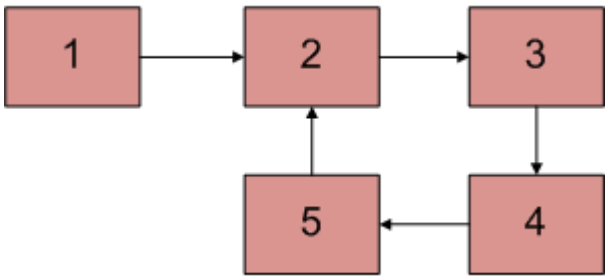
# 19.Write a C function to detect loop in a linked list

```
int detectloop(struct node *list)
{
  struct node  *slow_p = list, *fast_p = list;

  while(slow_p && fast_p &&
          fast_p->next )
  {
    slow_p = slow_p->next;
    fast_p  = fast_p->next->next;
    if (slow_p == fast_p)
    {
       printf("Found Loop");
       return 1;
    }
  }
  return 0;
}
```

# 20.Detect and Remove Loop in a Linked List

Write a function *detectAndRemoveLoop()* that checks whether a given Linked List contains loop and if loop is present then removes the loop and returns true. And if the list doesn't contain loop then returns false. Below diagram shows a linked list with a loop. *detectAndRemoveLoop()* must change the below list to 1->2->3->4->5->NULL.




To remove loop, all we need to do is to get pointer to the last node of the loop. For example, node with value 5 in the above diagram. Once we have pointer to the last node, we can make the next of this node as NULL and loop is gone.

We can easily use Hashing or Visited node techniques (discussed in the aobve mentioned post) to get the pointer to the last node. Idea is simple: the very first node whose next is already visited (or hashed) is the last node.

We can also use Floyd Cycle Detection algorithm to detect and remove the loop. In the Floyd's algo, the slow and fast pointers meet at a loop node. We can use this loop node to remove cycle. There are following two different ways of removing loop when Floyd's algorithm is used for Loop detection.

**Method 1 (Check one by one)**

We know that Floyd's Cycle detection algorithm terminates when fast and slow pointers meet at a common point. We also know that this common point is one of the loop nodes (2 or 3 or 4 or 5 in the above diagram). We store the address of this in a pointer variable say ptr2. Then we start from the head of the Linked List and check for nodes one by one if they are reachable from ptr2. When we find a node that is reachable, we know that this node is the starting node of the loop in Linked List and we can get pointer to the previous of this node.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. Used by detectAndRemoveLoop() */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
   struct node *ptr1;
   struct node *ptr2;

   /* Set a pointer to the beging of the Linked List and
```

```
      move it one by one to find the first node which is
      part of the Linked List */
   ptr1 = head;
   while(1)
   {
     /* Now start a pointer from loop_node and check if it ever
       reaches ptr2 */
     ptr2 = loop_node;
     while(ptr2->next != loop_node && ptr2->next != ptr1)
     {
         ptr2 = ptr2->next;
     }

     /* If ptr2 reahced ptr1 then there is a loop. So break the
        loop */
     if(ptr2->next == ptr1)
        break;

     /* If ptr2 did't reach ptr1 then try the next node after ptr1 */
     else
       ptr1 = ptr1->next;
   }

   /* After the end of loop ptr2 is the last node of the loop. So
     make next of ptr2 as NULL */
   ptr2->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct 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;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main()
{
```

```
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

**Method 2 (Efficient Solution)**

This method is also dependent on Floyd's Cycle detection algorithm.

1) Detect Loop using Floyd's Cycle detection algo and get the pointer to a loop node.

2) Count the number of nodes in loop. Let the count be k.

3) Fix one pointer to the head and another to kth node from head.

4) Move both pointers at the same pace, they will meet at loop starting node.

5) Get pointer to the last node of loop and make next of it as NULL.

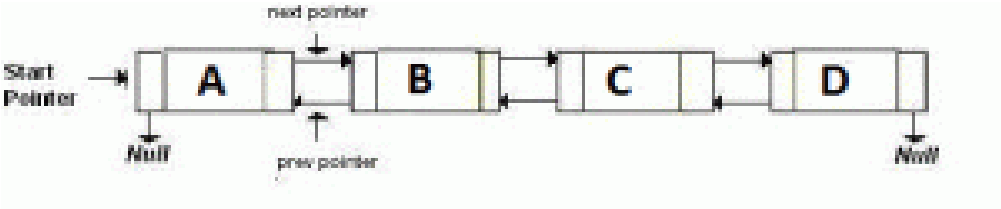
```
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Function to remove loop. */
void removeLoop(struct node *, struct node *);

/* This function detects and removes loop in the list
  If loop was there in the list then it returns 1,
  otherwise returns 0 */
int detectAndRemoveLoop(struct node *list)
{
    struct node  *slow_p = list, *fast_p = list;

    while (slow_p && fast_p && fast_p->next)
    {
        slow_p = slow_p->next;
        fast_p  = fast_p->next->next;

        /* If slow_p and fast_p meet at some point then there
           is a loop */
```

```
        if (slow_p == fast_p)
        {
            removeLoop(slow_p, list);

            /* Return 1 to indicate that loop is found */
            return 1;
        }
    }

    /* Return 0 to indeciate that ther is no loop*/
    return 0;
}

/* Function to remove loop.
 loop_node --> Pointer to one of the loop nodes
 head -->  Pointer to the start node of the linked list */
void removeLoop(struct node *loop_node, struct node *head)
{
    struct node *ptr1 = loop_node;
    struct node *ptr2 = loop_node;

    // Count the number of nodes in loop
    unsigned int k = 1, i;
    while (ptr1->next != ptr2)
    {
        ptr1 = ptr1->next;
        k++;
    }

    // Fix one pointer to head
    ptr1 = head;

    // And the other pointer to k nodes after head
    ptr2 = head;
    for(i = 0; i < k; i++)
      ptr2 = ptr2->next;

    /*  Move both pointers at the same pace,
      they will meet at loop starting node */
    while(ptr2 != ptr1)
    {
        ptr1 = ptr1->next;
        ptr2 = ptr2->next;
    }

    // Get pointer to the last node
    ptr2 = ptr2->next;
    while(ptr2->next != ptr1)
       ptr2 = ptr2->next;

    /* Set the next node of the loop ending node
      to fix the loop */
    ptr2->next = NULL;
}

/* UTILITY FUNCTIONS */
/* Given a reference (pointer to pointer) to the head
  of a list and an int, pushes a new node on the front
  of the list. */
void push(struct node** head_ref, int new_data)
{
```

```
    /* allocate node */
    struct node* new_node =
        (struct node*) malloc(sizeof(struct 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;
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

    push(&head, 10);
    push(&head, 4);
    push(&head, 15);
    push(&head, 20);
    push(&head, 50);

    /* Create a loop for testing */
    head->next->next->next->next->next = head->next->next;

    detectAndRemoveLoop(head);

    printf("Linked List after removing loop \n");
    printList(head);

    getchar();
    return 0;
}
```

# 21.XOR Linked List – A Memory Efficient Doubly Linked List

An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes. A memory efficient version of Doubly Linked List can be created using only one space for address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient as the list uses bitwise XOR operation to save space for one address. In the XOR linked list, instead of storing actual memory addresses, every node stores the XOR of addresses of previous and next nodes.




Consider the above Doubly Linked List. Following are the Ordinary and XOR (or Memory Effiecient) representations of the Doubly Linked List.

**Ordinary Representation:**

Node A:

prev = NULL, next = add(B) // previous is NULL and next is address of B

Node B:

prev = add(A), next = add(C) // previous is address of A and next is address of C

Node C:

prev = add(B), next = add(D) // previous is address of B and next is address of D

Node D:

prev = add(C), next = NULL // previous is address of C and next is NULL

**XOR List Representation:**

Let us call the address variable in XOR representation npx (XOR of next and previous)

Node A:

npx = 0 XOR add(B) // bitwise XOR of zero and address of B

Node B:

npx = add(A) XOR add(C) // bitwise XOR of address of A and address of C

Node C:

npx = add(B) XOR add(D) // bitwise XOR of address of B and address of D

Node D:

npx = add(C) XOR 0 // bitwise XOR of address of C and 0

**Traversal of XOR Linked List:**

We can traverse the XOR list in both forward and reverse direction. While traversing the list we need to remember the address of the previously accessed node in order to calculate the next node's address. For example when we are at node C, we must have address of B. XOR of add(B) and *npx* of C gives us the add(D). The reason is simple: npx(C) is "add(B) XOR add(D)". If we do xor of npx(C) with add(B), we get the result as "add(B) XOR add(D) XOR add(B)" which is "add(D) XOR 0" which is

“add(D)”. So we have the address of next node. Similarly we can traverse the list in backward direction

# 22.Segregate even and odd nodes in a Linked List

Given a Linked List of integers, write a function to modify the linked list such that all even numbers appear before all the odd numbers in the modified linked list. Also, keep the order of even and odd numbers same.

Examples:
Input: 17->15->8->12->10->5->4->1->7->6->NULL
Output: 8->12->10->4->6->17->15->5->1->7->NULL

Input: 8->12->10->5->4->1->6->NULL
Output: 8->12->10->4->6->5->1->NULL

// If all numbers are even then do not change the list
Input: 8->12->10->NULL
Output: 8->12->10->NULL

// If all numbers are odd then do not change the list
Input: 1->3->5->7->NULL
Output: 1->3->5->7->NULL

**Method 1**
The idea is to get pointer to the last node of list. And then traverse the list starting from the head node and move the odd valued nodes from their current position to end of the list.

Algorithm:
…1) Get pointer to the last node.
…2) Move all the odd nodes to the end.
……..a) Consider all odd nodes before the first even node and move them to end.
……..b) Change the head pointer to point to the first even node.
……..b) Consider all odd nodes after the first even node and move them to the end.

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

/* a node of the singly linked list */
struct node
{
  int data;
  struct node *next;
};
```

```
void segregateEvenOdd(struct node **head_ref)
{
  struct node *end = *head_ref;

  struct node *prev = NULL;
  struct node *curr = *head_ref;

  /* Get pointer to the last node */
  while(end->next != NULL)
       end = end->next;

  struct node *new_end = end;

  /* Consider all odd nodes before the first even node
     and move then after end */
  while(curr->data %2 != 0 && curr != end)
  {
    new_end->next = curr;
    curr = curr->next;
    new_end->next->next = NULL;
    new_end = new_end->next;
  }

  // 10->8->17->17->15
  /* Do following steps only if there is any even node */
  if (curr->data%2 == 0)
  {
    /* Change the head pointer to point to first even node */
    *head_ref = curr;

    /* now current points to the first even node */
    while(curr != end)
    {
      if ( (curr->data)%2 == 0 )
      {
         prev = curr;
         curr = curr->next;
      }
      else
      {
         /* break the link between prev and current */
         prev->next = curr->next;

          /* Make next of curr as NULL  */
         curr->next = NULL;

         /* Move curr to end */
         new_end->next = curr;

         /* make curr as new end of list */
         new_end = curr;

         /* Update current pointer to next of the moved node */
         curr = prev->next;
      }
    }
  }

  /* We must have prev set before executing lines following this
     statement */
  else
```

```
    prev = curr;

  /* If the end of the original list is odd then move this node to
    end to maintain same order of odd numbers in modified list */
  if((end->data)%2 != 0)
  {
      prev->next = end->next;
      end->next = NULL;
      new_end->next = end;
  }
  return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
       (struct node*) malloc(sizeof(struct 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;
}

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

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

  /* Let us create a sample linked list as following
    17->15->8->12->10->5->4->1->7->6 */
  push(&head, 6);
  push(&head, 7);
  push(&head, 1);
  push(&head, 4);
  push(&head, 5);
  push(&head, 10);
  push(&head, 12);
  push(&head, 8);
  push(&head, 15);
  push(&head, 17);
```

```
  printf("\n Original Linked list ");
  printList(head);

  segregateEvenOdd(&head);

  printf("\n Modified Linked list ");
  printList(head);

  getchar();
  return 0;
}
```

Output:

```
Original Linked list 17 15 8 12 10 5 4 1 7 6
Modified Linked list 8 12 10 4 6 17 15 5 1 7
```

Time complexity: O(n)

**Method 2**

The idea is to split the linked list into two: one containing all even nodes and other containing all odd nodes. And finally attach the odd node linked list after the even node linked list.

To split the Linked List, traverse the original Linked List and move all odd nodes to a separate Linked List of all odd nodes. At the end of loop, the original list will have all the even nodes and the odd node list will have all the odd nodes. To keep the ordering of all nodes same, we must insert all the odd nodes at the end of the odd node list. And to do that in constant time, we must keep track of last pointer in the odd node list.

Time complexity: O(n)

# 23.Delete nodes which have a greater value on right side

Given a singly linked list, remove all the nodes which have a greater value on right side.

Examples:

**a)** The list 12->15->10->11->5->6->2->3->NULL should be changed to 15->11->6->3->NULL. Note that 12, 10, 5 and 2 have been deleted because there is a greater value on the right side.

When we examine 12, we see that after 12 there is one node with value greater than 12 (i.e. 15), so we delete 12.

When we examine 15, we find no node after 15 that has value greater than 15 so we keep this node.

When we go like this, we get 15->6->3

**b)** The list 10->20->30->40->50->60->NULL should be changed to 60->NULL. Note that 10, 20, 30, 40 and 50 have been deleted because they all have a greater value on the right side.

**c)** The list 60->50->40->30->20->10->NULL should not be changed.

**Method 1 (Simple)**

Use two loops. In the outer loop, pick nodes of the linked list one by one. In the inner loop, check if there exist a node whose value is greater than the picked node. If there exists a node whose value is greater, then delete the picked node.

Time Complexity: O(n^2)

**Method 2 (Use Reverse)**

1. Reverse the list.

2. Traverse the reversed list. Keep max till now. If next node < max, then delete the next node, otherwise max = next node.

3. Reverse the list again to retain the original order.

Time Complexity: O(n)

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

/* structure of a linked list node */
struct node
{
     int data;
     struct node *next;
};

/* prototype for utility functions */
void reverseList(struct node **headref);
void _delLesserNodes(struct node *head);

/* Deletes nodes which have a node with greater value node
  on left side */
void delLesserNodes(struct node **head_ref)
{
    /* 1) Reverse the linked list */
    reverseList(head_ref);

    /* 2) In the reversed list, delete nodes which have a node
       with greater value node on left side. Note that head
       node is never deleted because it is the leftmost node.*/
    _delLesserNodes(*head_ref);

    /* 3) Reverse the linked list again to retain the
       original order */
    reverseList(head_ref);
}

/* Deletes nodes which have greater value node(s) on left side */
void _delLesserNodes(struct node *head)
{
     struct node *current = head;

     /* Initialize max */
     struct node *maxnode = head;
```

```
    struct node *temp;

    while (current != NULL && current->next != NULL)
    {
        /* If current is smaller than max, then delete current */
        if(current->next->data < maxnode->data)
        {
            temp = current->next;
            current->next = temp->next;
            free(temp);
        }

        /* If current is greater than max, then update max and
           move current */
        else
        {
            current = current->next;
            maxnode = current;
        }

    }
}

/* Utility function to insert a node at the begining */
void push(struct node **head_ref, int new_data)
{
    struct node *new_node =
              (struct node *)malloc(sizeof(struct node));
    new_node->data = new_data;
    new_node->next = *head_ref;
    *head_ref = new_node;
}

/* Utility function to reverse a linked list */
void reverseList(struct node **headref)
{
    struct node *current = *headref;
    struct node *prev = NULL;
    struct node *next;
    while(current != NULL)
    {
         next = current->next;
         current->next = prev;
         prev = current;
         current = next;
    }
    *headref = prev;
}

/* Utility function to print a linked list */
void printList(struct node *head)
{
    while(head!=NULL)
    {
       printf("%d ",head->data);
       head=head->next;
    }
    printf("\n");
}

/* Driver program to test above functions */
int main()
```

```
{
    struct node *head = NULL;

    /* Create following linked list
      12->15->10->11->5->6->2->3 */
    push(&head,3);
    push(&head,2);
    push(&head,6);
    push(&head,5);
    push(&head,11);
    push(&head,10);
    push(&head,15);
    push(&head,12);

    printf("Given Linked List: ");
    printList(head);

    delLesserNodes(&head);

    printf("\nModified Linked List: ");
    printList(head);

    getchar();
    return 0;
}
```

Output:

```
Given Linked List: 12 15 10 11 5 6 2 3
Modified Linked List: 15 11 6 3
```

# 24.Reverse alternate K nodes in a Singly Linked List

Given a linked list, write a function to reverse every alternate k nodes (where k is an input to the function) in an efficient way. Give the complexity of your algorithm.

Example:
Inputs: 1->2->3->4->5->6->7->8->9->NULL and k = 3
Output: 3->2->1->4->5->6->9->8->7->NULL.

**Method 1 (Process 2k nodes and recursively call for rest of the list)**
This method is basically an extension of the method discussed in this post.

kAltReverse(struct node *head, int k)

1) Reverse first k nodes.
2) In the modified list head points to the kth node. So change next of head to (k+1)th node
3) Move the current pointer to skip next k nodes.

4) Call the kAltReverse() recursively for rest of the n - 2k nodes.

5) Return new head of the list.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses alternate k nodes and
   returns the pointer to the new head node */
struct node *kAltReverse(struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;

    /*1) reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* 2) Now head points to the kth node.  So change next
       of head to (k+1)th node*/
    if(head != NULL)
      head->next = current;

    /* 3) We do not want to reverse next k nodes. So move the current
        pointer to skip next k nodes */
    count = 0;
    while(count < k-1 && current != NULL )
    {
      current = current->next;
      count++;
    }

    /* 4) Recursively call for the list starting from current->next.
       And make rest of the list as next of first node */
    if(current !=  NULL)
       current->next = kAltReverse(current->next, k);

    /* 5) prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
```

```
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

    // create a list 1->2->3->4->5...... ->20
    for(int i = 20; i > 0; i--)
      push(&head, i);

     printf("\n Given linked list \n");
     printList(head);
     head = kAltReverse(head, 3);

     printf("\n Modified Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

Output:

*Given linked list*

*1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20*

*Modified Linked list*

*3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19*

Time Complexity: O(n)

**Method 2 (Process k nodes and recursively call for rest of the list)**

The method 1 reverses the first k node and then moves the pointer to k nodes ahead. So method 1 uses two while loops and processes 2k nodes in one recursive call.

This method processes only k nodes in a recursive call. It uses a third bool parameter b which decides whether to reverse the k elements or simply move the pointer.

_kAltReverse(struct node *head, int k, bool b)

1) If b is true, then reverse first k nodes.
2) If b is false, then move the pointer k nodes ahead.
3) Call the kAltReverse() recursively for rest of the n - k nodes and link rest of the modified list with end of first k nodes.
4) Return new head of the list.

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Helper function for kAltReverse() */
struct node * _kAltReverse(struct node *node, int k, bool b);

/* Alternatively reverses the given linked list in groups of
   given size k. */
struct node *kAltReverse(struct node *head, int k)
{
  return _kAltReverse(head, k, true);
}

/*  Helper function for kAltReverse().  It reverses k nodes of the list
only if
    the third parameter b is passed as true, otherwise moves the pointer k
    nodes ahead and recursively calls iteself  */
struct node * _kAltReverse(struct node *node, int k, bool b)
{
   if(node == NULL)
       return NULL;

   int count = 1;
   struct node *prev = NULL;
   struct node  *current = node;
   struct node *next;

   /* The loop serves two purposes
      1) If b is true, then it reverses the k nodes
      2) If b is false, then it moves the current pointer */
```

```
    while(current != NULL && count <= k)
    {
        next = current->next;

        /* Reverse the nodes only if b is true*/
        if(b == true)
           current->next = prev;

        prev = current;
        current = next;
        count++;
    }

    /* 3) If b is true, then node is the kth node.
        So attach rest of the list after node.
      4) After attaching, return the new head */
    if(b == true)
    {
         node->next = _kAltReverse(current,k,!b);
         return prev;
    }

    /* If b is not true, then attach rest of the list after prev.
      So attach rest of the list after prev */
    else
    {
         prev->next = _kAltReverse(current, k, !b);
         return node;
    }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Function to print linked list */
void printList(struct node *node)
{
    int count = 0;
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
        count++;
    }
}
```

```
/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;
    int i;

    // create a list 1->2->3->4->5...... ->20
    for(i = 20; i > 0; i--)
      push(&head, i);

    printf("\n Given linked list \n");
    printList(head);
    head = kAltReverse(head, 3);

    printf("\n Modified Linked list \n");
    printList(head);

    getchar();
    return(0);
}
```

Output:

*Given linked list*

*1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20*

*Modified Linked list*

*3 2 1 4 5 6 9 8 7 10 11 12 15 14 13 16 17 18 20 19*

Time Complexity: O(n)

# 25.Add two numbers represented by linked lists

Given two numbers represented by two lists, write a function that returns sum list. The sum list is list representation of addition of two input numbers.

Example 1

Input:
First List: 5->6->3 // represents number 365
Second List: 8->4->2 // represents number 248
Output
Resultant list: 3->1->6 // represents number 613

Example 2

Input:
First List: 7->5->9->4->6 // represents number 64957
Second List: 8->4 // represents number 48
Output

Resultant list: 5->0->0->5->6  // represents number 65005

**Solution**

Traverse both lists. One by one pick nodes of both lists and add the values. If sum is more than 10 then make carry as 1 and reduce sum. If one list has more elements than the other then consider remaining values of this list as 0. Following is C implementation of this approach.

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

/* Linked list node */
struct node
{
   int data;
   struct node* next;
};

/* Function to create a new node with given data */
struct node *newNode(int data)
{
   struct node *new_node = (struct node *) malloc(sizeof(struct node));
   new_node->data = data;
   new_node->next = NULL;
   return new_node;
}

/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
   /* allocate node */
   struct node* new_node = newNode(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;
}

/* Adds contents of two linked lists and return the head node of resultant list */
struct node* addTwoLists (struct node* first, struct node* second)
{
   struct node* res = NULL; // res is head node of the resultant list
   struct node *temp, *prev = NULL;
   int carry = 0, sum;

   while (first != NULL || second != NULL) //while both lists exist
   {
      // Calculate value of next digit in resultant list.
      // The next digit is sum of following things
      // (i)  Carry
```

```
        // (ii) Next digit of first list (if there is a next digit)
        // (ii) Next digit of second list (if there is a next digit)
        sum = carry + (first? first->data: 0) + (second? second->data: 0);

        // update carry for next calulation
        carry = (sum >= 10)? 1 : 0;

        // update sum if it is greater than 10
        sum = sum % 10;

        // Create a new node with sum as data
        temp = newNode(sum);

        // if this is the first node then set it as head of the resultant list
        if(res == NULL)
           res = temp;
        else // If this is not the first node then connect it to the rest.
           prev->next = temp;

        // Set prev for next insertion
        prev  = temp;

        // Move first and second pointers to next nodes
        if (first) first = first->next;
        if (second) second = second->next;
    }

    if (carry > 0)
      temp->next = newNode(carry);

    // return head of the resultant list
    return res;
}

// A utility function to print a linked list
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d ", node->data);
        node = node->next;
    }
    printf("\n");
}

/* Drier program to test above function */
int main(void)
{
    struct node* res = NULL;
    struct node* first = NULL;
    struct node* second = NULL;

    // create first list 7->5->9->4->6
```

```
    push(&first, 6);
    push(&first, 4);
    push(&first, 9);
    push(&first, 5);
    push(&first, 7);
    printf("First List is ");
    printList(first);

    // create second list 8->4
    push(&second, 4);
    push(&second, 8);
    printf("Second List is ");
    printList(second);

    // Add the two lists and see result
    res = addTwoLists(first, second);
    printf("Resultant list is ");
    printList(res);

   return 0;
}
```

Output:

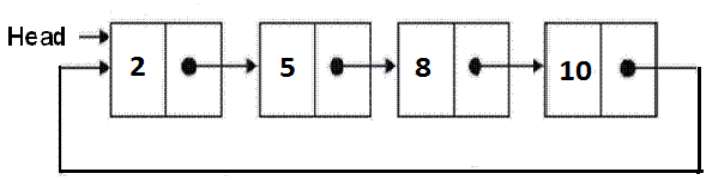
```
First List is 7 5 9 4 6
Second List is 8 4
Resultant list is 5 0 0 5 6
```

Time Complexity: O(m + n) where m and n are number of nodes in first and second lists respectively.

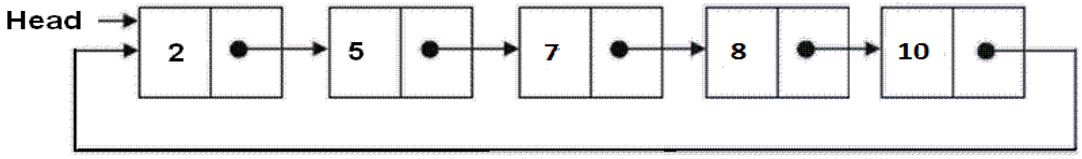
## 26.Sorted insert for circular linked list

**Difficulty Level:** Rookie

Write a C function to insert a new value in a sorted Circular Linked List (CLL). For example, if the input CLL is following.



After insertion of **7**, the above CLL should be changed to following




**Algorithm:**

Allocate memory for the newly inserted node and put data in the newly allocated node. Let the pointer to the new node be new_node. After memory allocation, following are the three cases that need to be handled.

1) *Linked List is empty:*

a) since new_node is the only node in CLL, make a self loop.

```
new_node->next = new_node;
```

b) change the head pointer to point to new node.

```
*head_ref = new_node;
```

2) *New node is to be inserted just before the head node:*

(a) Find out the last node using a loop.

```
while(current->next != *head_ref)
    current = current->next;
```

(b) Change the next of last node.

```
current->next = new_node;
```

(c) Change next of new node to point to head.

```
new_node->next = *head_ref;
```

(d) change the head pointer to point to new node.

```
*head_ref = new_node;
```

3) *New node is to be inserted somewhere after the head:*

(a) Locate the node after which new node is to be inserted.

```
while ( current->next!= *head_ref &&
      current->next->data < new_node->data)
{  current = current->next;  }
```

(b) Make next of new_node as next of the located pointer

```
new_node->next = current->next;
```

(c) Change the next of the located pointer

```
current->next = new_node;
```

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

/* structure for a node */
struct node
```

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

/* function to insert a new_node in a list in sorted way.
   Note that this function expects a pointer to head node
   as this can modify the head of the input linked list */
void sortedInsert(struct node** head_ref, struct node* new_node)
{
  struct node* current = *head_ref;

  // Case 1 of the above algo
  if (current == NULL)
  {
     new_node->next = new_node;
     *head_ref = new_node;
  }

  // Case 2 of the above algo
  else if (current->data >= new_node->data)
  {
    /* If value is smaller than head's value then
      we need to change next of last node */
    while(current->next != *head_ref)
        current = current->next;
    current->next = new_node;
    new_node->next = *head_ref;
    *head_ref = new_node;
  }

  // Case 3 of the above algo
  else
  {
    /* Locate the node before the point of insertion */
    while (current->next!= *head_ref && current->next->data < new_node-
>data)
      current = current->next;

    new_node->next = current->next;
    current->next = new_node;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *start)
{
  struct node *temp;

  if(start != NULL)
  {
    temp = start;
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != start);
  }
}

/* Driver program to test above functions */
int main()
```

```
{
  int arr[] = {12, 56, 2, 11, 1, 90};
  int list_size, i;

  /* start with empty linked list */
  struct node *start = NULL;
  struct node *temp;

  /* Create linked list from the array arr[].
    Created linked list will be 1->2->11->56->12 */
  for(i = 0; i< 6; i++)
  {
    temp = (struct node *)malloc(sizeof(struct node));
    temp->data = arr[i];
    sortedInsert(&start, temp);
  }

  printList(start);
  getchar();
  return 0;
}
```

Output:

*1 2 11 12 56 90*

Time Complexity: O(n) where n is the number of nodes in the given linked list.

Case 2 of the above algorithm/code can be optimized.

implement the suggested change we need to modify the case 2 to following.

```
// Case 2 of the above algo
else if (current->data >= new_node->data)
{
  // swap the data part of head node and new node
  swap(&(current->data), &(new_node->data));  // assuming that we have a function
swap(int *, int *)

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

# 27.Linked List vs Array

**Difficulty Level:** Rookie

Both Arrays and Linked List can be used to store linear data of similar types, but they both have some advantages and disadvantages over each other.

Following are the points in favour of Linked Lists.

(1) The size of the arrays is fixed: So we must know the upper limit on the number of elements in advance. Also, generally, the allocated memory is equal to the upper limit irrespective of the usage, and in practical uses, upper limit is rarely reached.

(2) Inserting a new element in an array of elements is expensive, because room has to be created for the new elements and to create room existing elements have to shifted.

For example, suppose we maintain a sorted list of IDs in an array id[].

id[] = [1000, 1010, 1050, 2000, 2040, .....].

And if we want to insert a new ID 1005, then to maintain the sorted order, we have to move all the elements after 1000 (excluding 1000).

Deletion is also expensive with arrays until unless some special techniques are used. For example, to delete 1010 in id[], everything after 1010 has to be moved.

So Linked list provides following two advantages over arrays
1) Dynamic size
2) Ease of insertion/deletion

Linked lists have following drawbacks:
1) Random access is not allowed. We have to access elements sequentially starting from the first node. So we cannot do binary search with linked lists.
2) Extra memory space for a pointer is required with each element of the list.
3) Arrays have better cache locality that can make a pretty big difference in performance

# 28.Reverse a Linked List in groups of given size

Given a linked list, write a function to reverse every k nodes (where k is an input to the function).

```
Example:
Inputs:  1->2->3->4->5->6->7->8->NULL and k = 3
Output:  3->2->1->6->5->4->8->7->NULL.

Inputs:   1->2->3->4->5->6->7->80->NULL and k = 5
Output:  5->4->3->2->1->8->7->6->NULL.
```

Algorithm: *reverse(head, k)*
1) Reverse the first sub-list of size k. While reversing keep track of the next node and previous node. Let the pointer to the next node be *next* and pointer to the previous node be *prev*. See this post for reversing a linked list.

2) *head->next = reverse(next, k)* /* Recursively call for rest of the list and link the two sub-lists */

3) return *prev* /* *prev* becomes the new head of the list (see the diagrams of iterative method of this post) */

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* Reverses the linked list in groups of size k and returns the pointer to the new
head node */
struct node *reverse (struct node *head, int k)
{
    struct node* current = head;
    struct node* next;
    struct node* prev = NULL;
    int count = 0;

    /*reverse first k nodes of the linked list */
    while (current != NULL && count < k)
    {
       next  = current->next;
       current->next = prev;
       prev = current;
       current = next;
       count++;
    }

    /* next is now a pointer to (k+1)th node
       Recursively call for the list starting from current.
       And make rest of the list as next of first node */
    if(next !=  NULL)
    {  head->next = reverse(next, k); }

    /* prev is new head of the input list */
    return prev;
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
```

```
}

/* Function to print linked list */
void printList(struct node *node)
{
    while(node != NULL)
    {
        printf("%d  ", node->data);
        node = node->next;
    }
}

/* Drier program to test above function*/
int main(void)
{
    /* Start with the empty list */
    struct node* head = NULL;

     /* Created Linked list is 1->2->3->4->5->6->7->8 */
     push(&head, 8);
     push(&head, 7);
     push(&head, 6);
     push(&head, 5);
     push(&head, 4);
     push(&head, 3);
     push(&head, 2);
     push(&head, 1);

     printf("\n Given linked list \n");
     printList(head);
     head = reverse(head, 3);

     printf("\n Reversed Linked list \n");
     printList(head);

     getchar();
     return(0);
}
```

Time Complexity: O(n) where n is the number of nodes in the given list.

# 29.Merge two sorted linked lists

Write a SortedMerge() function that takes two lists, each of which is sorted in increasing order, and merges the two together into one list which is in increasing order. SortedMerge() should return the new list. The new list should be made by splicing
together the nodes of the first two lists.

For example if the first linked list a is 5->10->15 and the other linked list b is 2->3->20, then SortedMerge() should return a pointer to the head node of the merged list 2->3->5->10->15->20.

There are many cases to deal with: either ‘a’ or ‘b’ may be empty, during processing either ‘a’ or ‘b’ may run out first, and finally there’s the problem of starting the result list empty, and building it up while going through ‘a’ and ‘b’.

**Method 1 (Using Dummy Nodes)**

The strategy here uses a temporary dummy node as the start of the result list. The pointer Tail always points to the last node in the result list, so appending new nodes is easy.

The dummy node gives tail something to point to initially when the result list is empty. This dummy node is efficient, since it is only temporary, and it is allocated in the stack. The loop proceeds, removing one node from either ‘a’ or ‘b’, and adding it to tail. When

we are done, the result is in dummy.next.

```
/*Program to alternatively split a linked list into two halves */
#include<stdio.h>
#include<stdlib.h>
#include<assert.h>

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* pull off the front node of the source and put it in dest */
void MoveNode(struct node** destRef, struct node** sourceRef);

/* Takes two lists sorted in increasing order, and splices their nodes together to make
is returned.  */
struct node* SortedMerge(struct node* a, struct node* b)
{
   /* a dummy first node to hang the result on */
   struct node dummy;

   /* tail points to the last result node  */
   struct node* tail = &dummy;

   /* so tail->next is the place to add new nodes
     to the result. */
   dummy.next = NULL;
   while(1)
   {
      if(a == NULL)
      {
         /* if either list runs out, use the other list */
         tail->next = b;
         break;
      }
      else if (b == NULL)
      {
         tail->next = a;
         break;
      }
      if (a->data <= b->data)
      {
         MoveNode(&(tail->next), &a);
```

```
      }
      else
     {
        MoveNode(&(tail->next), &b);
     }
     tail = tail->next;
  }
  return(dummy.next);
}

/* UTILITY FUNCTIONS */
/*MoveNode() function takes the node from the front of the source,
and move it to the front of the dest.
   It is an error to call this with the source list empty.

   Before calling MoveNode():
   source == {1, 2, 3}
   dest == {1, 2, 3}

   Affter calling MoveNode():
   source == {2, 3}
   dest == {1, 1, 2, 3}
*/
void MoveNode(struct node** destRef, struct node** sourceRef)
{
  /* the front source node  */
  struct node* newNode = *sourceRef;
  assert(newNode != NULL);

  /* Advance the source pointer */
  *sourceRef = newNode->next;

  /* Link the old dest off the new node */
  newNode->next = *destRef;

  /* Move dest to point to the new node */
  *destRef = newNode;
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
```

```
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create two sorted linked lists to test the functions
   Created lists shall be a: 5->10->15,  b: 2->3->20 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);

  push(&b, 20);
  push(&b, 3);
  push(&b, 2);

  /* Remove duplicates from linked list */
  res = SortedMerge(a, b);

  printf("\n Merged Linked List is: \n");
  printList(res);

  getchar();
  return 0;
}
```

**Method 2 (Using Local References)**

This solution is structurally very similar to the above, but it avoids using a dummy node. Instead, it maintains a struct node** pointer, lastPtrRef, that always points to the last pointer of the result list. This solves the same case that the dummy node did — dealing with the result list when it is empty. If you are trying to build up a list at its tail, either the dummy node or the struct node** "reference" strategy can be used (see Section 1 for details).

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* point to the last result pointer */
  struct node** lastPtrRef = &result;

  while(1)
  {
    if (a == NULL)
    {
      *lastPtrRef = b;
       break;
    }
    else if (b==NULL)
    {
       *lastPtrRef = a;
```

```
       break;
    }
    if(a->data <= b->data)
    {
      MoveNode(lastPtrRef, &a);
    }
    else
    {
      MoveNode(lastPtrRef, &b);
    }

    /* tricky: advance to point to the next ".next" field */
    lastPtrRef = &((*lastPtrRef)->next);
  }
  return(result);
}
```

**Method 3 (Using Recursion)**

Merge is one of those nice recursive problems where the recursive solution code is much cleaner than the iterative code. You probably wouldn't want to use the recursive version for production code however, because it will use stack space which is proportional to the length of the lists.

```
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}
```

# 30.Merge Sort for Linked Lists

Merge sort is often preferred for sorting a linked list. The slow random-access performance of a linked list makes some other algorithms (such as quicksort) perform poorly, and others (such as heapsort) completely impossible.

Let head be the first node of the linked list to be sorted and headRef be the pointer to head. Note that we need a reference to head in MergeSort() as the below implementation changes next links to

sort the linked lists (not data at the nodes), so head node has to be changed if the data at original head is not the smallest value in linked list.

```
MergeSort(headRef)
1) If head is NULL or there is only one element in the Linked List
    then return.
2) Else divide the linked list into two halves.
      FrontBackSplit(head, &a, &b); /* a and b are two halves */
3) Sort the two halves a and b.
      MergeSort(a);
      MergeSort(b);
4) Merge the sorted a and b (using SortedMerge() discussed here)
   and update the head pointer using headRef.
     *headRef = SortedMerge(a, b);
```

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

/* Link list node */
struct node
{
    int data;
    struct node* next;
};

/* function prototypes */
struct node* SortedMerge(struct node* a, struct node* b);
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef);

/* sorts the linked list by changing next pointers (not data) */
void MergeSort(struct node** headRef)
{
  struct node* head = *headRef;
  struct node* a;
  struct node* b;

  /* Base case -- length 0 or 1 */
  if ((head == NULL) || (head->next == NULL))
  {
    return;
  }

  /* Split head into 'a' and 'b' sublists */
  FrontBackSplit(head, &a, &b);

  /* Recursively sort the sublists */
  MergeSort(&a);
  MergeSort(&b);

  /* answer = merge the two sorted lists together */
  *headRef = SortedMerge(a, b);
```

```
}

/* See http://geeksforgeeks.org/?p=3622 for details of this
   function */
struct node* SortedMerge(struct node* a, struct node* b)
{
  struct node* result = NULL;

  /* Base cases */
  if (a == NULL)
     return(b);
  else if (b==NULL)
     return(a);

  /* Pick either a or b, and recur */
  if (a->data <= b->data)
  {
     result = a;
     result->next = SortedMerge(a->next, b);
  }
  else
  {
     result = b;
     result->next = SortedMerge(a, b->next);
  }
  return(result);
}

/* UTILITY FUNCTIONS */
/* Split the nodes of the given list into front and back halves,
     and return the two lists using the reference parameters.
     If the length is odd, the extra node should go in the front list.
     Uses the fast/slow pointer strategy.  */
void FrontBackSplit(struct node* source,
          struct node** frontRef, struct node** backRef)
{
  struct node* fast;
  struct node* slow;
  if (source==NULL || source->next==NULL)
  {
    /* length < 2 cases */
    *frontRef = source;
    *backRef = NULL;
  }
  else
  {
    slow = source;
    fast = source->next;

    /* Advance 'fast' two nodes, and advance 'slow' one node */
    while (fast != NULL)
    {
      fast = fast->next;
      if (fast != NULL)
      {
        slow = slow->next;
        fast = fast->next;
      }
    }

    /* 'slow' is before the midpoint in the list, so split it in two
      at that point. */
```

```
    *frontRef = source;
    *backRef = slow->next;
    slow->next = NULL;
  }
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

/* Function to insert a node at the beginging of the linked list */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Drier program to test above functions*/
int main()
{
  /* Start with the empty list */
  struct node* res = NULL;
  struct node* a = NULL;
  struct node* b = NULL;

  /* Let us create a unsorted linked lists to test the functions
   Created lists shall be a: 2->3->20->5->10->15 */
  push(&a, 15);
  push(&a, 10);
  push(&a, 5);
  push(&a, 20);
  push(&a, 3);
  push(&a, 2);

  /* Remove duplicates from linked list */
  MergeSort(&a);

  printf("\n Sorted Linked List is: \n");
  printList(a);

  getchar();
  return 0;
}
```

Time Complexity: O(nLogn)

# 31.Delete alternate nodes of a Linked List

Given a Singly Linked List, starting from the second node delete all alternate nodes of it. For example, if the given linked list is 1->2->3->4->5 then your function should convert it to 1->3->5, and if the given linked list is 1->2->3->4 then convert it to 1->3.

**Method 1 (Iterative)**

Keep track of previous of the node to be deleted. First change the next link of previous node and then free the memory allocated for the node.

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

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

/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
  if (head == NULL)
    return;

   /* Initialize prev and node to be deleted */
  struct node *prev = head;
  struct node *node = head->next;

  while (prev != NULL && node != NULL)
  {
     /* Change next link of previous node */
     prev->next = node->next;

     /* Free memory */
     free(node);

     /* Update prev and node */
     prev = prev->next;
     if(prev != NULL)
        node = prev->next;
  }
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* 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(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct 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;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Drier program to test above functions */
int main()
{
  int arr[100];

  /* Start with the empty list */
  struct node* head = NULL;

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n List before calling deleteAlt() ");
  printList(head);

  deleteAlt(head);

  printf("\n List after calling deleteAlt() ");
  printList(head);

  getchar();
  return 0;
}
```

Time Complexity: O(n) where n is the number of nodes in the given Linked List.

**Method 2 (Recursive)**

Recursive code uses the same approach as method 1. The recursive coed is simple and short, but causes O(n) recursive function calls for a linked list of size n.

```
/* deletes alternate nodes of a list starting with head */
void deleteAlt(struct node *head)
{
```

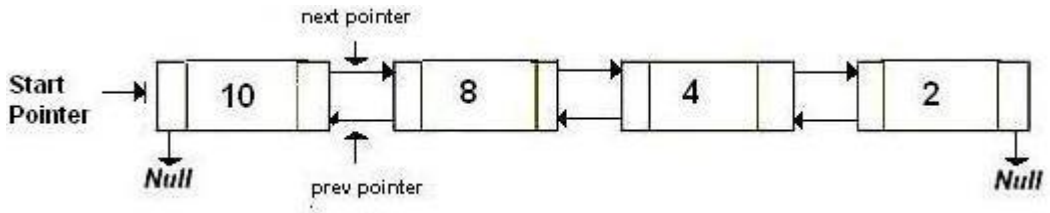
```
  if (head == NULL)
    return;

  struct node *node = head->next;

  if (node == NULL)
    return;

  /* Change the next link of head */
  head->next = node->next;

  /* free memory allocated for node */
  free(node);

  /* Recursively call for the new next of head */
  deleteAlt(head->next);
}
```

Time Complexity: O(n)

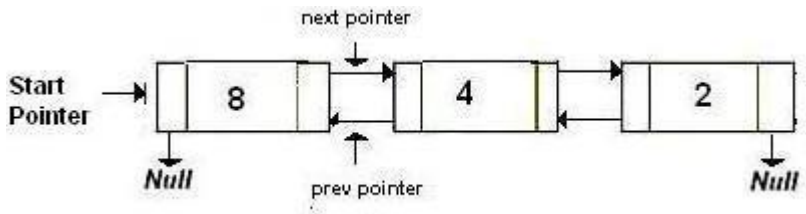
# 32.Delete a node in a Doubly Linked List

Write a function to delete a given node in a doubly linked list.

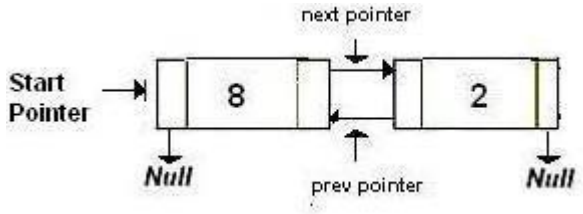
**(a) Original Doubly Linked List**



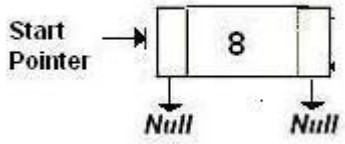

**(a) After deletion of head node**




**(a) After deletion of middle node**




**(a) After deletion of last node**




Algorithm

Let the node to be deleted is *del*.

1) If node to be deleted is head node, then change the head pointer to next current head.

2) Set *next* of previous to *del*, if previous to *del* exixts.

3) Set *prev* of next to *del*, if next to *del* exixts.

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

/* a node of the doubly linked list */
struct node
{
  int data;
  struct node *next;
  struct node *prev;
};

/* Function to delete a node in a Doubly Linked List.
   head_ref --> pointer to head node pointer.
   del  -->  pointer to node to be deleted. */
void deleteNode(struct node **head_ref, struct node *del)
{
  /* base case */
  if(*head_ref == NULL || del == NULL)
    return;

  /* If node to be deleted is head node */
  if(*head_ref == del)
    *head_ref = del->next;

  /* Change next only if node to be deleted is NOT the last node */
  if(del->next != NULL)
    del->next->prev = del->prev;

  /* Change prev only if node to be deleted is NOT the first node */
  if(del->prev != NULL)
    del->prev->next = del->next;
```

```
  /* Finally, free the memory occupied by del*/
  free(del);
  return;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the beginning of the Doubly Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
       (struct node*) malloc(sizeof(struct node));

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

  /* since we are adding at the begining,
    prev is always NULL */
  new_node->prev = NULL;

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

  /* change prev of head node to new node */
  if((*head_ref) !=  NULL)
   (*head_ref)->prev = new_node ;

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

/* Function to print nodes in a given doubly linked list
   This function is same as printList() of singly linked lsit */
void printList(struct node *node)
{
  while(node!=NULL)
  {
   printf("%d ", node->data);
   node = node->next;
  }
}

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

  /* Let us create the doubly linked list 10<->8<->4<->2 */
  push(&head, 2);
  push(&head, 4);
  push(&head, 8);
  push(&head, 10);

  printf("\n Original Linked list ");
  printList(head);

  /* delete nodes from the doubly linked list */
  deleteNode(&head, head);  /*delete first node*/
```

```
  deleteNode(&head, head->next);  /*delete middle node*/
  deleteNode(&head, head->next);  /*delete last node*/

  /* Modified linked list will be NULL<-8->NULL */
  printf("\n Modified Linked list ");
  printList(head);

  getchar();
}
```

Time Complexity: O(1)

Time Complexity: O(1)

# 33.Pairwise swap elements of a given linked list

Given a singly linked list, write a function to swap elements pairwise. For example, if the linked list is 1->2->3->4->5 then the function should change it to 2->1->4->3->5, and if the linked list is 1->2->3->4->5->6 then the function should change it to 2->1->4->3->6->5.

**METHOD 1 (Iterative)**

Start from the head node and traverse the list. While traversing swap data of each node with its next node's data.

```
/* Program to pairwise swap elements in a given linked list */
#include<stdio.h>
#include<stdlib.h>

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

/*Function to swap two integers at addresses a and b */
void swap(int *a, int *b);

/* Function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  struct node *temp = head;

   /* Traverse further only if there are at-least two nodes left */
  while(temp != NULL && temp->next != NULL)
  {
    /* Swap data of node with its next node's data */
    swap(&temp->data, &temp->next->data);

    /* Move temp by 2 for the next pair */
    temp = temp->next->next;
  }
}

/* UTILITY FUNCTIONS */
```

```
/* Function to swap two integers */
void swap(int *a, int *b)
{
  int temp;
  temp = *a;
  *a = *b;
  *b = temp;
}

/* Function to add a node at the begining of Linked List */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}

/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   1->2->3->4->5 */
  push(&start, 5);
  push(&start, 4);
  push(&start, 3);
  push(&start, 2);
  push(&start, 1);

  printf("\n Linked list before calling  pairWiseSwap() ");
  printList(start);

  pairWiseSwap(start);

  printf("\n Linked list after calling  pairWiseSwap() ");
  printList(start);

  getchar();
  return 0;
}
```

Time complexity: O(n)

**METHOD 2 (Recursive)**

If there are 2 or more than 2 nodes in Linked List then swap the first two nodes and recursively call for rest of the list.

```
/* Recursive function to pairwise swap elements of a linked list */
void pairWiseSwap(struct node *head)
{
  /* There must be at-least two nodes in the list */
  if(head != NULL && head->next != NULL)
  {
    /* Swap the node's data with data of next node */
    swap(&head->data, &head->next->data);

    /* Call pairWiseSwap() for rest of the list */
    pairWiseSwap(head->next->next);
  }
}
```

Time complexity: O(n)

# 34.Move last element to front of a given Linked List

Write a C function that moves last element to front in a given Singly Linked List. For example, if the given Linked List is 1->2->3->4->5, then the function should change the list to 5->1->2->3->4.

Algorithm:

Traverse the list till last node. Use two pointers: one to store the address of last node and other for address of second last node. After the end of loop do following operations.

i) Make second last as last (secLast->next = NULL).

ii) Set next of last as head (last->next = *head_ref).

iii) Make last as head ( *head_ref = last)

# 35.Practice questions for Linked List and Recursion

Assume the structure of a Linked List node is as follows.

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

Explain the functionality of following C functions.

**1. What does the following function do for a given Linked List?**

```
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}
```

fun1() prints the given Linked List in reverse manner. For Linked List 1->2->3->4->5, fun1() prints 5->4->3->2->1.

**2. What does the following function do for a given Linked List ?**

```
void fun2(struct node* head)
{
  if(head== NULL)
    return;
  printf("%d  ", head->data);

  if(head->next != NULL )
    fun2(head->next->next);
  printf("%d  ", head->data);
}
```

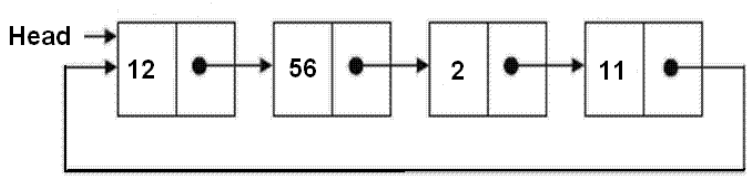
fun2() prints alternate nodes of the given Linked List, first from head to end, and then from end to head. If Linked List has even number of nodes, then fun2() skips the last node. For Linked List 1->2->3->4->5, fun2() prints 1 3 5 5 3 1. For Linked List 1->2->3->4->5->6, fun2() prints 1 3 5 5 3 1.

Below is a complete running program to test above functions.

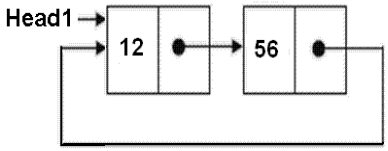
```
#include<stdio.h>
#include<stdlib.h>

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

/* Prints a linked list in reverse manner */
void fun1(struct node* head)
{
  if(head == NULL)
    return;

  fun1(head->next);
  printf("%d  ", head->data);
}

/* prints alternate nodes of a Linked List, first
  from head to end, and then from end to head. */
```

```
void fun2(struct node* start)
{
  if(start == NULL)
    return;
  printf("%d  ", start->data);

  if(start->next != NULL )
    fun2(start->next->next);
  printf("%d  ", start->data);
}

/* UTILITY FUNCTIONS TO TEST fun1() and fun2() */
/* 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(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
          (struct node*) malloc(sizeof(struct 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;
}

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

  /* Using push() to construct below list
    1->2->3->4->5  */
  push(&head, 5);
  push(&head, 4);
  push(&head, 3);
  push(&head, 2);
  push(&head, 1);

  printf("\n Output of fun1() for list 1->2->3->4->5 \n");
  fun1(head);

  printf("\n Output of fun2() for list 1->2->3->4->5 \n");
  fun2(head);

  getchar();
  return 0;
}
```

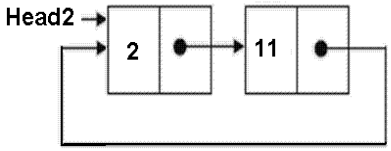
# 36.Split a Circular Linked List into two halves




Original Linked List




Result Linked List 1




Result Linked List 2

Algo:.

1) Store the mid and last pointers of the circular linked list using tortoise and hare algorithm.

2) Make the second half circular.

3) Make the first half circular.

4) Set head (or start) pointers of the two linked lists.

In the below implementation, if there are odd nodes in the given circular linked list then the first result list has 1 more node than the second result list.

```
/* Program to split a circular linked list into two halves */
#include<stdio.h>
#include<stdlib.h>

/* structure for a node */
struct node
{
  int data;
  struct node *next;
};
```

```
/* Function to split a list (starting with head) into two lists.
   head1_ref and head2_ref are references to head nodes of
    the two resultant linked lists */
void splitList(struct node *head, struct node **head1_ref,
                                                struct node **head2_ref)
{
  struct node *slow_ptr = head;
  struct node *fast_ptr = head;

  if(head == NULL)
    return;

  /* If there are odd nodes in the circular list then
     fast_ptr->next becomes head and for even nodes
     fast_ptr->next->next becomes head */
  while(fast_ptr->next != head &&
        fast_ptr->next->next != head)
  {
     fast_ptr = fast_ptr->next->next;
     slow_ptr = slow_ptr->next;
  }

 /* If there are even elements in list then move fast_ptr */
  if(fast_ptr->next->next == head)
    fast_ptr = fast_ptr->next;

  /* Set the head pointer of first half */
  *head1_ref = head;

  /* Set the head pointer of second half */
  if(head->next != head)
    *head2_ref = slow_ptr->next;

  /* Make second half circular */
  fast_ptr->next = slow_ptr->next;

  /* Make first half circular */
  slow_ptr->next = head;
}

/* UTILITY FUNCTIONS */
/* Function to insert a node at the begining of a Circular
   linked lsit */
void push(struct node **head_ref, int data)
{
  struct node *ptr1 = (struct node *)malloc(sizeof(struct node));
  struct node *temp = *head_ref;
  ptr1->data = data;
  ptr1->next = *head_ref;

  /* If linked list is not NULL then set the next of
    last node */
  if(*head_ref != NULL)
  {
    while(temp->next != *head_ref)
      temp = temp->next;
    temp->next = ptr1;
  }
  else
     ptr1->next = ptr1; /*For the first node */
```

```
  *head_ref = ptr1;
}

/* Function to print nodes in a given Circular linked list */
void printList(struct node *head)
{
  struct node *temp = head;
  if(head != NULL)
  {
    printf("\n");
    do {
      printf("%d ", temp->data);
      temp = temp->next;
    } while(temp != head);
  }
}

/* Driver program to test above functions */
int main()
{
  int list_size, i;

  /* Initialize lists as empty */
  struct node *head = NULL;
  struct node *head1 = NULL;
  struct node *head2 = NULL;

  /* Created linked list will be 12->56->2->11 */
  push(&head, 12);
  push(&head, 56);
  push(&head, 2);
  push(&head, 11);

  printf("Original Circular Linked List");
  printList(head);

  /* Split the list */
  splitList(head, &head1, &head2);

  printf("\nFirst Circular Linked List");
  printList(head1);

  printf("\nSecond Circular Linked List");
  printList(head2);

  getchar();
  return 0;
}
```

Time Complexity: O(n)

# 37.Remove duplicates from an unsorted linked list

Write a removeDuplicates() function which takes a list and deletes any duplicate nodes from the list. The list is not sorted.

For example if the linked list is 12->11->12->21->41->43->21 then removeDuplicates() should convert the list to 12->11->21->41->43.

**METHOD 1 (Using two loops)**

This is the simple way where two loops are used. Outer loop is used to pick the elements one by one and inner loop compares the picked element with rest of the elements.

```
/* Program to remove duplicates in an unsorted array */

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

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

/* Function to remove duplicates from a unsorted linked list */
void removeDuplicates(struct node *start)
{
  struct node *ptr1, *ptr2, *dup;
  ptr1 = start;

  /* Pick elements one by one */
  while(ptr1 != NULL && ptr1->next != NULL)
  {
     ptr2 = ptr1;

     /* Compare the picked element with rest of the elements */
     while(ptr2->next != NULL)
     {
       /* If duplicate then delete it */
       if(ptr1->data == ptr2->next->data)
       {
          /* sequence of steps is important here */
          dup = ptr2->next;
          ptr2->next = ptr2->next->next;
          free(dup);
       }
       else /* This is tricky */
       {
          ptr2 = ptr2->next;
       }
     }
     ptr1 = ptr1->next;
  }
}

/* UTILITY FUNCTIONS */
/* Function to push a node */
void push(struct node** head_ref, int new_data);

/* Function to print nodes in a given linked list */
void printList(struct node *node);
```

```
/* Druver program to test above function */
int main()
{
  struct node *start = NULL;

  /* The constructed linked list is:
   10->12->11->11->12->11->10*/
  push(&start, 10);
  push(&start, 11);
  push(&start, 12);
  push(&start, 11);
  push(&start, 11);
  push(&start, 12);
  push(&start, 10);

  printf("\n Linked list before removing duplicates ");
  printList(start);

  removeDuplicates(start);

  printf("\n Linked list after removing duplicates ");
  printList(start);

  getchar();
}

/* Function to push a node */
void push(struct node** head_ref, int new_data)
{
  /* allocate node */
  struct node* new_node =
            (struct node*) malloc(sizeof(struct 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;
}

/* Function to print nodes in a given linked list */
void printList(struct node *node)
{
  while(node != NULL)
  {
    printf("%d ", node->data);
    node = node->next;
  }
}
```

Time Complexity: O(n^2)

**METHOD 2 (Use Sorting)**

In general, Merge Sort is the best suited sorting algorithm for sorting linked lists efficiently.

1) Sort the elements using Merge Sort. We will soon be writing a post about sorting a linked list. O(nLogn)
2) Remove duplicates in linear time using the algorithm for removing duplicates in sorted Linked List. O(n)

Time Complexity: O(nLogn)

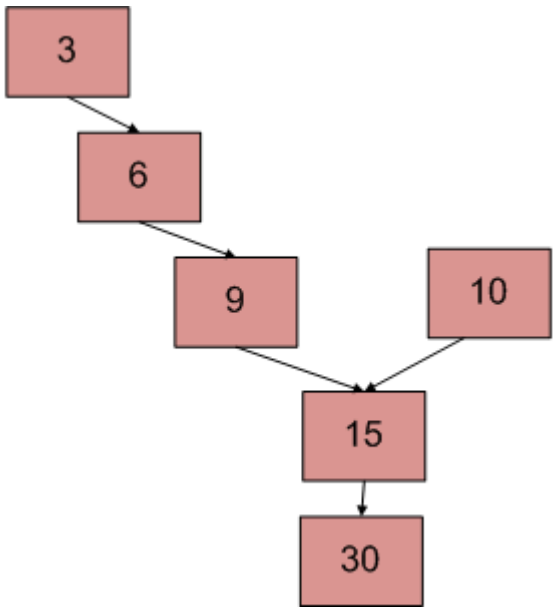
**METHOD 3 (Use Hashing)**
We traverse the link list from head to end. For the newly encountered element, we check whether it is in the hash table: if yes, we remove it; otherwise we put it in the hash table.

Thanks to bearwang for suggesting this method.

Time Complexity: O(n) on average (assuming that hash table access time is O(1) on average).

## 38.Write a function to get the intersection point of two Linked Lists.

There are two singly linked lists in a system. By some programming error the end node of one of the linked list got linked into the second list, forming a inverted Y shaped list. Write a program to get the point where two linked list merge.




Above diagram shows an example with two linked list having 15 as intersection point.

**Method 1(Simply use two loops)**
Use 2 nested for loops. Outer loop will be for each node of the 1st list and inner loop will be for 2nd

list. In the inner loop, check if any of nodes of 2nd list is same as the current node of first linked list. Time complexity of this method will be O(mn) where m and n are the number of nodes in two lists.

**Method 2 (Mark Visited Nodes)**

This solution requires modifications to basic linked list data structure. Have a visited flag with each node. Traverse the first linked list and keep marking visited nodes. Now traverse second linked list, If you see a visited node again then there is an intersection point, return the intersecting node. This solution works in O(m+n) but requires additional information with each node. A variation of this solution that doesn't require modification to basic data structure can be implemented using hash. Traverse the first linked list and store the addresses of visited nodes in a hash. Now traverse the second linked list and if you see an address that already exists in hash then return the intersecting node.

**Method 3(Using difference of node counts)**

1) Get count of the nodes in first list, let count be c1.

2) Get count of the nodes in second list, let count be c2.

3) Get the difference of counts d = abs(c1 – c2)

4) Now traverse the bigger list from the first node till d nodes so that from here onwards both the lists have equal no of nodes.

5) Then we can traverse both the lists in parallel till we come across a common node. (Note that getting a common node is done by comparing the address of the nodes)

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

/* Link list node */
struct node
{
  int data;
  struct node* next;
};

/* Function to get the counts of node in a linked list */
int getCount(struct node* head);

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2);

/* function to get the intersection point of two linked
   lists head1 and head2 */
int getIntesectionNode(struct node* head1, struct node* head2)
{
  int c1 = getCount(head1);
  int c2 = getCount(head2);
  int d;
```

```
  if(c1 > c2)
  {
    d = c1 - c2;
    return _getIntesectionNode(d, head1, head2);
  }
  else
  {
    d = c2 - c1;
    return _getIntesectionNode(d, head2, head1);
  }
}

/* function to get the intersection point of two linked
   lists head1 and head2 where head1 has d more nodes than
   head2 */
int _getIntesectionNode(int d, struct node* head1, struct node* head2)
{
  int i;
  struct node* current1 = head1;
  struct node* current2 = head2;

  for(i = 0; i < d; i++)
  {
    if(current1 == NULL)
    {  return -1; }
    current1 = current1->next;
  }

  while(current1 !=  NULL && current2 != NULL)
  {
    if(current1 == current2)
      return current1->data;
    current1= current1->next;
    current2= current2->next;
  }

  return -1;
}

/* Takes head pointer of the linked list and
   returns the count of nodes in the list */
int getCount(struct node* head)
{
  struct node* current = head;
  int count = 0;

  while (current != NULL)
  {
    count++;
    current = current->next;
  }
```

```
  return count;
}

/* IGNORE THE BELOW LINES OF CODE. THESE LINES
   ARE JUST TO QUICKLY TEST THE ABOVE FUNCTION */
int main()
{
  /*
    Create two linked lists

    1st 3->6->9->15->30
    2nd 10->15->30

    15 is the intersection point
  */

  struct node* newNode;
  struct node* head1 =
         (struct node*) malloc(sizeof(struct node));
  head1->data  = 10;

  struct node* head2 =
         (struct node*) malloc(sizeof(struct node));
  head2->data  = 3;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 6;
  head2->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 9;
  head2->next->next = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 15;
  head1->next = newNode;
  head2->next->next->next  = newNode;

  newNode = (struct node*) malloc (sizeof(struct node));
  newNode->data = 30;
  head1->next->next= newNode;

  head1->next->next->next = NULL;

  printf("\n The node of intersection is %d \n",
        getIntesectionNode(head1, head2));

  getchar();
}
```

**Time Complexity:** O(m+n)

**Auxiliary Space:** O(1)

**Method 4(Make circle in first list)**

1. Traverse the first linked list(count the elements) and make a circular linked list. (Remember last node so that we can break the circle later on).
2. Now view the problem as find the loop in the second linked list. So the problem is solved.
3. Since we already know the length of the loop(size of first linked list) we can traverse those many number of nodes in second list, and then start another pointer from the beginning of second list. we have to traverse until they are equal, and that is the required intersection point.
4. remove the circle from the linked list.

**Time Complexity:** O(m+n)
**Auxiliary Space:** O(1)

**Method 5 (Reverse the first list and make equations)**

1) Let X be the length of the first linked list until intersection point.

```
   Let Y be the length of the second linked list until the
intersection point.
   Let Z be the length of the linked list from intersection point to
End of
   the linked list including the intersection node.
   We Have
           X + Z = C1;
           Y + Z = C2;
2) Reverse first linked list.
3) Traverse Second linked list. Let C3 be the length of second list
- 1.
     Now we have
        X + Y = C3
     We have 3 linear equations. By solving them, we get
       X = (C1 + C3 - C2)/2;
       Y = (C2 + C3 - C1)/2;
       Z = (C1 + C2 - C3)/2;
      WE GOT THE INTERSECTION POINT.
4)  Reverse first linked list.
```

Advantage: No Comparison of pointers.
Disadvantage : Modifying linked list(Reversing list).

**Time complexity:** O(m+n)

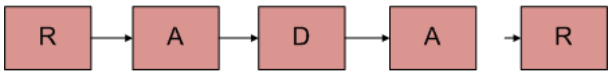
**Auxiliary Space:** O(1)

**Method 6 (Traverse both lists and compare addresses of last nodes)** This method is only to detect if there is an intersection point or not

```
1) Traverse the list 1, store the last node address
2) Traverse the list 2, store the last node address.
3) If nodes stored in 1 and 2 are same then they are intersecting.
```

Time complexity of this method is O(m+n) and used Auxiliary space is O(1)

# 39.Function to check if a singly linked list is palindrome




**METHOD 1 (By reversing the list)**

```
   1.  Get the middle of the linked list.
   2.  Reverse the second half of the linked list.
   3.  Compare the first half and second half.
   4.  Construct the original linked list by reversing the
       second half again and attaching it back to the first half
```

Implementation:

```
/* Program to check if a linked list is palindrome */
#include<stdio.h>
#include<stdlib.h>
#define bool int

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

void reverse(struct node**);
bool compareLists(struct node*, struct node *);

/* Function to check if given linked list is
  palindrome or not */
bool isPalindrome(struct node *head)
```

```
{
   struct node *slow_ptr = head;
   struct node *fast_ptr = head;
   struct node *second_half;
   struct node *prev_of_slow_ptr = head;
   char res;

   if(head!=NULL)
   {
       /* Get the middle of the list. Move slow_ptr by 1
         and fast_ptrr by 2, slow_ptr will have the |_n/2_|th
         node */
       while((fast_ptr->next)!=NULL &&
               (fast_ptr->next->next)!=NULL)
       {
          fast_ptr = fast_ptr->next->next;

          /*We need previous of the slow_ptr for
           linked lists  with odd elements */
          prev_of_slow_ptr = slow_ptr;
          slow_ptr = slow_ptr->next;
       }

       /* Case where we have even no of elements */
       if(fast_ptr->next != NULL)
       {
         second_half = slow_ptr->next;
         reverse(&second_half);
         slow_ptr->next = NULL;
         res = compareLists(head, second_half);

         /*construct the original list back*/
         reverse(&second_half);
         slow_ptr->next = second_half;
       }

       /* Case where we have odd no. of elements. Neither first
          nor second list should have the middle element */
       else
       {
          second_half = slow_ptr->next;
          prev_of_slow_ptr->next = NULL;
          reverse(&second_half);
          res = compareLists(head, second_half);

         /*construct the original list back*/
         reverse(&second_half);
         prev_of_slow_ptr->next = slow_ptr;
         slow_ptr->next = second_half;
       }

       return res;
   }
}

/* Function to reverse the linked list  Note that this
    function may change the head */
void reverse(struct node** head_ref)
{
    struct node* prev   = NULL;
    struct node* current = *head_ref;
    struct node* next;
```

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

/* Function to check if two input lists have same data*/
int compareLists(struct node* head1, struct node *head2)
{
    struct node* temp1 = head1;
    struct node* temp2 = head2;

    while(temp1 && temp2)
    {
       if(temp1->data == temp2->data)
       {
          temp1 = temp1->next;
          temp2 = temp2->next;
       }
       else return 0;
    }

    /* Both are empty reurn 1*/
    if(temp1 == NULL && temp2 == NULL)
       return 1;

    /* Will reach here when one is NULL
      and other is not */
    return 0;
}

/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct 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 pochar to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 'p');
     push(&head, 'e');
```

```
    push(&head, 'e');
    push(&head, 'p');

    /* p->e->e->p */
    if(isPalindrome(head) == 1)
      printf("Linked list is Palindrome");
    else
      printf("Linked list is not Palindrome");

    getchar();
    return 0;
}
```

Time Complexity O(n)

Auxiliary Space: O(1)

**METHOD 2 (Using Recursion)**

Use two pointers left and right. Move right and left using recursion and check for following in each recursive call.

1) Sub-list is palindrome.

2) Value at current left and right are matching.

If both above conditions are true then return true.

```
#define bool int
#include<stdio.h>
#include<stdlib.h>

/* Link list node */
struct node
{
    char data;
    struct node* next;
};

bool isPalindrome(struct node **left, struct node *right)
{
   /* stop recursion here */
   if (!right)
      return true;

   /* If sub-list is not palindrome then no need to
       check for current left and right, return false */
   bool isp = isPalindrome(left, right->next);
   if (isp == false)
      return false;

   /* Check values at current left and right */
   bool isp1 = (right->data == (*left)->data);

   /* Move left to next node */
   *left = (*left)->next; /* save next pointer */

   return isp1;
}
```

```
/* UTILITY FUNCTIONS */
/* Push a node to linked list. Note that this function
  changes the head */
void push(struct node** head_ref, char new_data)
{
    /* allocate node */
    struct node* new_node =
            (struct node*) malloc(sizeof(struct 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 pochar to the new node */
    (*head_ref)    = new_node;
}

/* Drier program to test above function*/
int main()
{
    /* Start with the empty list */
    struct node* head = NULL;

     push(&head, 'r');
     push(&head, 'a');
     push(&head, 'd');
     push(&head, 'a');
     push(&head, 'r');

     /* r->a->d->a->r*/
     if(isPalindrome(&head, head) == 1)
       printf("Linked list is Palindrome");
     else
       printf("Linked list is not Palindrome");

     getchar();
     return 0;
}
```

Time Complexity: O(n)

Auxiliary Space: O(n) if Function Call Stack size is considered, otherwise O(1)

# 40.Copy a linked list with next and arbit pointer

**Question:**

You are given a Double Link List with one pointer of each node pointing to the next node just like in a single link list. The second pointer however CAN point to any node in the list and not just the previous node. Now write a program in O(n) time to duplicate this list. That is, write a program which will create a copy of this list.

Let us call the second pointer as arbit pointer as it can point to any arbitrary node in the linked list.

**1. Solution for Restricted Version of the Question:**

Let us first solve the restricted version of the original question. The restriction here is that a node will be pointed by only one arbit pointer in a linked list. In below diagram, we have obeyed the restriction.

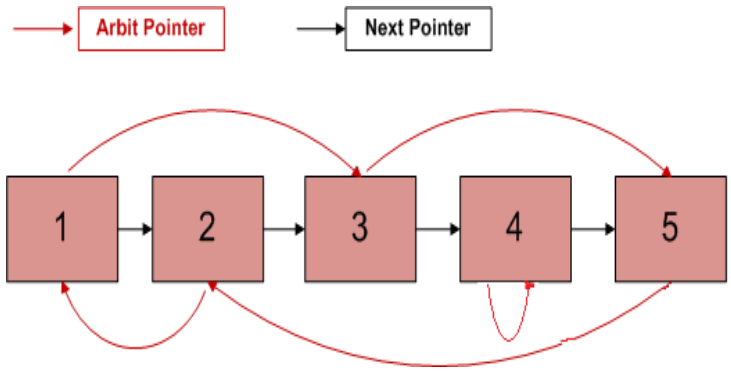



Figure 1

**Algorithm:**

1) Create all nodes in copy linked list using next pointers.

2) Change next of original linked list to the corresponding node in copy linked list.

3) Change the arbit pointer of copy linked list to point corresponding node in original linked list.

See below diagram after above three steps.

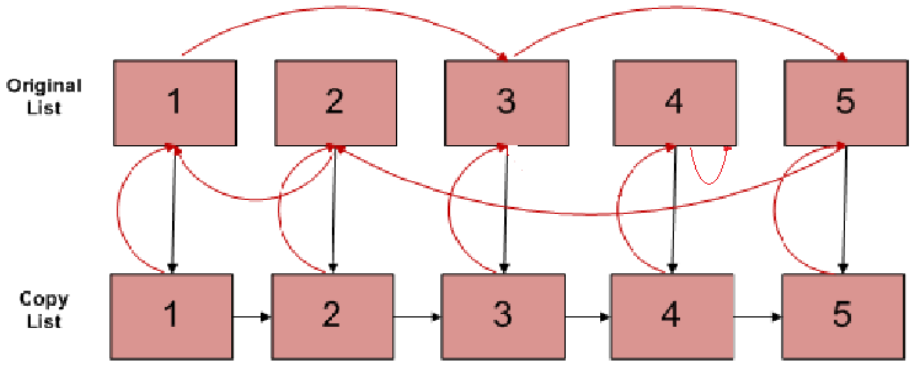



Figure 2

4) Now construct the arbit pointer in copy linked list as below and restore the next pointer of the original linked list.

```
        copy_list_node->arbit =
                        copy_list_node->arbit->arbit->next

        /* This can not be done if we remove the restriction*/
```

```
        orig_list_node->next =
                        orig_list_node->next->next->arbit

        /* After the above pointers are set properly, move next*/
        copy_list_node = copy_list_node->next
        orig_list_node = copy_list_node->arbit
```

Time Complexity:  O(n)

Auxiliary Space :  O(1)

**2. Solution for the Original Question:**

If we remove the restriction given in above solution then we CANNOT restore the next pointer of the original linked in step 4 of above solution. We have to store the node and its next pointer mapping in original linked list. Below diagram doesn't obey the restriction as node '3′ is pointed by arbit of '1′ and '4′.

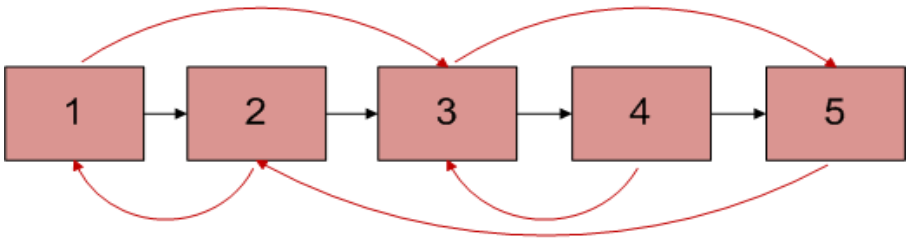



Figure 3

**Algorithm:**

1) Create all nodes in copy linked list using next pointers.

3) Store the node and its next pointer mappings of original linked list.

3) Change next of original linked list to the corresponding node in copy linked list.

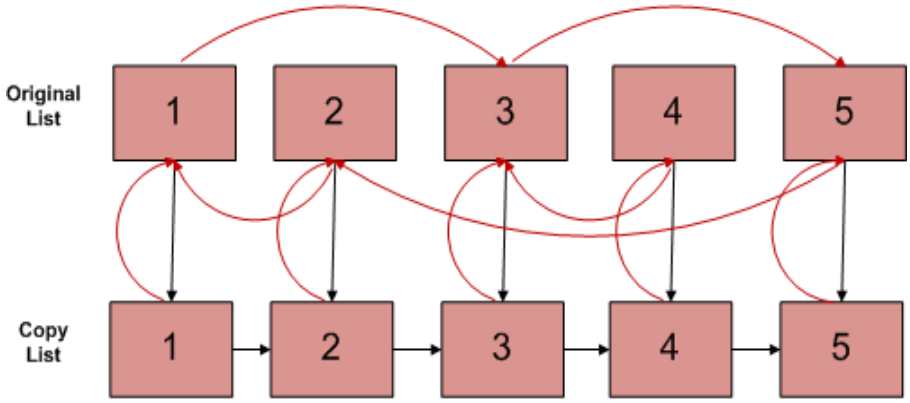



Figure 4

4) Change the arbit pointer of copy linked list to point corresponding node in original linked list.
5) Now construct the arbit pointer in copy linked list as below and restore the next pointer of the original linked list.

```
       copy_list_node->arbit =
                        copy_list_node->arbit->arbit->next
```

6) Restore the node and its next pointer of original linked list from the stored mappings(in step 2).

Time Complexity:  O(n)
Auxiliary Space:  O(n)

**3. A Constant Space and O(n) Time Solution for the Original Question:**

. This is the best solution among all three as it works in constant space and without any restriction on original linked list.

1) Create the copy of 1 and insert it between 1 & 2, create the copy of 2 and insert it between 2 & 3.. Continue in this fashion, add the copy of N to Nth node

2) Now copy the arbitrary link in this fashion

```
     original->next->arbitrary = original->arbitrary->next;
/*TRAVERSE TWO NODES*/
```

This works because original->next is nothing but copy of original and Original->arbitrary->next is nothing but copy of arbitrary.

3) Now restore the original and copy linked lists in this fashion in a single loop.

```
     original->next = original->next->next;
     copy->next = copy->next->next;
```

4) Make sure that last element of original->next is NULL.

Time Complexity: O(n)
Auxiliary Space: O(1)