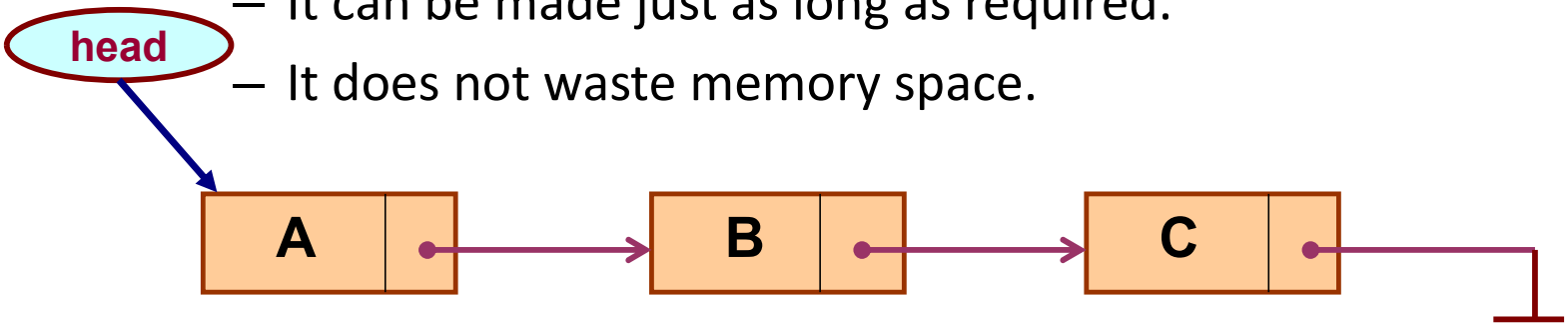
# Lecture

Basic Concepts of Linked Lists

# Introduction

- A linked list is a data structure which can change during execution.
  - Successive elements 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.



# Linked List with 3 Nodes- Implement

```
#include <bits/stdc++.h>
using namespace std;
class Node {
public:
   int data;
   Node* next;
};
int main()
{
   Node* head = NULL;
   Node* second = NULL;
   Node* third = NULL;

   // allocate 3 nodes in the heap
   head = new Node();
   second = new Node();
   third = new Node();
```

```
head->data = 1; // assign data in first node
   head->next = second; // Link first node with the second node


 second->data = 2; // assign data to second node
 second->next = third; // Link second node with the third node


 third->data = 3;
 third->next = null; /* data has been assigned to the data part of the
 third  block . And next pointer of the third block is made NULL to
 indicate that the linked list is terminated here.


 return 0;
 }
```

# Basic Functions

- Keeping track of a linked list:
  - Must know the pointer to the first element of the list (called *start*, *head*, etc.).

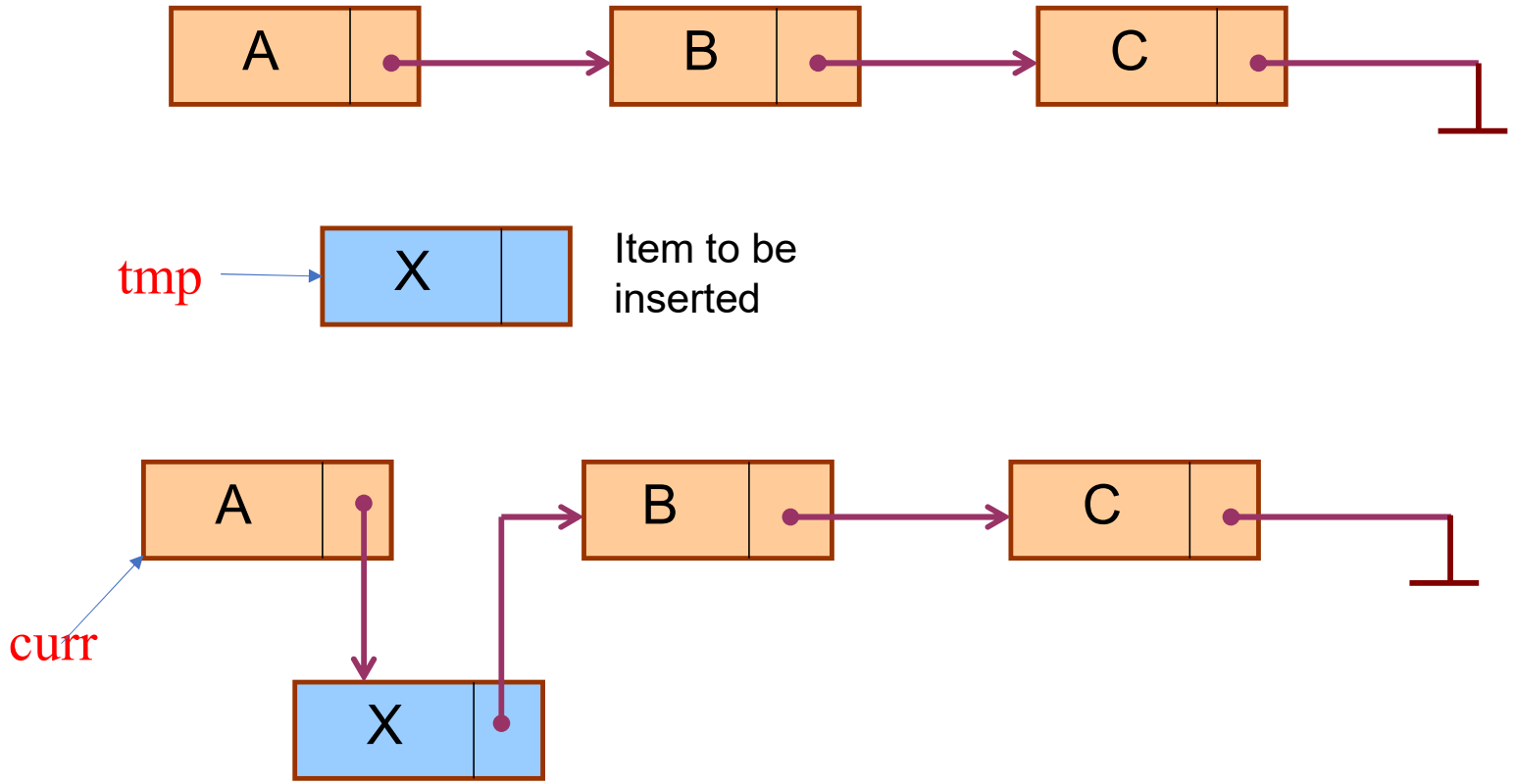
- Linked lists provide flexibility in allowing the items to be rearranged efficiently. Three basic functions:
  - Traversal.
  - Insert an element
  - Delete an element.

# Traversal

```
#include <bits/stdc++.h>
using namespace std;

class Node {
public:
        int data;
        Node* next;
};

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

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

        // allocate 3 nodes in the heap
        head = new Node();
        second = new Node();
        third = new Node();

        head->data = 1;
        head->next = second;

        second->data = 2;
        second->next = third;

        third->data = 3;
        third->next = NULL;

        printList(head);

        return 0;
}
```

# Illustration: Insertion

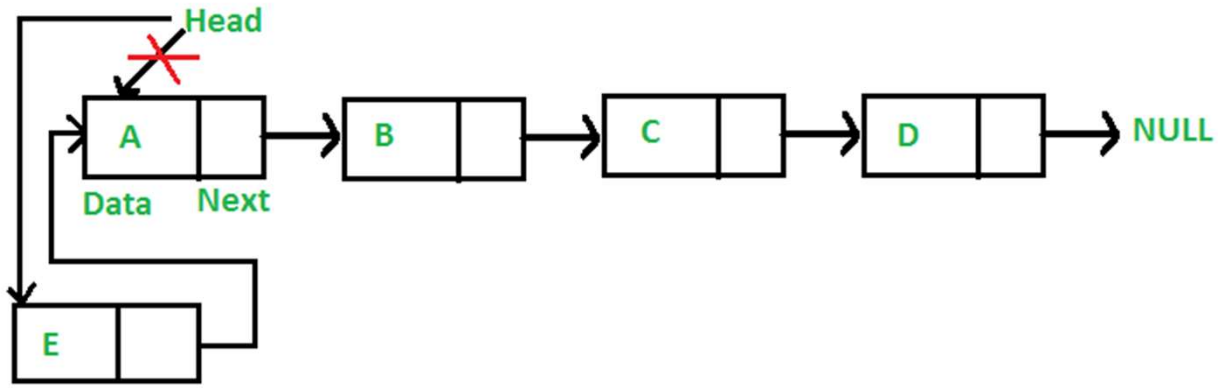


# Inserting a Node

- A node can be added in three ways

**1)** At the front of the linked list
**2)** After a given node.
**3)** At the end of the linked list..

# At the front of a link list




4 step process

1. allocate node

2. put in the data

3. Make next of new node to (previous) head

4. move the head to point to the new node

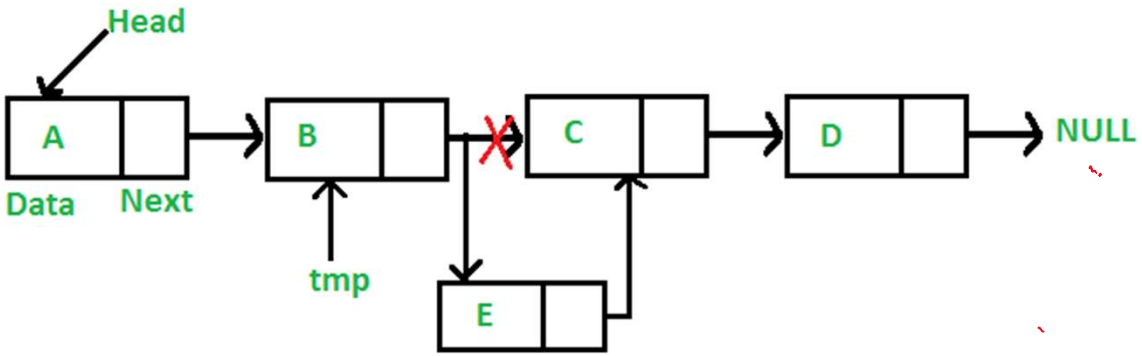
- /* Given a reference (pointer to pointer) to the head of a list and an int,
- inserts a new node on the front of the list. */

```
void push(Node** head_ref, int new_data)
{
        /* 1. allocate node */
        Node* new_node = new Node();

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

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

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

- Time complexity of push() is O(1) as it does constant amount of work.

# Add a node after a given node

- We are given pointer to a node, and the new node is inserted after the given node.



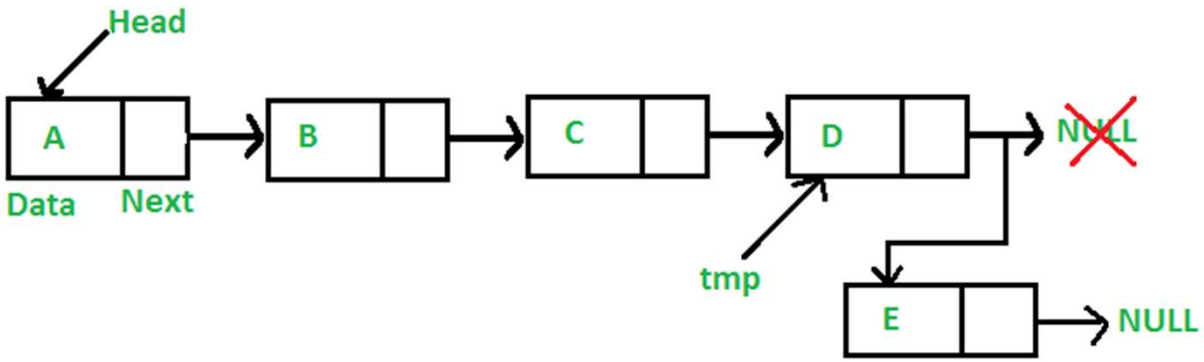

**(5 steps process)**

1. Check if the given prev_node is NULL
2. Allocate new node
3. Put in the data
4. Make next of new node as next of prev_node
5. move the next of prev_node as new_node

```
// Given a node prev_node, insert a new node after the given
// prev_node
void insertAfter(Node* prev_node, int new_data)
{

        // 1. Check if the given prev_node is NULL
        if (prev_node == NULL)
        {
        cout << "the given previous node cannot be NULL";
                return;
        }

        // 2. Allocate new node
        Node* new_node = new Node();

        // 3. Put in the data
        new_node->data = new_data;

// 4. Make next of new node as next of prev_node
        new_node->next = prev_node->next;

//5. move the next of prev_node  as new_node
        prev_node->next = new_node;
          }
```

Time complexity of insertAfter() is O(1) as it does constant amount of work.

# Add a node at the end



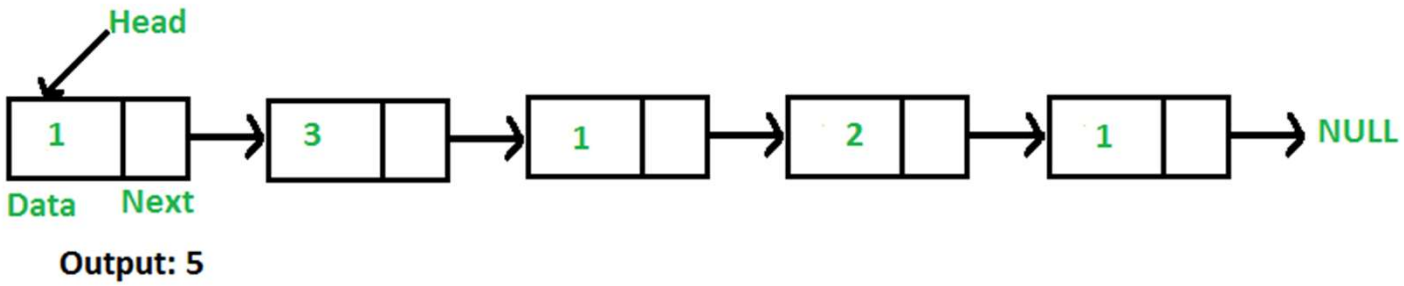

**(6 steps process)**

1. allocate node
2. Put in the data
3. This new node is going to be the last node, so make next of it as NULL
4. If the Linked List is empty, then make the new node as head
5. Else traverse till the last node
6. Change the next of last node

```
// Given a reference (pointer to pointer) to the head
// of a list and an int, appends a new node at the end
void append(Node** head_ref, int new_data)
{
        // 1. allocate node
        Node* new_node = new Node();

        // Used in step 5
        Node *last = *head_ref;

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

        // 3. This new node is going to be the last node, so make nex
of it as NULL
        new_node->next = NULL;

        //4. If the Linked List is empty, then make the new node as
head
        if (*head_ref == NULL)
        {
                *head_ref = new_node;
                return;
        }

        // 5. Else traverse till the last node
        while (last->next != NULL)
                last = last->next;

        // 6. Change the next of last node
        last->next = new_node;
        return;
}
```

Time complexity of append is O(n) where n is the number of nodes in linked list. Since there is a loop from head to end, the function does O(n) work.

# Count the number of nodes in a given singly linked list




**Output: 5**

For example, the function should return 5 for linked list 1->3->1->2->1.

**Iterative Solution**

1) Initialize count as 0

2) Initialize a node pointer, current = head.

3) Do following while current is not NULL
   a) current = current -> next
   b) count++;

4) Return count

```
#include <bits/stdc++.h>
using namespace std;
class Node
{
        public:
        int data;
        Node* next;
};

/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front of the list. */
void push(Node** head_ref, int new_data)
{
        /* allocate node */
        Node* new_node =new Node();

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

        /* link the old list off the new node */
        new_node->next = (*head_ref);
/* move the head to point to the new node */
        (*head_ref) = new_node;
}
```

```
/* Counts no. of nodes in linked list */
int getCount(Node* head)
{
        int count = 0; // Initialize count
        Node* current = head; // Initialize current
        while (current != NULL)
        {
                count++;
                current = current->next;
        }
        return count;
}
int main()
{       Node* head = NULL;
/* Use push() to construct below list 1->2->1->3->1 */
        push(&head, 1);
        push(&head, 3);
        push(&head, 1);
        push(&head, 2);
        push(&head, 1);

        /* Check the count function */
        cout<<"count of nodes is "<< getCount(head);
        return 0; }
```

# Search an element in a Linked List

Write a function that searches a given key ‘x’ in a given singly linked list. The function should return true if x is present in linked list and false otherwise.

bool search(Node *head, int x)

For example, if the key to be searched is 15 and linked list is 14->21->11->30->10, then function should return false.

If key to be searched is 14, then the function should return true.

1) Initialize a node pointer, current = head.

2) Do following while current is not NULL
   a) current->key is equal to the key being searched return true.
   b) current = current->next

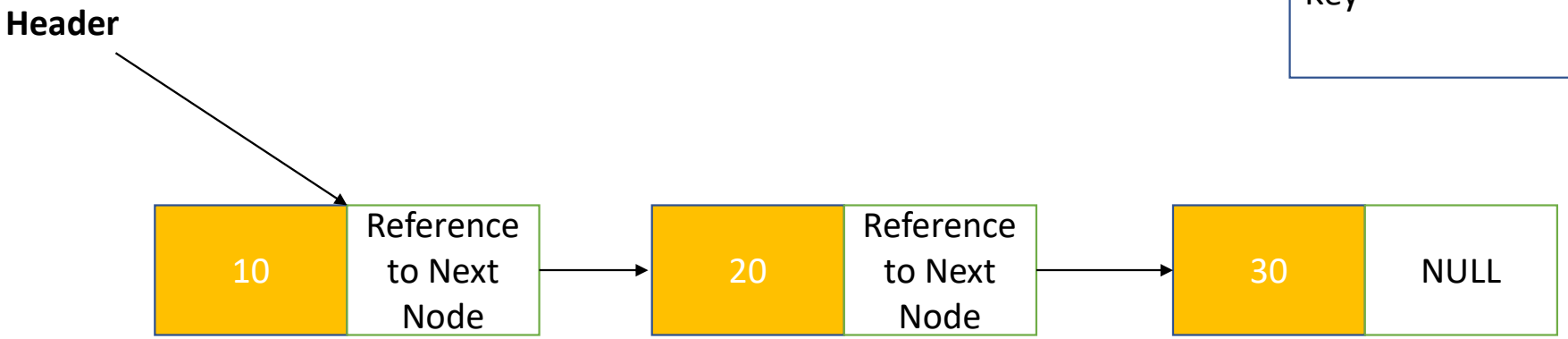
3) Return false

```
#include <bits/stdc++.h>
using namespace std;
class Node
{
        public:
        int key;
        Node* next;
};
/* Given a reference (pointer to pointer) to the head
of a list and an int, push a new node on the front
of the list. */
void push(Node** head_ref, int new_key)
{
        /* allocate node */
        Node* new_node = new Node();

        /* put in the key */
        new_node->key = new_key;

        /* link the old list off the new node */
        new_node->next = (*head_ref);

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

```
/* Checks whether the value x is present in linked list */
bool search(Node* head, int x)
{
        Node* current = head; // Initialize current
        while (current != NULL)
        {
                if (current->key == x)
                        return true;
                current = current->next; }
        return false;
}
int main()
{
        /* Start with the empty list */
        Node* head = NULL;
        int x = 21;
/* Use push() to construct below list 14->21->11->30->10 */
        push(&head, 10);
        push(&head, 30);
        push(&head, 11);
        push(&head, 21);
        push(&head, 14);

        search(head, 21)? cout<<"Yes" : cout<<"No";
        return 0; }
```

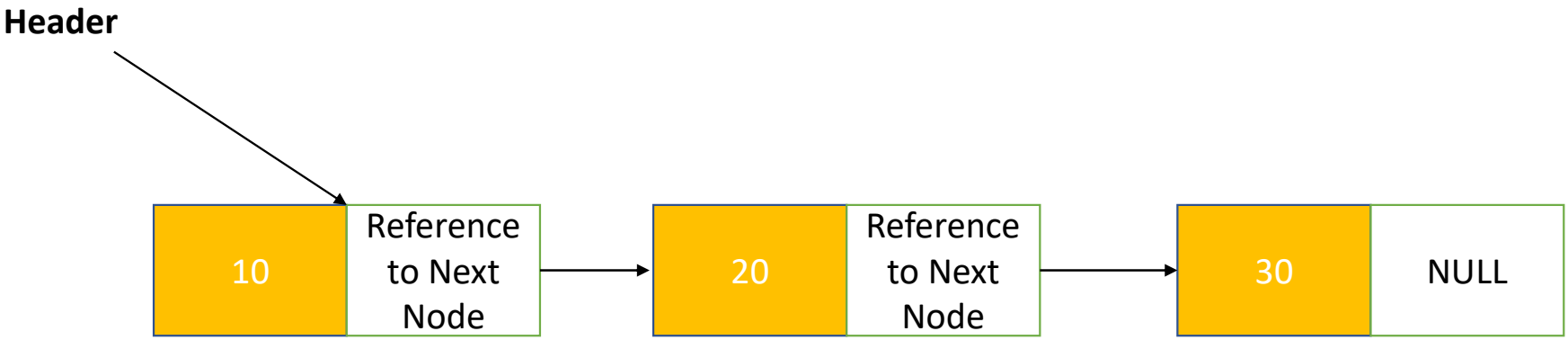
# Deletion of node in a linked list

Deletion of a node by value of Key

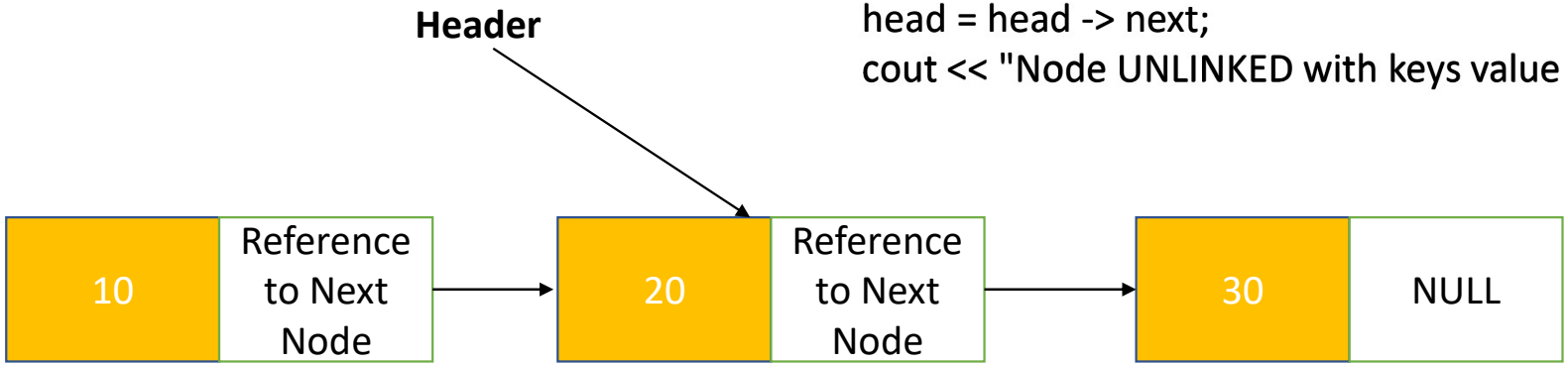


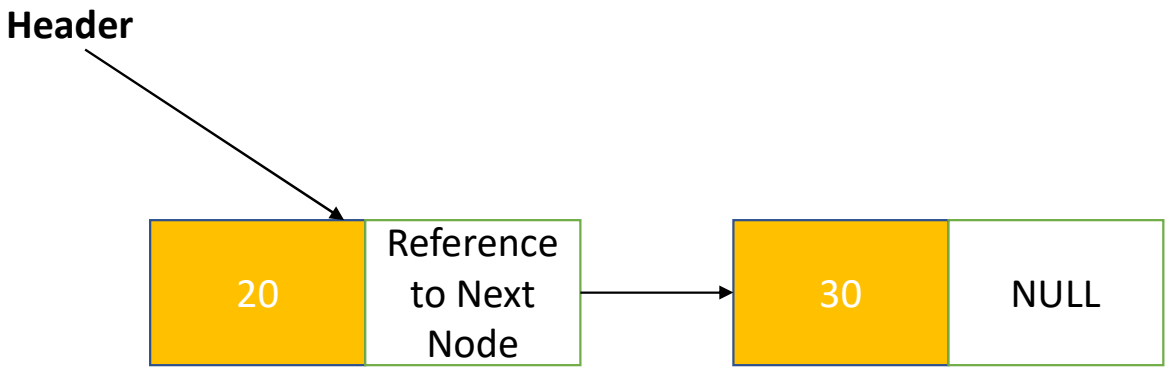

Suppose key value is 10 which means delete first node.

# Deletion of first node in a linked list

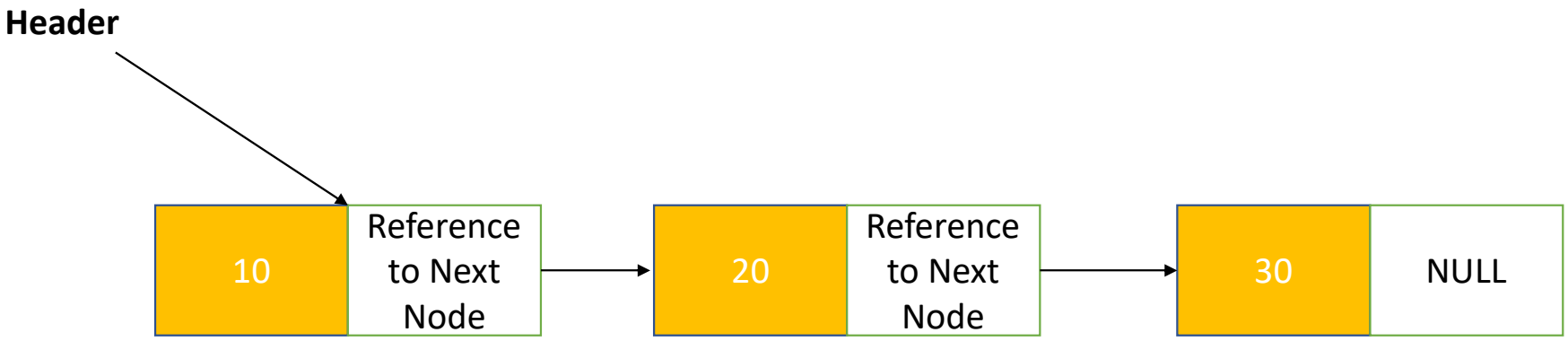


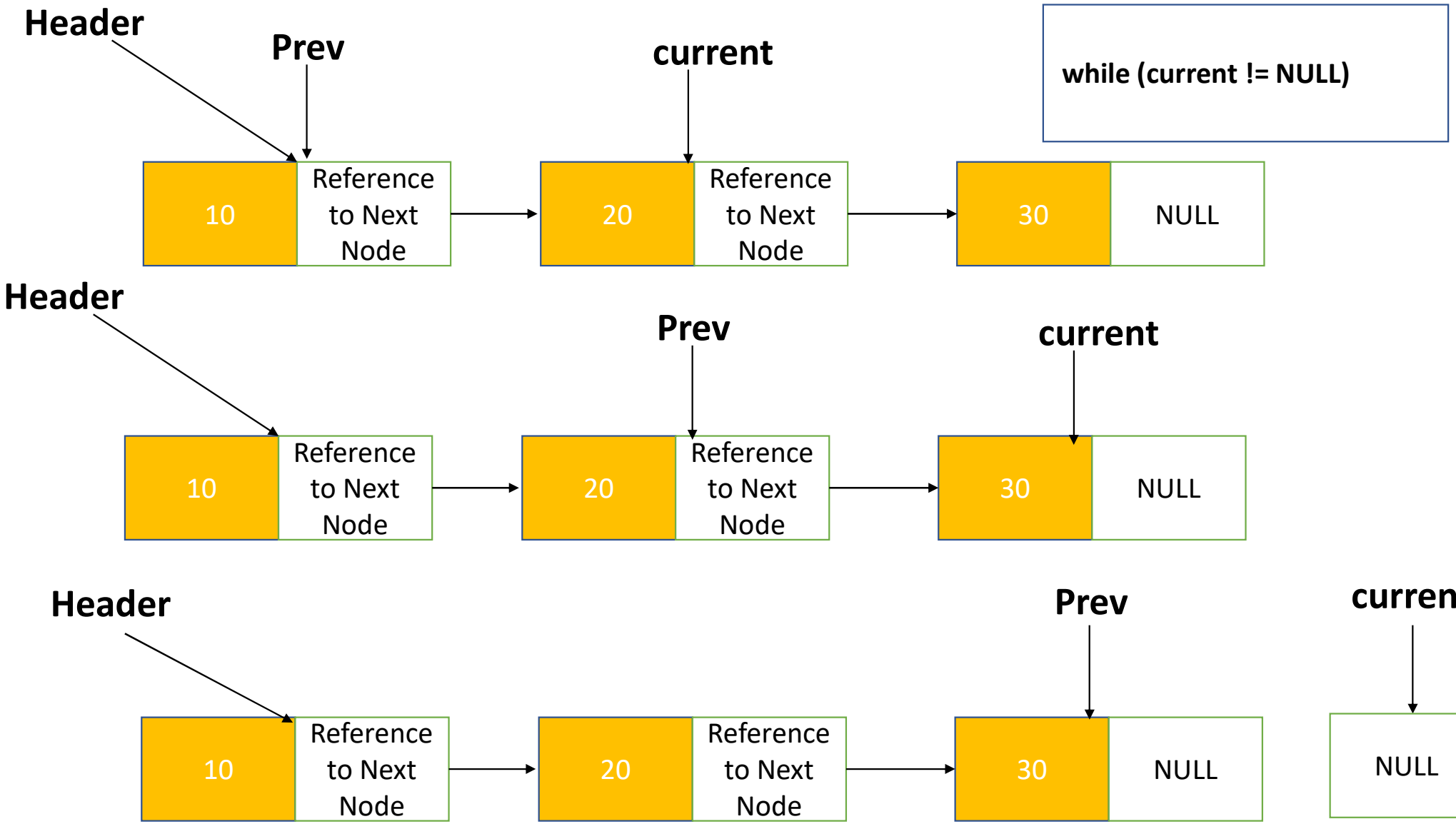

```
if (head != NULL) {
    if (head -> data == k) {
      head = head -> next;
      cout << "Node UNLINKED with keys value : " << k << endl; }
```




Header
20
Reference to Next Node
30
NULL

# Deletion of node in a linked list with data value k (not first node)

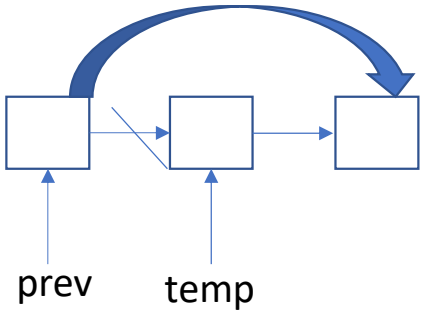



Header
Prev
current
while (current != NULL)
10
Reference to Next Node
20
Reference to Next Node
30
NULL
Header
Prev
current
10
Reference to Next Node
20
Reference to Next Node
30
NULL
Header
Prev
current
10
Reference to Next Node
20
Reference to Next Node
30
NULL
NULL

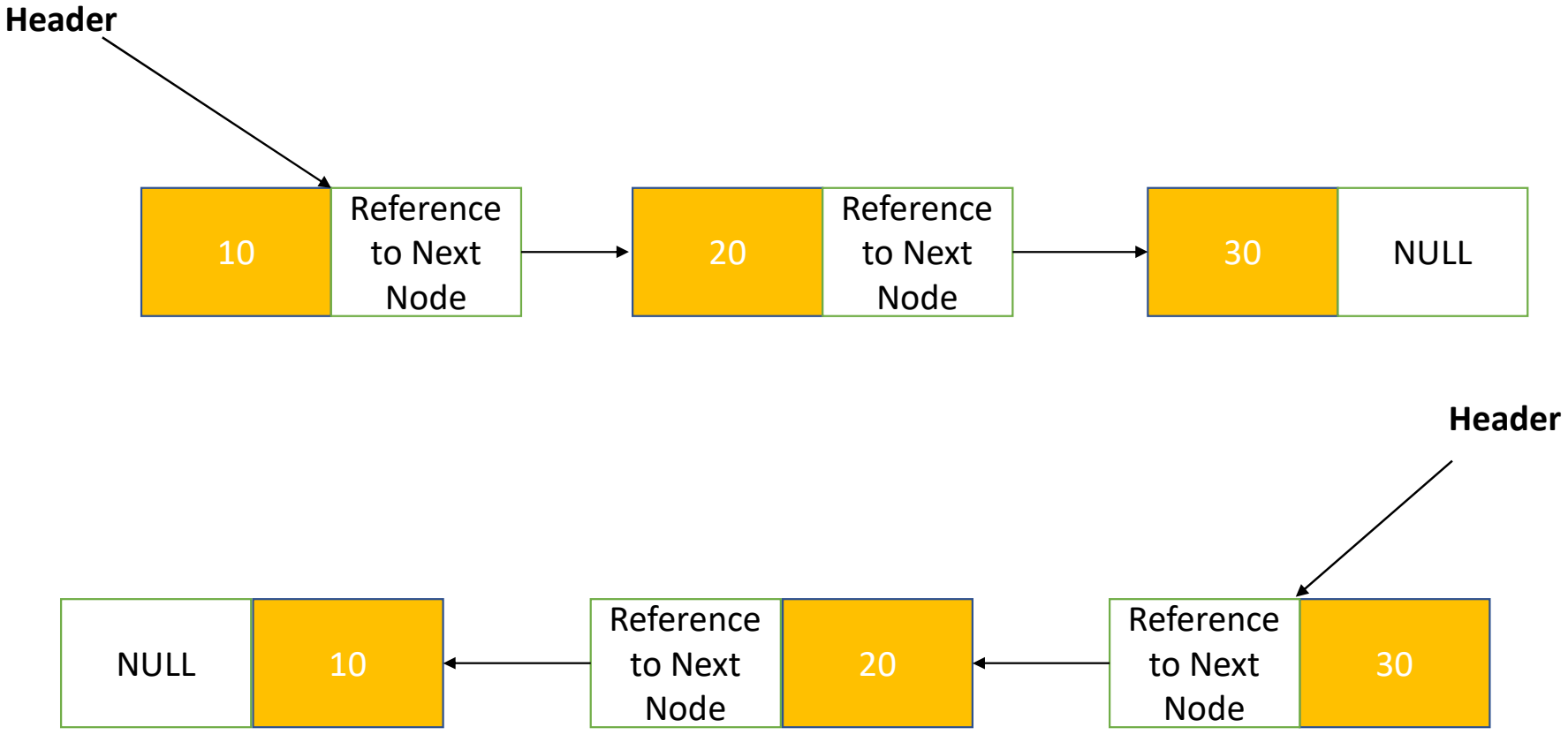
```
Node * temp = NULL;
      Node * prev = head;
      Node * current = head -> next;
      while (current != NULL) {
       if (current -> data == x) {
         temp = current;
         current = NULL;
       }
        else {
         prev = prev -> next;
         current = current -> next;
               }
      }
```

```
temp  = current;
```

```
if (temp != NULL)
      {
  prev -> next = temp -> next;
 cout << "Node detached with data value : " << x<<"\n";
       }
else
     {
cout << "A Node is not found with key value : " << x <<"\n";
     }
```



# Reverse


Header
10
Reference to Next Node
20
Reference to Next Node
30
NULL
Header
NULL
10
Reference to Next Node
20
Reference to Next Node
30

# Solution

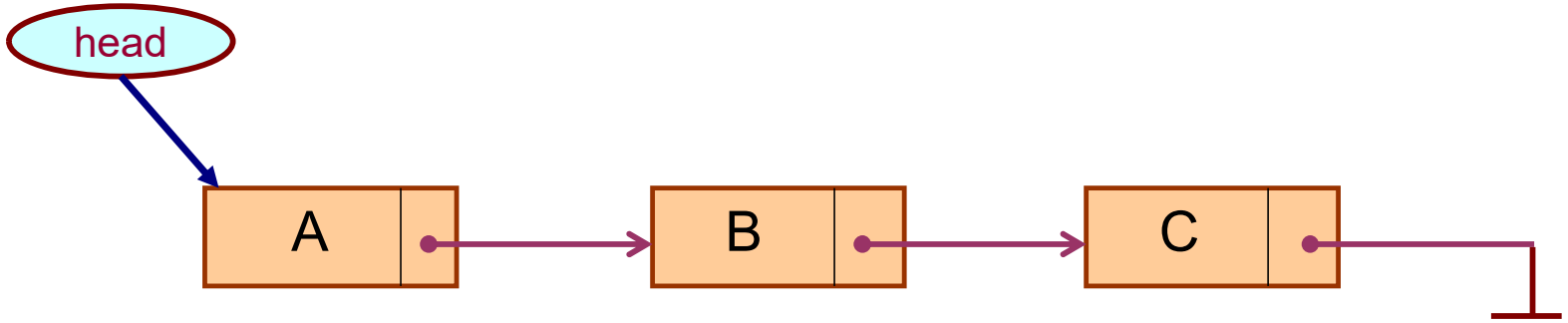
```
while(current!=NULL)
{

temp    =   current->next;
current->next  =  prev;
prev     =    current;
current =    temp;
}
head = prev;
```

# Operations on Linked List

- create_list { 16, 8, 10, 2, 34, 20, 12}
- Insert_front { 55, 16, 8, 10, 2, 34, 20, 12}
- Insert_rear { 16, 8, 10, 2, 34, 20, 12, 55}
- Delete_front { 8, 10, 2, 34, 20, 12}
- Delete_rear { 16, 8, 10, 2, 34, 20 }
- Delete_kth { 16, 8, 10, 2, 20, 12}
- Print_list 16-8-10-2-34-20-12
- Count_list 7
- check_if_empty No
- Clear_list { }

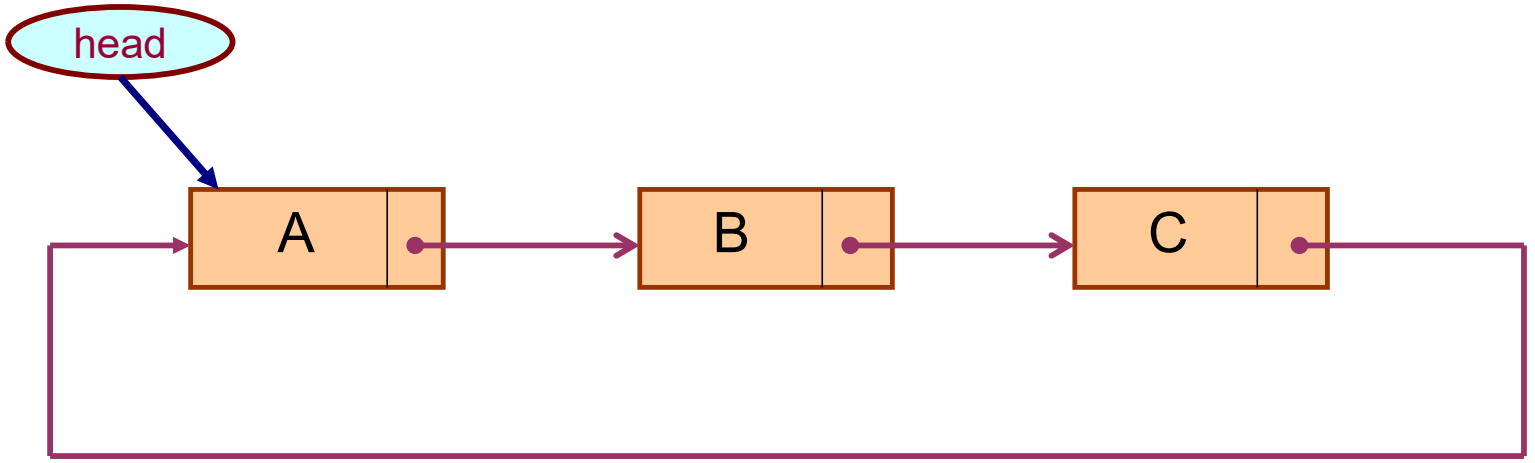
# Types of Lists

- Depending on the way in which the links are used to maintain adjacency, several different types of linked lists are possible.
  - Linear singly-linked list (or simply linear list)
    - One we have discussed so far.



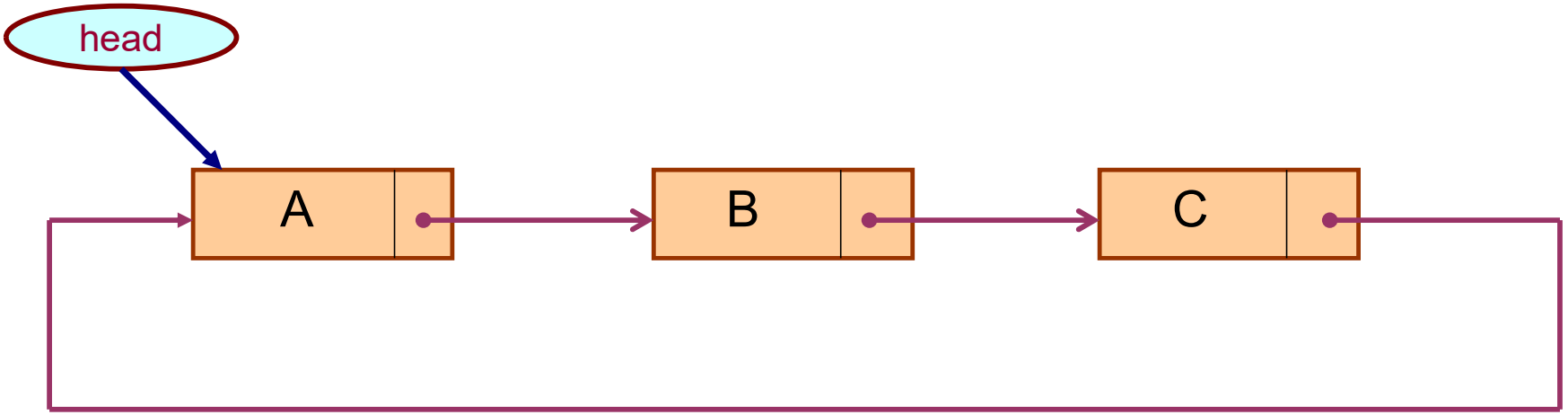
# Circular linked list

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



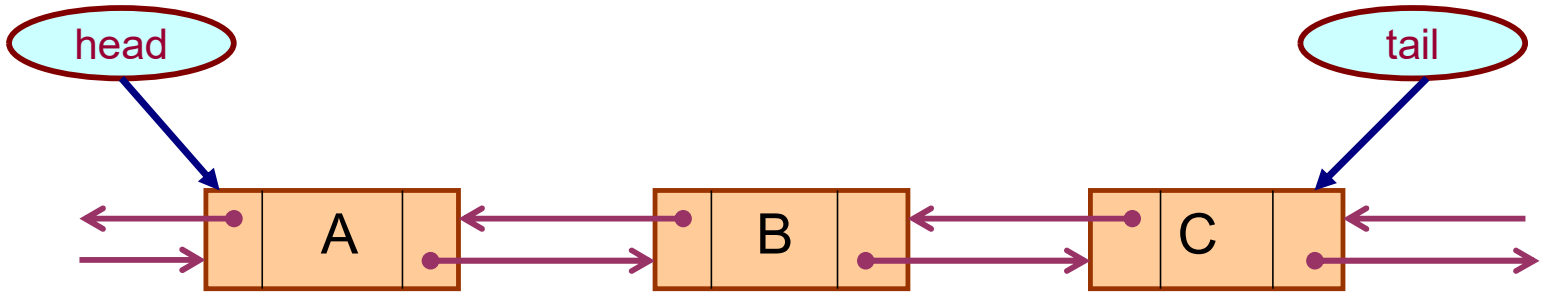
# Circular linked list

- Any node can be a starting point. We can traverse the whole list by starting from any point. We just need to stop when the first visited node is visited again.
- It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in between nodes.



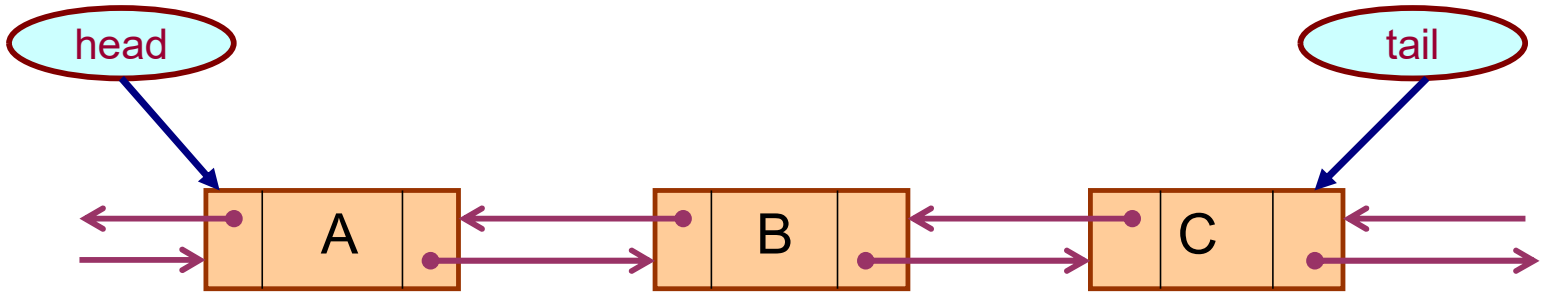
# Doubly linked list

- 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*.



# Doubly linked list

- A doubly linked list (DLL) can be traversed in both forward and backward direction.
- The delete operation in DLL is more efficient if pointer to the node to be deleted is given.
- We can quickly insert a new node before a given node.



# Your Task

- Join 2 linked list