header points to the first node
        temp = newNode;       //First node is the current node
```

Example 2: Creating a Single Linked List

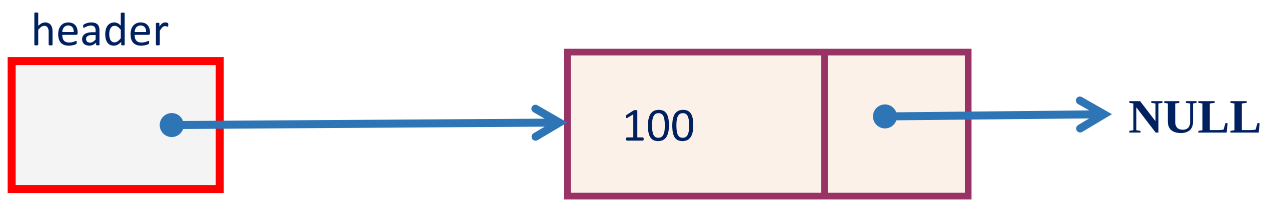
```
    for(i=2; i<= n; i++)
    {
      /* A newNode is created by allocating memory */
      newNode = (struct node *)malloc(sizeof(struct node));

      if(newNode == NULL)
      {
         printf("Unable to allocate memory.");
         break;
      }
      else
      {
         printf("Enter the data of node %d: ", i);
         scanf("%d", &data);

         newNode->data = data; //Links the data field of newNode with data
         newNode->next = NULL; //Links the address field of newNode with NULL

         temp->next = newNode; //Links previous node i.e. temp to the newNode
         temp = temp->next;
       }
     }
   }
}
```

Example 2: Illustration

- To start with, we have to create a node (the first node), and make header point to it.

```
newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = data;        //Links the data field with data
newNode->next = NULL;        //Links the address field to NULL
header = newNode;
temp = newNode;
```

It creates a single node. For example, if the data entered is 100 then the list look like



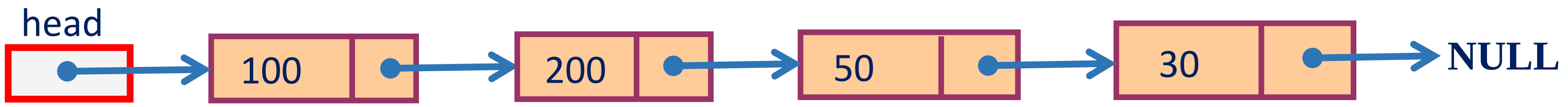
Creating a single linked list

If we need n number of nodes in the linked list:

- Allocate n newNodes, one by one.
- Read in the data for the newNodes.
- Modify the links of the newNodes so that the chain is formed.

```
newNode = (struct node *)malloc(sizeof(struct node));
newNode->data = data;      //Links the data field of newNode with data
newNode->next = NULL;      //Links the address field of newNode with NULL

temp->next = newNode;      //Links previous node i.e. temp to the newNode
temp = temp->next;
```

It creates n number of nodes . For e.g. if the data entered is 200, 50, 30 then the list look like



Creating a single linked list

Python

```python
# A single node of a singly linked list
class Node:
  # constructor
  def __init__(self, data = None, next=None):
    self.data = data
    self.next = next

# A Linked List class with a single head node
class LinkedList:
  def __init__(self):
    self.head = None

  # insertion method for the linked list
  def insert(self, data):
    newNode = Node(data)
    if(self.head):
      current = self.head
      while(current.next):
        current = current.next
      current.next = newNode
    else:
      self.head = newNode
```

```python
  # print method for the linked list
  def printLL(self):
    current = self.head
    while(current):
      print(current.data)
      current = current.next

# Singly Linked List with insertion
and print methods
LL = LinkedList()
LL.insert(3)
LL.insert(4)
LL.insert(5)
LL.printLL()
```

Example 3: Creating a Single Linked List

C-program to copy an array to a single linked list.

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

struct node {
    int data;                    //Data part
    struct node *next;           //Address part
};

int main()
{
    struct node *header, *newNode, *temp;
     int data, i, n, a[100];
    printf("Enter the total number of data: ");
    scanf("%d", &n);
    // Write code here to initialize the array
a with n elements //

. . .
```

Header node initialization

Example 2: Creating a Single Linked List

```
    /* A node is created by allocating memory to a
structure */
    newNode = (struct node *)malloc(sizeof(struct node));

    /* If unable to allocate memory for head node */
    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = a[0]; //Links the data field with
data
        newNode->next = NULL; //Links the address field
to NULL
        header = newNode;     //Header points to the
first node
        temp = header;
```

First node as Header node initialization

Example 2: Creating a Single Linked List

```
    for(i = 1; i <= n; i++)
    {
      /* A newNode is created by allocating memory */
      newNode = (struct node *)malloc(sizeof(struct node));

      if(newNode == NULL)
      {
         printf("Unable to allocate memory.");
         break;
      }
      else
      {
         newNode->data = a[i]; //Links the data field of newNode with a[i]
         newNode->next = NULL; //Links the address field of newNode with NULL

         temp->next = newNode; //Links previous node i.e. temp to the newNode
         temp = temp->next;
       }
     }
   }
}
```

For remaining node

Traversing a Linked List

Single Linked List: Traversing

Once the linked list has been constructed and `header` points to the first node of the list,

- Follow the pointers.
- Display the contents of the nodes as they are traversed.
- Stop when the `next` pointer points to NULL.

The function **`traverseList`**`(struct Node *)` is given in the next slide. This function to be called from **`main`**`()` function as:

```
int main()
{
    // Assume header, the pointer to the linked list is given as an input
    printf("\n Data in the list \n");
    traverseList(header);
    return 0;
}
```

Single linked list: Traversing

```
void traverseList(struct Node *header)
{
  struct node *temp;

  /* If the list is empty i.e. head = NULL */
  if(header == NULL)
  {
    printf("List is empty.");
  }
  else
  {
    temp = header;
    while(temp != NULL)
    {
     printf("Data = %d\n", temp->data); //Prints the data of current node
     temp = temp->next;         //Advances the position of current node
    }
  }
}
```

Single linked list: Traversing

Python

```
# Python program for traversal of a linked list
# Node class

class Node:

    # Function to initialise the node object
    def __init__(self, data):
        self.data = data  # Assign data
        self.next = None # Initialize next as null

# Linked List class contains a Node object
class LinkedList:
     # Function to initialize head
    def __init__(self):
        self.head = None

    # This function prints contents of linked list
    # starting from head
    def printList(self):
        temp = self.head
        while (temp):
            print(temp.data)
            temp = temp.next
```

```
# Code execution starts here
if __name__ == '__main__':

    # Start with the empty list
    llist = LinkedList()

    llist.head = Node(1)
    second = Node(2)
    third = Node(3)

    llist.head.next = second  # Link first node
with second
    second.next = third  # Link second node with
the third node

    llist.printList()
```

Insertion in a Linked List

Single Linked List: Insertion

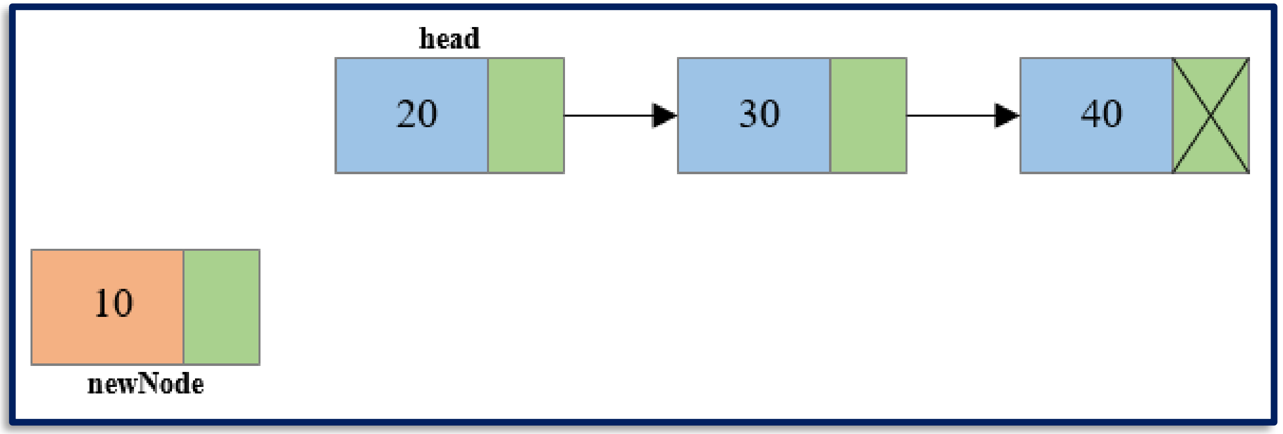
Insertion steps:

- Create a new node
- Start from the header node
- Manage links to
  - Insert at front
  - Insert at end
  - Insert at any position

Insertion at Front

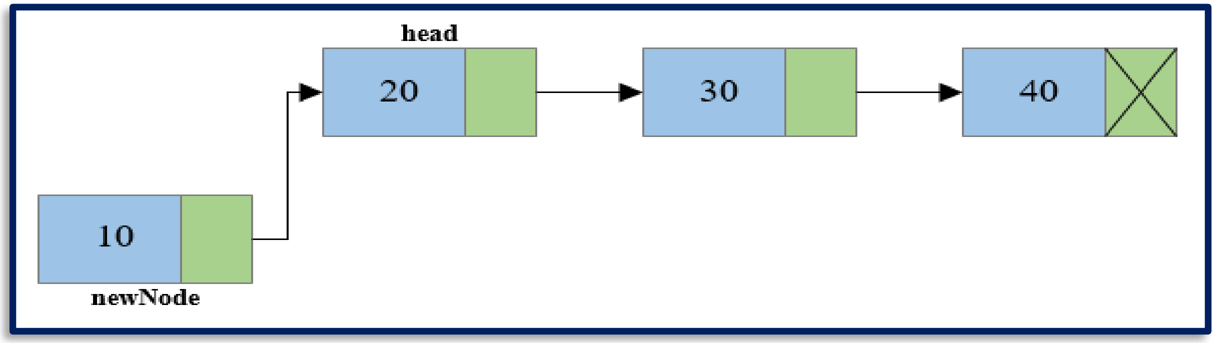
Steps to insert node at the beginning of singly linked list

Step 1: Create a new node.

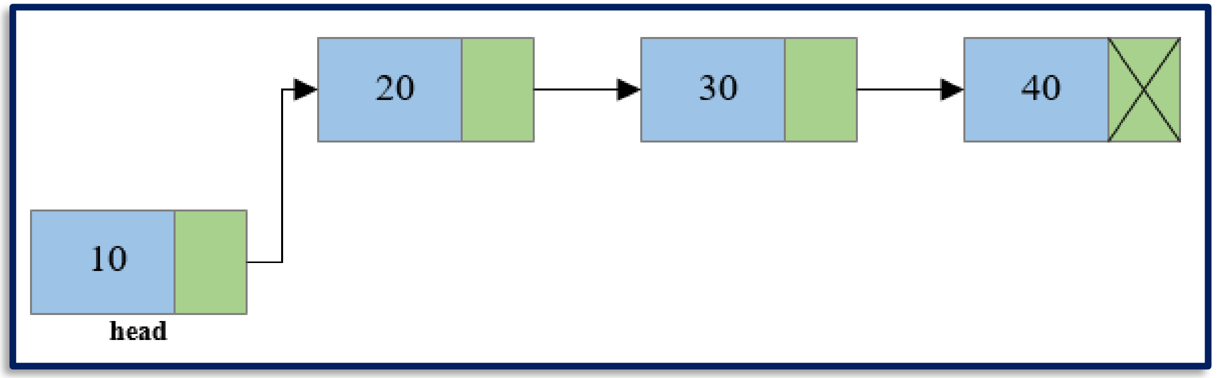


Insertion at Front

Step 2: Link the newly created node with the head node, i.e. the **newNode** will now point to **head** node.



Step 3: Make the new node as the head node, i.e. now **head** node will point to **newNode**.



Insertion at front

```
/*Create a new node and insert at the beginning of the linked list.*/

void insertNodeAtBeginning(int data)
{
    struct node *newNode;
    newNode = (struct node*)malloc(sizeof(struct node));

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data;    //Links the data part
        newNode->next = head;    //Links the address part

        head = newNode;          //Makes newNode as first node

        printf("DATA INSERTED SUCCESSFULLY\n");
    }
}
```

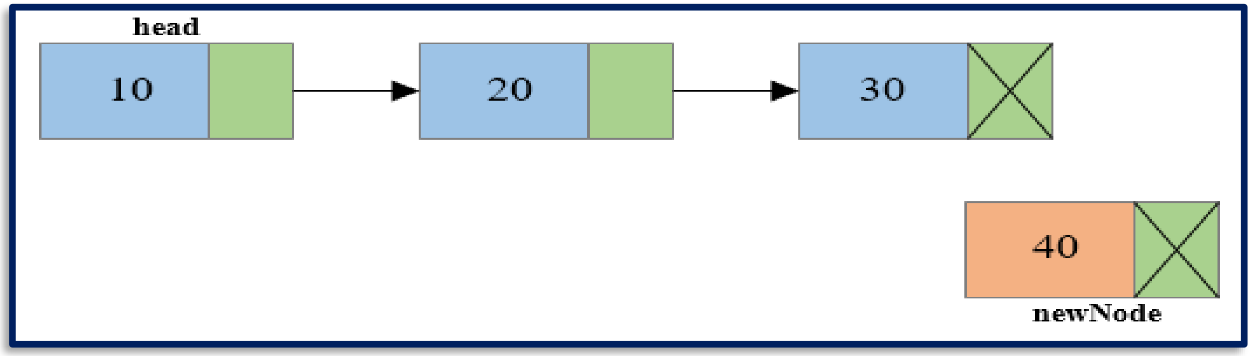
Insertion at front

```
# This function is in LinkedList class
# Function to insert a new node at the beginning
def push(self, new_data):


    # 1 & 2: Allocate the Node &
    #        Put in the data
    new_node = Node(new_data)

    # 3. Make next of new Node as head
    new_node.next = self.head

    # 4. Move the head to point to new Node
    self.head = new_node
```

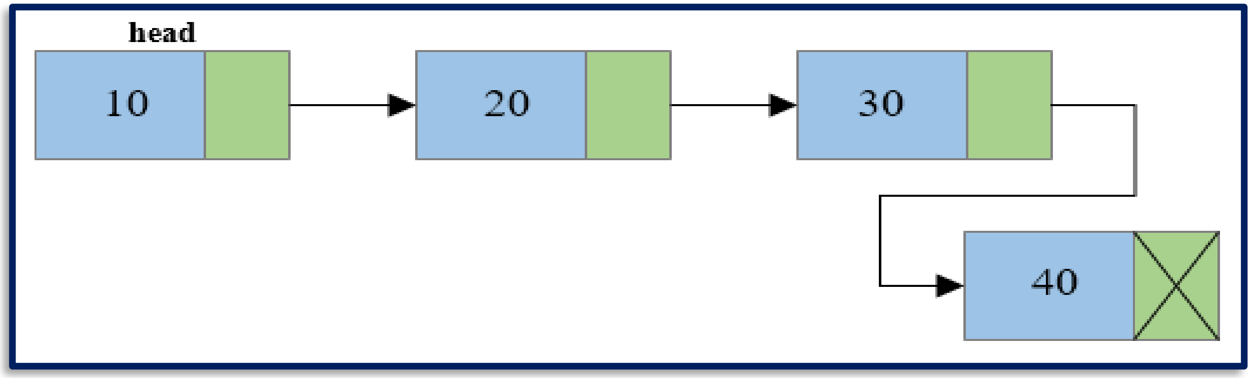
Single Linked List: Insertion at End

Steps to insert node at the end of Singly linked list

Step 1: Create a new node and make sure that the address part of the new node points to NULL.
i.e. `newNode->next=NULL`



Step 2: Traverse to the last node of the linked list and connect the last node of the list with the new node, i.e. last node will now point to new node. (`lastNode->next = newNode`).



Insertion at End

```c
/* Create a new node and insert at the end of the linked list. */
void insertNodeAtEnd(int data)
{
    struct node *newNode, *temp;
    newNode = (struct node*)malloc(sizeof(struct node));
    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data; //Links the data part
        newNode->next = NULL;
        temp = head;

        while(temp->next != NULL) //Traverse to the last node
            temp = temp->next;

        temp->next = newNode; //Links the address part
        printf("DATA INSERTED SUCCESSFULLY\n");
    }
}
```

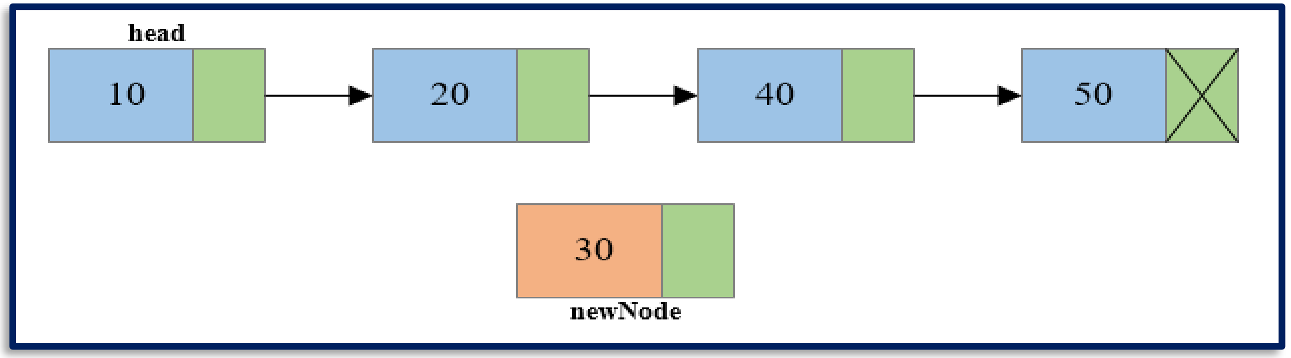
Insertion at End

```
# This function is defined in Linked List class appends a new node
at the end.
# This method is defined inside LinkedList class shown above
def append(self, new_data):

   # 1. Create a new node
    # 2. Put in the data
    # 3. Set next as None
    new_node = Node(new_data)


    # 4. If the Linked List is empty, then
    #    make the new node as head
    if self.head is None:
        self.head = new_node
        return


    # 5. Else traverse till the last node
    last = self.head
    while (last.next):
        last = last.next

      # 6. Change the next of last node
    last.next = new_node
```

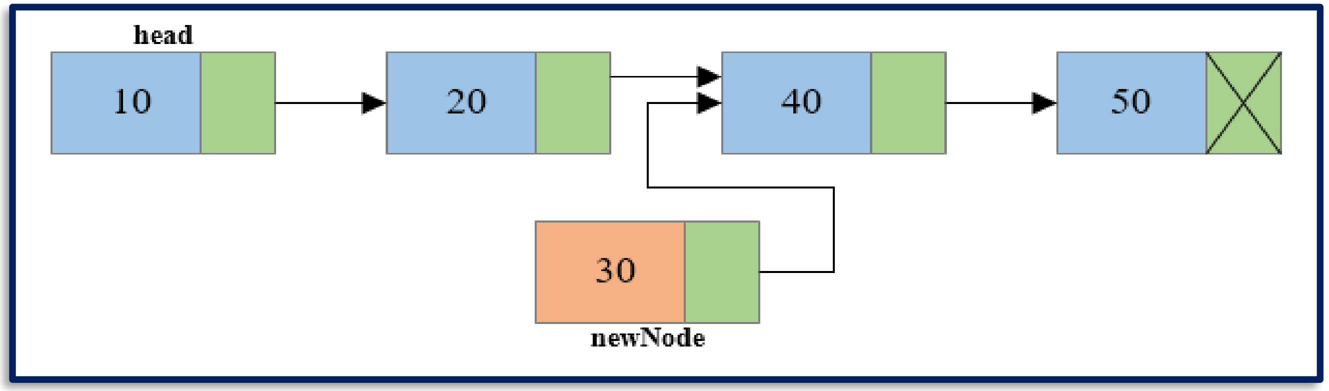
Single Linked List: Insertion at any Position

Steps to insert node at any position of Singly Linked List

Step 1: Create a new node.



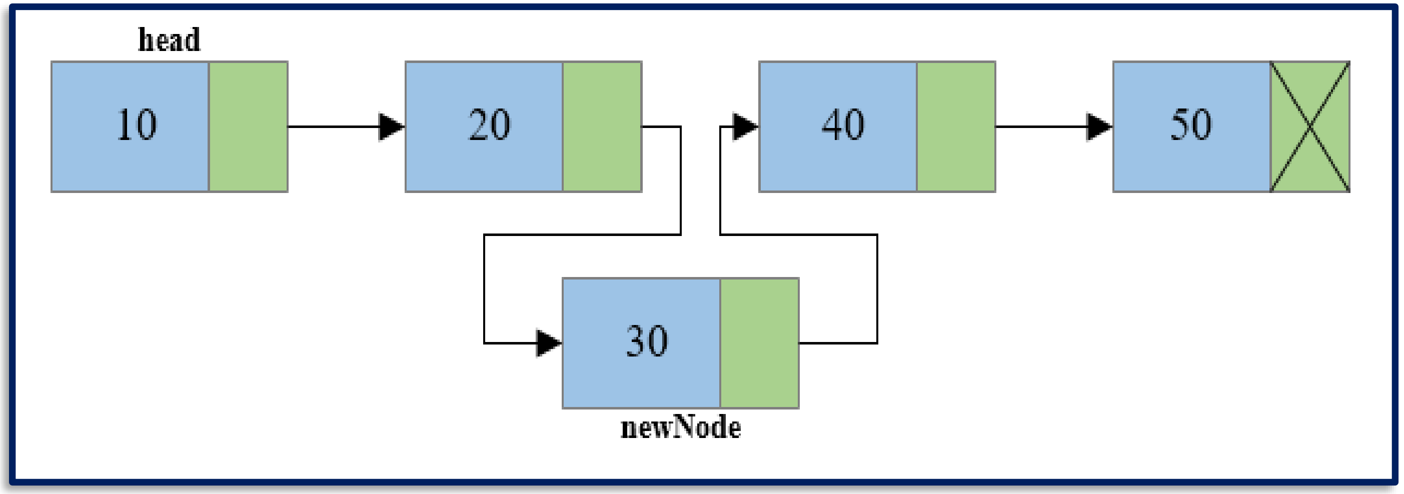
Step 2: Traverse to the n-1th position of the linked list and connect the new node with the n+1th node. (`newNode->next = temp->next`) where temp is the n-1th node.



Single Linked List: Insertion at any position

Step 3: Now at last connect the n-1th node with the new node i.e. the n-1th node will now point to new node.

(`temp->next = newNode`) where temp is the n-1th node.



Insertion at any Position

```
/* Create a new node and insert at middle of the linked list.*/

void insertNodeAtMiddle(int data, int position)
{
    int i;
    struct node *newNode, *temp;

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

    if(newNode == NULL)
    {
        printf("Unable to allocate memory.");
    }
    else
    {
        newNode->data = data;    //Links the data part
        newNode->next = NULL;

        temp = head;
```

Insertion at any Position

```
        for(i=2; i<=position-1; i++) /*  Traverse to the n-1 position */
        {
            temp = temp->next;

            if(temp == NULL)
                break;
        }
        if(temp != NULL)
        {
            /* Links the address part of new node */
            newNode->next = temp->next;

            /* Links the address part of n-1 node */
            temp->next = newNode;

            printf("DATA INSERTED SUCCESSFULLY\n");
        }
        else
        {
            printf("UNABLE TO INSERT DATA AT THE GIVEN POSITION\n");
        }
    }
}
```

Insertion at any Position

```
# This function is in LinkedList class.
# Inserts a new node after the given
# prev_node. This method is defined
# inside LinkedList class shown above
def insertAfter(self, prev_node, new_data):

    # 1. Check if the given prev_node exists
    if prev_node is None:
        print "The given previous node must in LinkedList."
        return

    # 2. Create new node &
    # 3. Put in the data
    new_node = Node(new_data)

    # 4. Make next of new Node as next of prev_node
    new_node.next = prev_node.next

    # 5. make next of prev_node as new_node
    prev_node.next = new_node
```

Insertion - SL

```
# Node class
class Node:

      # Function to initialize the
    # node object
    def __init__(self, data):


        # Assign data
        self.data = data


        # Initialize next as null
        self.next = None


# Linked List class contains a
# Node object
class LinkedList:


    # Function to initialize head
    def __init__(self):
        self.head = None
```

```
# Utility function to print the
    # linked list
    def printList(self):
        temp = self.head
        while (temp):
            print temp.data,
            temp = temp.next
```

```
# Code execution starts here
if __name__=='__main__':

    # Start with the empty list
    llist = LinkedList()

    # Insert 6.  So linked list
    becomes 6->None
    llist.append(6)

    # Insert 7 at the beginning. So
    # linked list becomes 7->6->None
    llist.push(7);

    # Insert 1 at the beginning. So
    # linked list becomes 1->7->6->None
    llist.push(1);

    # Insert 4 at the end. So linked list
    # becomes 1->7->6->4->None
    llist.append(4)

    # Insert 8, after 7. So linked list
    # becomes 1 -> 7-> 8-> 6-> 4-> None
    llist.insertAfter(llist.head.next, 8)

    print 'Created linked list is:',
    llist.printList()
```

Deletion from a Linked List

Single Linked List: Deletion

Deletion steps

- Start from the header node
- Manage links to
  - Delete at front
  - Delete at end
  - Delete at any position
- freeingup the node as free space.

Free Memory after Deletion

- Do not forget to free() memory location dynamically allocated for a node after deletion of that node.

- It is the programmer's responsibility to free that memory block.

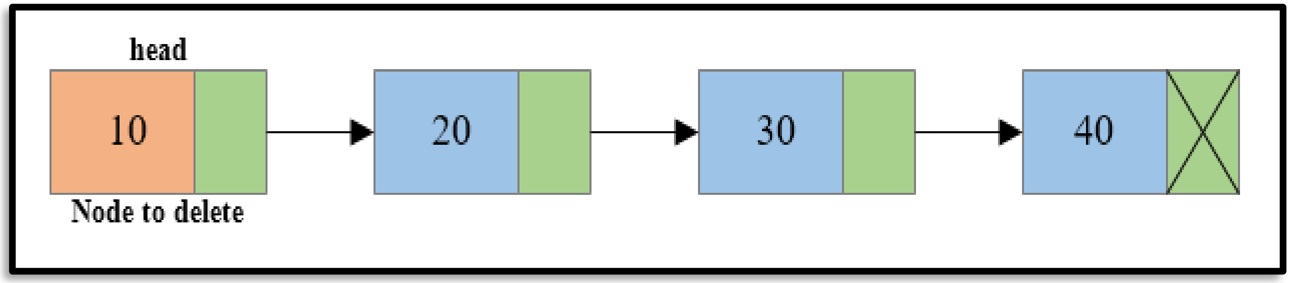
- Failure to do so may create a dangling pointer – a memory, that is not used either by the programmer or by the system.

- The content of a free memory is not erased until it is overwritten.

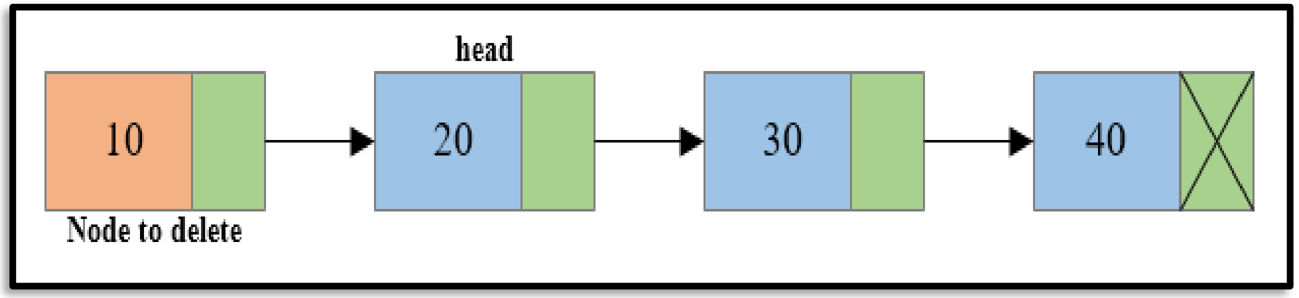
Single Linked List: Deletion at Front

Steps to delete first node of Singly Linked List

Step 1: Copy the address of first node i.e. **head** node to some temp variable say **toDelete**.

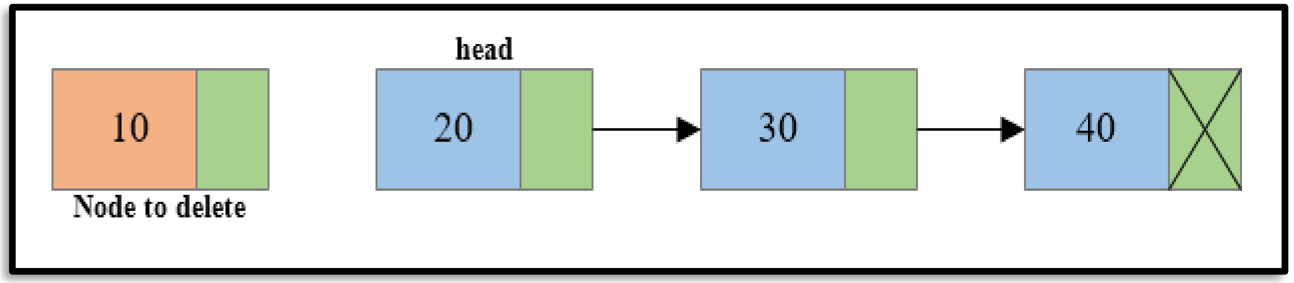


Step 2: Move the head to the second node of the linked list (`head = head->next`).

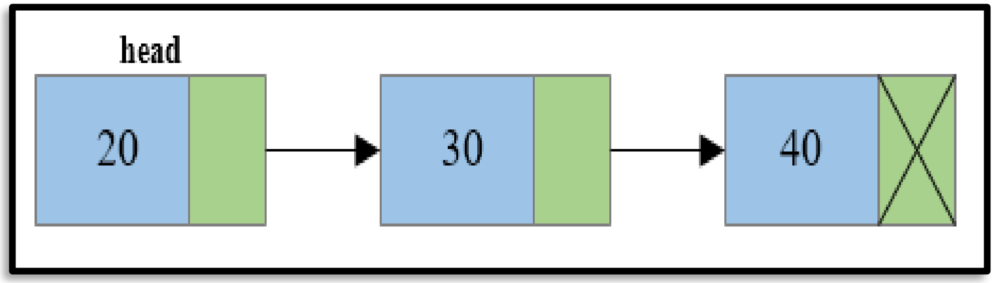


Single linked list: Deletion at front

Step 3: Disconnect the connection of first node to second node.



Step 4: Free the memory occupied by the first node.



Deletion at Front

```
/* Delete the first node of the linked list */
void deleteFirstNode()
{
    struct node *toDelete;

    if(head == NULL)
    {
        printf("List is already empty.");
    }
    else
    {
        toDelete = head;
        head = head->next;

        printf("\nData deleted = %d\n", toDelete->data);

        /* Clears the memory occupied by first node*/
        free(toDelete);

        printf("SUCCESSFULLY DELETED FIRST NODE FROM LIST\n");
    }
}
```

Deletion at Front

```
class LinkedList:

     # Function to initialize head
    def __init__(self):
        self.head = None


    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node


    # Given a reference to the head of a list and a 
key,
    # delete the first occurrence of key in linked list
    def deleteNode(self, key):


        # Store head node
        temp = self.head
```

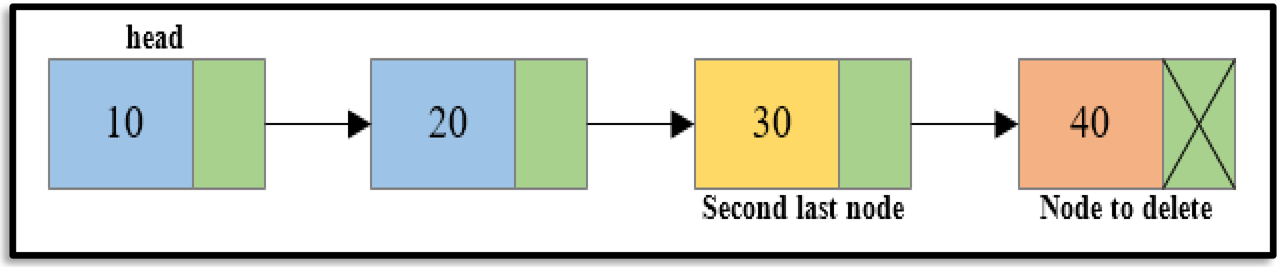
```
         # If head node itself holds the key to be 
deleted
        if (temp is not None):
            if (temp.data == key):
                self.head = temp.next
                temp = None
                return
```

```
        # Search for the key to be deleted, keep track of the
        # previous node as we need to change 'prev.next'
        while(temp is not None):
            if temp.data == key:
                break
            prev = temp
            temp = temp.next


        # if key was not present in linked list
        if(temp == None):
            return


        # Unlink the node from linked list
        prev.next = temp.next


        temp = None


    # Utility function to print the linked LinkedList


    def printList(self):
        temp = self.head
        while(temp):
            print(" %d" % (temp.data)),
            temp = temp.next
```

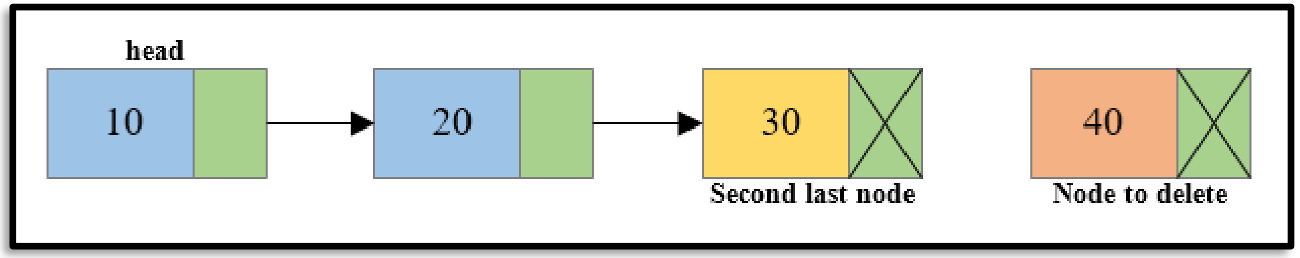
Single linked list: Deletion at End

Steps to delete last node of a Singly Linked List

Step 1: Traverse to the last node of the linked list keeping track of the second last node in some temp variable say **secondLastNode**.

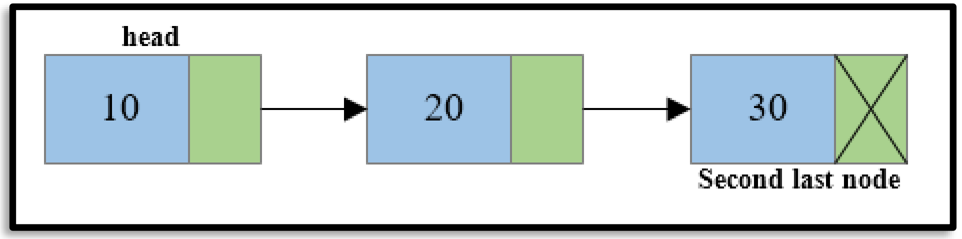


Step 2: If the last node is the head node then make the head node as NULL else disconnect the second last node with the last node i.e. `secondLastNode->next = NULL`



Single linked list: Deletion at End

Step 3: Free the memory occupied by the last node.



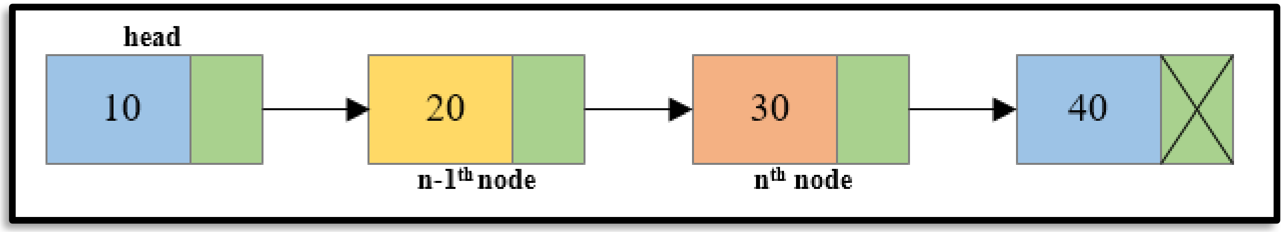
Deletion at End

```
/* Delete the last node of the linked list */
void deleteLastNode()
{
    struct node *toDelete, *secondLastNode;
    toDelete = head;
    secondLastNode = head;

    while(toDelete->next != NULL) /* Traverse to the last node of the list*/
     {
       secondLastNode = toDelete;
       toDelete = toDelete->next;
     }
    if(toDelete == head)
     {
       head = NULL;
     }
    else
     {
      /* Disconnects the link of second last node with last node */
       secondLastNode->next = NULL;
     }
      /* Delete the last node */
    free(toDelete);
}
```

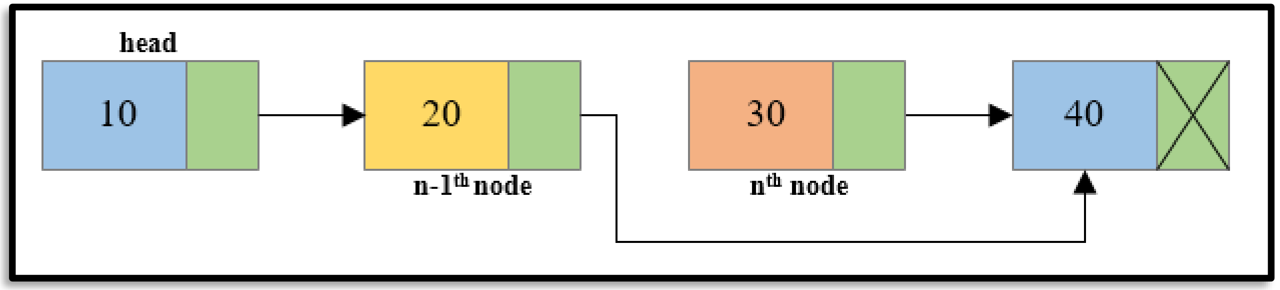
Single Linked List: Deletion at any Position

Steps to delete a node at any position of Singly Linked List

Step 1: Traverse to the nth node of the singly linked list and also keep reference of n-1th node in some temp variable say **prevNode**.

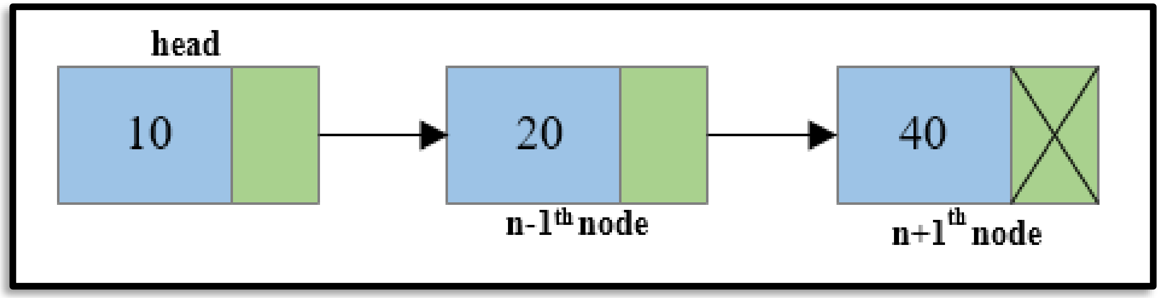


Step 2: Reconnect n-1th node with the n+1th node i.e. `prevNode->next = toDelete->next` (Where **prevNode** is n-1th node and **toDelete** node is the nth node and `toDelete->next` is the n+1th node).



Single Linked List: Deletion at any Position

Step 3: Free the memory occupied by the nth node i.e. **toDelete** node.



Deletion at any Position

```
/* Delete the node at any given position of the linked list  */
void deleteMiddleNode(int position)
{
    int i;
    struct node *toDelete, *prevNode;
    if(head == NULL)
    {
        printf("List is already empty.");
    }
    else
    {
        toDelete = head;
        prevNode = head;

        for(i=2; i<=position; i++)
        {
            prevNode = toDelete;
            toDelete = toDelete->next;

            if(toDelete == NULL)
                break;
        }
```

Deletion at any Position

```
        if(toDelete != NULL)
        {
            if(toDelete == head)
                head = head->next;

            prevNode->next = toDelete->next;
            toDelete->next = NULL;

            /* Deletes the n node */
            free(toDelete);

            printf("SUCCESSFULLY DELETED NODE FROM MIDDLE OF LIST\n");
        }
        else
        {
            printf("Invalid position unable to delete.");
        }
    }
}
```

Single Linked List: Reversing

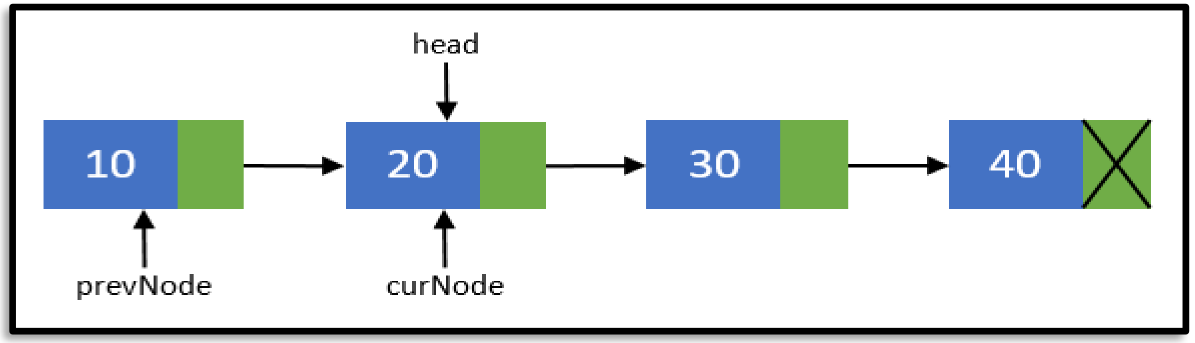
Reversing a list can be performed in two ways:

- Iterative method
- Recursive method

Steps to reverse a Singly Linked List using Iterative method

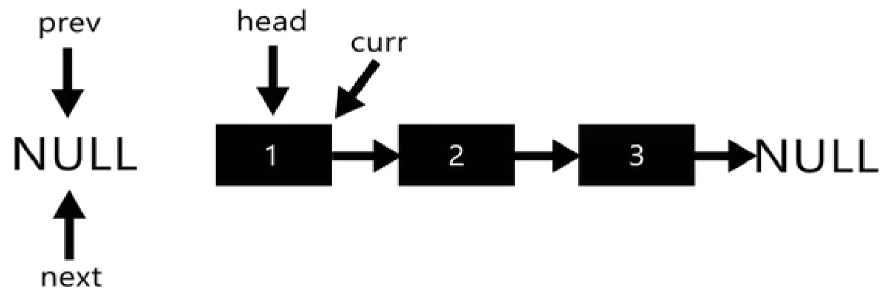
Step 1: Create two more pointers other than **head** namely **prevNode** and **curNode** that will hold the reference of previous node and current node respectively.

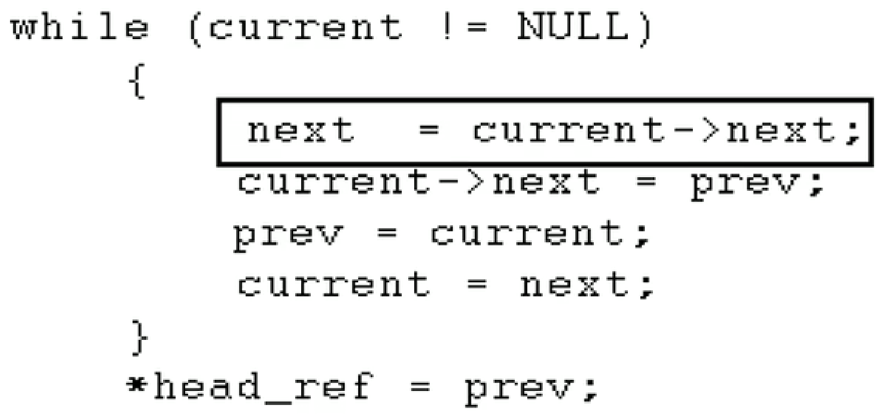
- Make sure that prevNode points to first node i.e. `prevNode = head.`
- head should now point to its next node i.e. `head = head->next.`
- curNode should also points to the second node i.e. `curNode = head.`



Single Linked List: Reversing

Follow the steps below to solve the problem:
Initialize three pointers
prev as NULL, **curr** as **head**, and **next** as NULL.

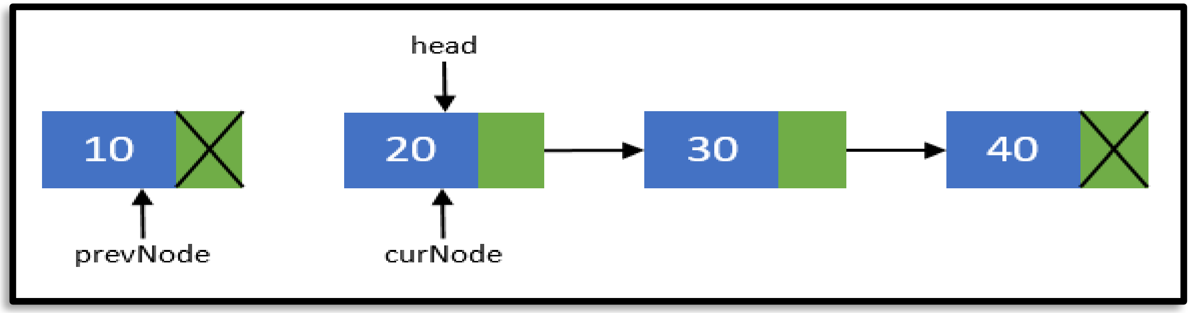




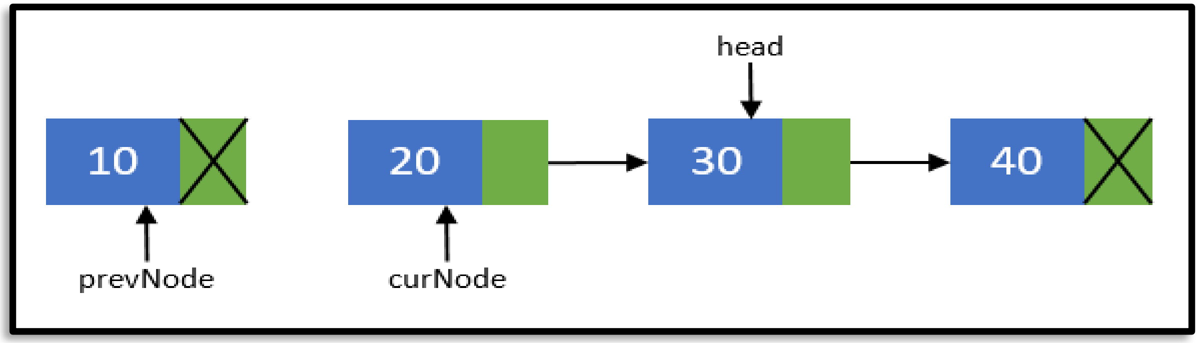
```
while (current != NULL)
    {
        next  = current->next;
        current->next = prev;
        prev = current;
        current = next;
    }
    *head_ref = prev;
```

Reversing a List

Step 2: Now, disconnect the first node from others. We will make sure that it points to none. As this node is going to be our last node. Perform operation `prevNode->next = NULL.`

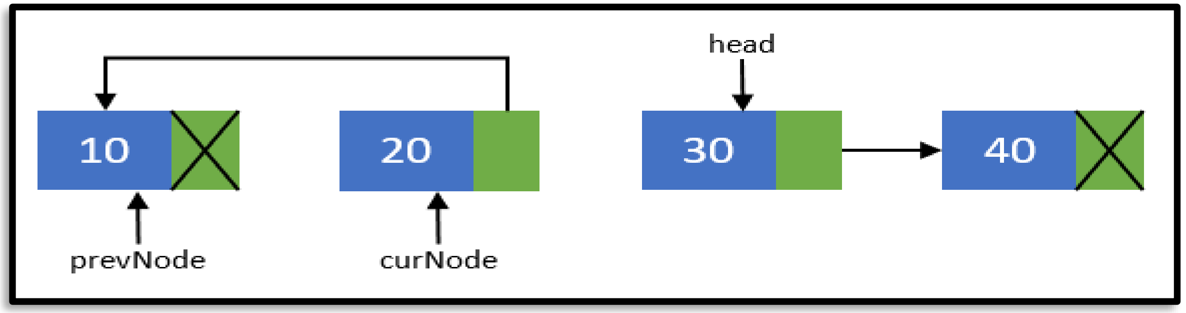


Step 3: Move the head node to its next node i.e. `head = head->next.`

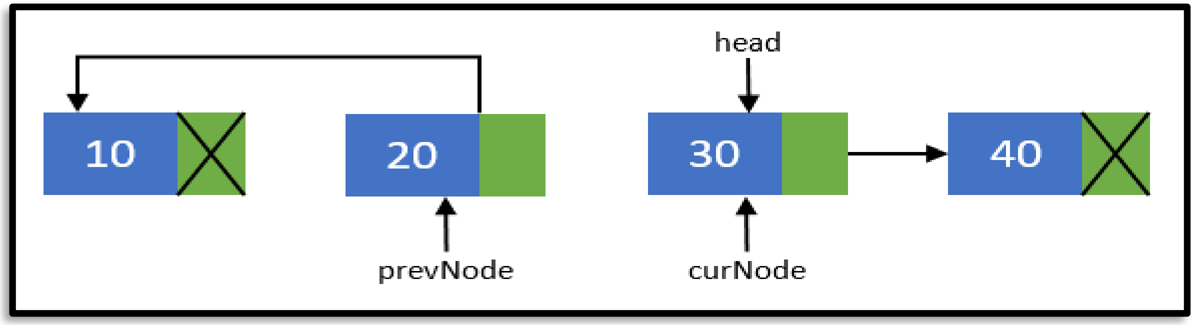


Reversing a List

Step 4: Now, re-connect the current node to its previous node i.e. `curNode->next = prevNode;`



Step 5: Point the previous node to current node and current node to head node. Means they should now point to `prevNode = curNode;` and `curNode = head.`

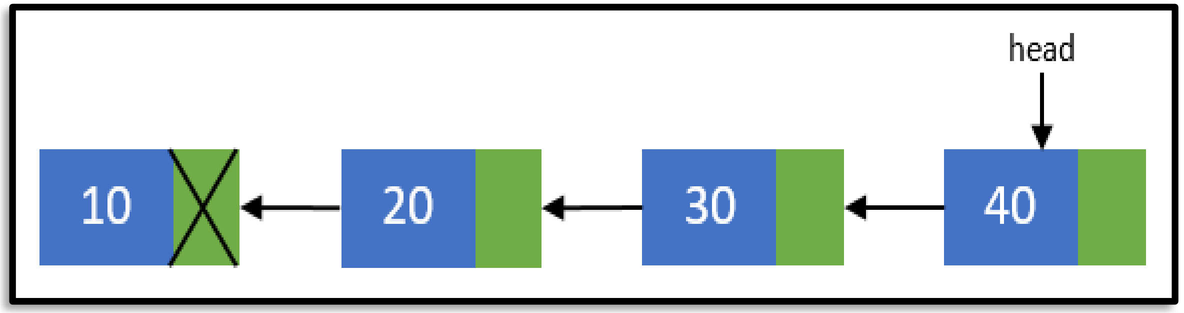


Reversing a List

Step 6: Repeat steps 3-5 till head pointer becomes NULL.

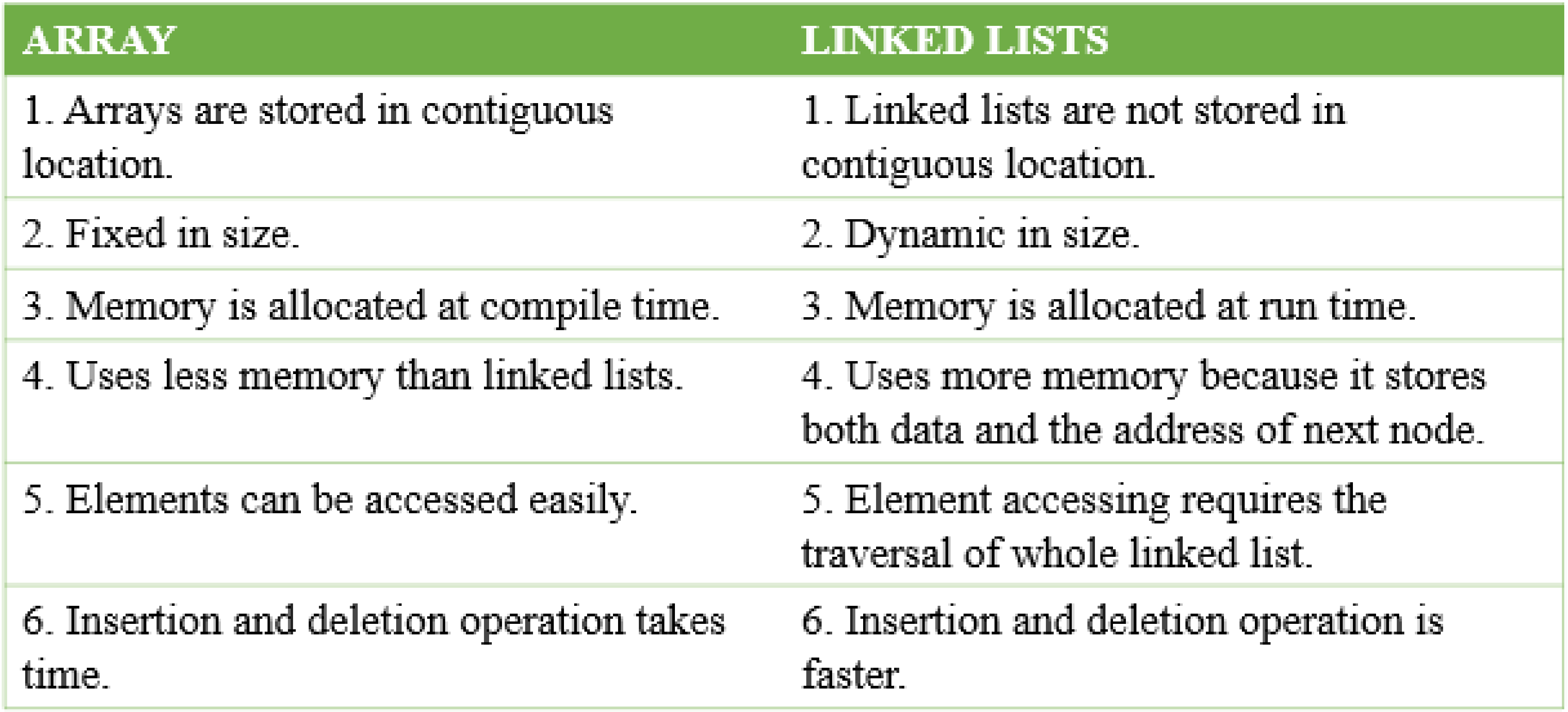
Step 7: Now, after all nodes has been re-connected in the reverse order. Make the last node as the first node. Means the head pointer should point to prevNode pointer.

- Perform `head = prevNode;` And finally you end up with a reversed linked list of its original.



Reversing a List

```
class LinkedList:
    # Function to initialize head
    def __init__(self):
        self.head = None

    # Function to reverse the linked list
    def reverse(self):
        prev = None
        current = self.head
        while(current is not None):
            next = current.next
            current.next = prev
            prev = current
            current = next
        self.head = prev

    # Function to insert a new node at the beginning
    def push(self, new_data):
        new_node = Node(new_data)
        new_node.next = self.head
        self.head = new_node

    # Utility function to print the LinkedList
    def printList(self):
        temp = self.head
        while(temp):
            print (temp.data,end=" ")
            temp = temp.next
```



ARRAY	LINKED LISTS
1. Arrays are stored in contiguous location.	1. Linked lists are not stored in contiguous location.
2. Fixed in size.	2. Dynamic in size.
3. Memory is allocated at compile time.	3. Memory is allocated at run time.
4. Uses less memory than linked lists.	4. Uses more memory because it stores both data and the address of next node.
5. Elements can be accessed easily.	5. Element accessing requires the traversal of whole linked list.
6. Insertion and deletion operation takes time.	6. Insertion and deletion operation is faster.

If you try to solve problems yourself, then you will learn many things automatically.

Spend few minutes and then enjoy the study.