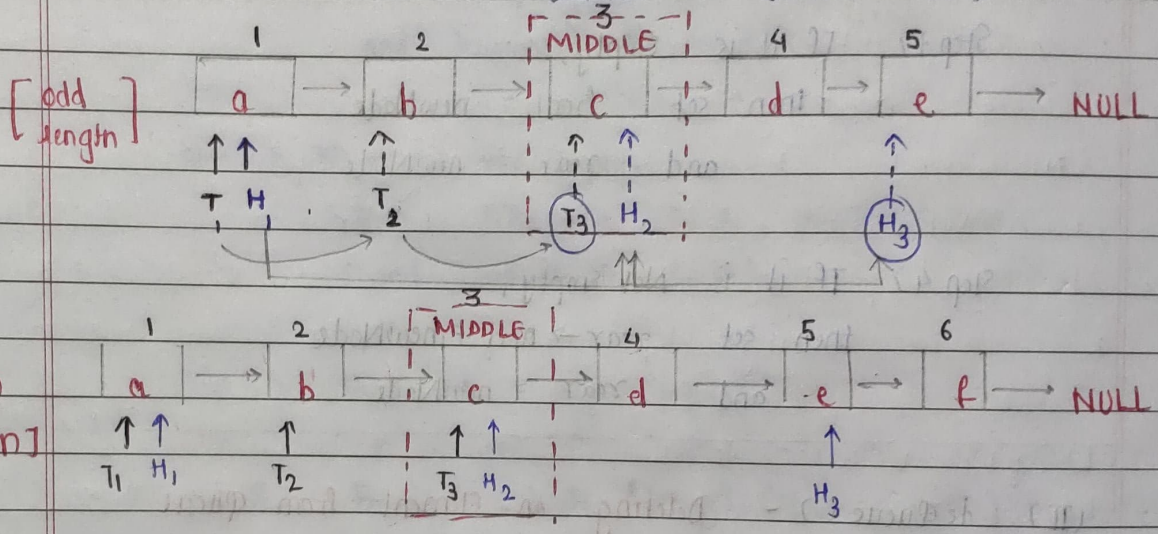
UI9CS012

TUTORIAL - II

CIRCULAR LINKED LIST

1.Y Discuss Hare and Tortoise Algorithm to find the middle point of the list

1.Y As name suggests (H) hare ← faster (2x) speed
(T) tortoise ← slower (1x)



(A) Algorithm:

1. Initialise two pointers (hare and tortoise) both pointing to head of linked list
2. Loop as long hare does not reach null
   - 2.1 Set tortoise to next node
   - 2.2 Set hare to next of next node

   End
3. After the loop ends, the node pointed by tortoise will be middle element of linked list as shown in above ~~dig~~ diagram.

(B) Implementation [C code]:

```
node * Middle_Element_Using_hare_tortoise (node *head)
{
    node *hare, *tortoise;
    hare = head;                                   } step 1
    tortoise = head;
    while ( tortoise != NULL && hare != NULL )     } step 2
    {
        tortoise = tortoise -> next;
        hare = hare -> next -> next;
    }
    return tortoise; // Middle Element of LL       } step 3
}
```

2) Write Algorithm for:

a) Traversal in Circular Linked List

Step 1> check whether list is Empty (head == NULL)

Step 2> If it is Empty, the display 'Empty List! Can't Traverse' and terminate the function

Step 3> If it is Not Empty, then define a Node pointer 'temp' and initialize with head.

Step 4> Keep displaying temp → data with an arrow (→) until temp reached to last node.

Step 5> Finally display temp → data with arrow pointing to head → data.

U19CS012

(b) Circular Linked List Insertion

(b.1) Inserting at Beginning of the List

Step 1> Create a newNode with given value.

Step 2> Check whether list is Empty (head == NULL)

Step 3> If it is empty then, set head = Newnode and newnode → next = head;

Step 4> If it is Not empty then, define a Node pointer 'temp' and initialize with 'head'

Step 5> Keep moving the 'temp' to its next node until it reaches to the last node (until temp → next == head)

Step 6> Set 'newNode → next = head', 'head = newNode' and 'temp → next = head'.

(b.2) Inserting at End of List

Step 1, 2 & 3 are same as insert at Beginning.

Step 4> If it is Not empty then, define a node pointer ('temp') and initialize with head.

Step 5> Keep moving the 'temp' to its next node until it reaches to the last node in the list (until temp → next == head)

Step 6 > ~~Ste~~ Set temp → next = newNode and
newNode → next = head

(b.3) Inserting At specific location in List (After a Node)

. Step 1, 2, 3 & 4 are same as insert at Beginning.

Step 5 > keep moving the temp to its next node until it reaches to the node after which we want to insert the newNode (temp → data is equal to location)

Step 6 > Every time check whether temp is reached to last node or not. If it is reached to last node then display 'Given node is not found in the list! Insertion not possible!' and terminate, otherwise

Step 7 > If temp is reached to exact node after which we want to insert the newNode then check whether it is last node (temp → next == head)

Step 8 > If temp is last node, then set temp → next = newNode and newNode → next = head

temp → next

Step 9 > If temp is not the last node, then set newNode → next = ^
[temp → next = newNode]

(c) Deletion in Circular Linked List

(c.1) Deletion from Beginning of the List

Step 1> Check whether list is Empty (head == NULL)

Step 2> If it is Empty, then dispay " List is Empty! Deletion not possible " and terminate this function

Step 3> If it is Not Empty, then define two Node pointers 'temp1' and 'temp2' and initialize both 'temp1' and 'temp2' with head

Step 4> Check whether list is having only one node (temp1 → next == head)

Step 5> If it is TRUE, then set head = NULL and delete temp1 (Setting Empty list condition)

Step 6> If it is FALSE, move the temp1 until it reaches to the last node (until temp1 → next == head)

Step 7> Then set head = temp2 → next temp1 → next = head and delete temp2.

(c.2) Delete from End of the List

Step 1, 2, 3, 4, 5 are same from delete from Begin.

Step 6> If it is FALSE, then set 'temp2 = temp1' and move temp1 to its next node. Repeat the same until temp1 reaches to the last node in the list. (until temp1 → next == head)

Step 7 > Set temp2 -> next = head and delete temp1.

(c.3) Deleting a Specific Node from list

Step 1, 2, 3 same as Delete from Beginning

Step 4 > Keep moving the temp1 until it reaches to the exact node to be deleted or to the last node. And every time set 'temp2 = temp1' before moving 'temp1' to its next node.

Step 5 > If it is reached to the last node then display 'Given node not found in the list! Deletion not Possible' & (terminate)

Step 6 > If it is reached to the exact node, which we want to delete, then check whether list is having only one node (temp1 -> next == head)

Step 7 > If list has only one node and that is the node to be deleted then set head = NULL and delete temp1 (free(temp1))

Step 8 > If list contains multiple nodes then check whether temp1 is the first node in the list (temp1 == head)

Step 9 > If temp1 is the first node then set temp2 = head and keep moving temp2 to its next node until temp2 reaches to last node. then set head = head -> next, temp2 -> next = head and delete temp1.

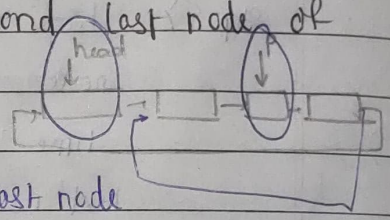
Step 10 > If temp1 is not first node then check whether it is last node in the list (temp -> next == head)

Step 11 > If temp1 is last node then set temp2 -> next = head and delete temp1 (free(temp1))

Step 12> If temp1 is not first node and not last node then set temp2 → next = temp1 → next and delete temp1 (free (temp1))

3>

(A) Write an Algorithm to exchange first and last nodes of circular linked list



TASK 1 : Find pointer to previous of last node

Step 1> Initialize a pointer 'p' to head

Step 2> Loop / Iterate the list till it reaches to Node previous to last Node ie (p → next → next != NULL)

TASK 2 : To exchange ~~head and~~ first and last node using head and p.

Step 3 :> p → next → next = head → next ;
head → next = p → next ;
p → next = head ;
head = head → next ;

} Link management

(B) Write an Algorithm to delete every alternate node of circular linked list

[or only one element]

Step 1 > check if list is empty (head == NULL), terminate & return back

Step 2.> Initialize two Node pointers *prev = head
*node = head → next

Step 3.) Iterate till (prev != NULL && node != NULL)

(3.1) Change next link of previous node

prev->next = node->next

(3.2) Free that node

free(node);

(3.3) Update prev and Node

prev = prev->next;

if (prev != NULL)

node = prev->next

(c) Split the Circular Linked List

Step 1> Store mid and last pointers of circular linked list using tortoise and hare algorithm.

Step 2.) Make second half circular

Step 3.> Make first half circular

Step 4.) Set head (or start) pointer of two linked list

if no. of nodes are odd => [first list will have one extra node]

[Implementation in Code]

x

TUTORIAL IX:
Circular Linked List Implementation
U19CS012 [D-12]

Implement the following operations in context to ***Circular linked list***:

1) Creation

2) Insertion (at beginning, middle and end)

3) Deletion (from beginning, middle and end)

----------Additional Functions in Assignment----------

4) Exchange First and Last Node of List

5) Delete Alternating Nodes of Circular Linked List

6.) Split Linked List into Two Halves

Code:

```c
// Implement the following operations in context to Circular linked list:
// 1) Creation
// 2) Insertion (at begining, middle and end)
// 3) Deletion (from begining, middle and end)

#include <stdio.h>
// For Exit Function
#include <stdlib.h>

// Structure for Each Node
struct node
{
    int data;
    struct node *next;
};

//Helper Functions

// 1 -> Creation of Circular Linked List

// Creation of the Circular Linked List
void CREATION_CLL();
// Display of the Whole Circular Linked List
void DISPLAY_CLL();
// Returns the Length of Circular Linked List
int LENGTH_CLL();
```

```c
// 2 -> Insertion in Circular Linked List

// Insert at the Beginning of Circular Linked List
void Insert_Begin();
// Insert at the End of Circular Linked List
void Insert_Last();
// Insert in the Middle Of the Circular Linked List
void Insert_Middle();

// 3 -> Deletion in the Circular Linked List

// Delete at the Beginning of Circular Linked List
void Delete_Begin();
// Delete at the End of Circular Linked List
void Delete_Last();
// Delete in the Middle Of the Circular Linked List
void Delete_Middle();
// 1 -> Deletes Node at Particular Position
void Delete_Position();
// 2 -> Deletes all Nodes with Particular Value
void Delete_Value();

// Exchange the First and Last Node
struct node *Exchange_First_And_Last(struct node *head);

//Delete Alternating Node
void Delete_Alternating_Node(struct node *head);

// Split List into Two Halves
void Split_CLL(struct node *head);

// head Pointer -> head of Linked List
struct node *head = NULL;

// For Splitted Head Pointers
struct node *head1 = NULL;
struct node *head2 = NULL;

int main()
{
    int choice;
    printf("\nCIRCULAR LINKED LIST\n");

    printf(" 1 -> Create a Circular Linked List\n");
    printf(" 2 -> Display the Circular Linked List\n");
    printf(" 3 -> Insert at the Beginning of Circular Linked List\n");
    printf(" 4 -> Insert at the End of Circular Linked List\n");
    printf(" 5 -> Insert at Middle of Circular Linked List\n");
    printf(" 6 -> Delete from Beginning\n");
    printf(" 7 -> Delete from the End\n");
```

```
    printf(" 8 -> Delete at Middle of Circular Linked List\n");
    printf(" 9 -> Exchange First and Last Node of List\n");
    printf(" 10 -> Delete Alternating Nodes of Circular Linked List\n");
    printf(" 11 -> Split Linked List into Two Halves\n");
    printf(" 12 -> Exit\n");

    while (1)
    {
        printf("Enter your choice : ");
        scanf("%d", &choice);

        switch (choice)
        {
        case 1:
            CREATION_CLL();
            break;
        case 2:
            DISPLAY_CLL(head);
            break;
        case 3:
            Insert_Begin();
            break;
        case 4:
            Insert_Last();
            break;
        case 5:
            // Insert at Middle of Circular LL
            Insert_Middle();
            break;
        case 6:
            Delete_Begin();
            break;
        case 7:
            Delete_Last();
            break;
        case 8:
            // Delete at Middle of Circular LL
            Delete_Middle();
            break;
        case 9:
            head = Exchange_First_And_Last(head);
            break;
        case 10:
            Delete_Alternating_Node(head);
            break;
        case 11:
            Split_CLL(head);
            printf("The Splitted Lists Are :\n");
            DISPLAY_CLL(head1);
            DISPLAY_CLL(head2);
```

```
            break;
        case 12:
            exit(0);
            break;

        default:
            printf("Enter a Valid Choice!");
            break;
        }
    }
    return 0;
}

// Creation of the Circular Linked List
void CREATION_CLL()
{
    struct node *ptr, *temp;
    int item;

    // Allocate Memory for One Node
    ptr = (struct node *)malloc(sizeof(struct node));

    if (ptr == NULL)
    {
        printf("No Memory Space on Device!\n");
        return;
    }
    else
    {
        // Creation of New Node { [?]->NULL }
        printf("Enter the Data to be stored in Node : ");
        scanf("%d", &item);
        ptr->data = item;

        if (head == NULL)
        {
            // If the Linked List is Empty
            head = ptr;
            // Since its Circular Linked List
            ptr->next = head;
        }
        else
        {
            // If the Linked List is Not Empty
            temp = head;

            // Traverse and Point temp to Rear of Circular Link List
            while (temp->next != head)
                temp = temp->next;
```

```
            // Node should Point to head
            ptr->next = head;
            // Rear Element next should Point to "New Node"
            temp->next = ptr;
            // Since "New Node" is Our New Head [Start]
            head = ptr;
        }
        // printf("Node Inserted!\n");
    }
}

// Display of the Whole Circular Linked List
void DISPLAY_CLL(struct node *h1)
{
    struct node *ptr;

    if (h1 == NULL)
    {
        printf("List is Empty!!\n");
        return;
    }
    else
    {
        // Head Pointer
        ptr = h1;

        printf("Elements of List : ");

        while (ptr->next != h1)
        {

            printf("%d -> ", ptr->data);
            ptr = ptr->next;
        }
        printf("%d -> ", ptr->data);
        printf("HEAD\n");
    }
}

// Returns the Length of Circular Linked List
int LENGTH_CLL()
{
    struct node *ptr;

    if (head == NULL)
    {
        return 0;
    }
    else
    {
```

```
        // Head Pointer
        int cnt = 0;
        ptr = head;
        while (ptr->next != head)
        {
            cnt++;
            ptr = ptr->next;
        }
        return cnt + 1;
    }
}

// Insert at the Beginning of Circular Linked List
void Insert_Begin()
{
    struct node *ptr, *temp;
    int item;

    // Allocate Memory for One Node
    ptr = (struct node *)malloc(sizeof(struct node));

    if (ptr == NULL)
    {
        printf("No Memory Space on Device!\n");
        return;
    }
    else
    {
        // Creation of New Node { [?]->NULL }
        printf("Enter the Data to be stored in Node : ");
        scanf("%d", &item);
        ptr->data = item;

        if (head == NULL)
        {
            // If the Linked List is Empty
            head = ptr;
            // Since its Circular Linked List
            ptr->next = head;
        }
        else
        {
            // If the Linked List is Not Empty
            temp = head;

            // Traverse and Point temp to Rear of Circular Link List
            while (temp->next != head)
                temp = temp->next;

            // Node should Point to head
```

```
            ptr->next = head;
            // Rear Element next should Point to "New Node"
            temp->next = ptr;
            // Since "New Node" is Our New Head [Start]
            head = ptr;
        }
        // printf("Node Inserted!\n");
    }
}

// Insert at the End of Circular Linked List
void Insert_Last()
{
    struct node *ptr, *temp;
    int item;

    // Allocate Memory for One Node
    ptr = (struct node *)malloc(sizeof(struct node));

    if (ptr == NULL)
    {
        printf("No Memory Space on Device!\n");
        return;
    }
    else
    {
        // Creation of New Node { [?]->NULL }
        printf("Enter the Data to be stored in Node : ");
        scanf("%d", &item);
        ptr->data = item;

        if (head == NULL)
        {
            // If the Linked List is Empty
            head = ptr;
            // Since its Circular Linked List
            ptr->next = head;
        }
        else
        {
            // If the Linked List is Not Empty
            temp = head;

            // Traverse and Point temp to Rear of Circular Link List
            while (temp->next != head)
                temp = temp->next;

            // Rear Element next shoudl Point to "New Node"
            temp->next = ptr;
```

```
            // Node should Point to head
            ptr->next = head;
        }
        // printf("Node Inserted!\n");
    }
}

// Delete at the Beginning of Circular Linked List
void Delete_Begin()
{
    // temporary pointer to store old head
    struct node *ptr;

    if (head == NULL)
    {
        printf("List is Empty! No Deletion Possible!!\n");
        return;
    }
    else if (head->next == head)
    {
        // Only One Element in Circular Linked List
        head = NULL;
        free(head);
        printf("Node Deleted!\n");
    }
    else
    {
        ptr = head;

        // Traverse and Point ptr to Rear of Circular Link List
        while (ptr->next != head)
            ptr = ptr->next;

        // Point the Rear Element of CLL to Second Element of CLL
        ptr->next = head->next;
        // free(head);
        // head is pointing to Second Element of CLL
        head = ptr->next;
        printf("Node Deleted!\n");
    }
}

// Delete at the End of Circular Linked List
void Delete_Last()
{
    struct node *ptr, *preptr;

    if (head == NULL)
    {
        printf("List is Empty! No Deletion Possible!!\n");
```

```
        return;
    }
    else if (head->next == head)
    {
        // Only One Element in Circular Linked List
        head = NULL;
        free(head);
        printf("Node Deleted!\n");
    }
    else
    {
        ptr = head;

        // Traverse and Point ptr to Rear of Circular Link List
        // Traverse and Point preptr to One Element Before Rear of Circular Link List
        while (ptr->next != head)
        {
            preptr = ptr;
            ptr = ptr->next;
        }

        // Second Rear Element Should Point to Next of Rear
        // Therby Deleting Rear Element of CLL
        preptr->next = ptr->next;

        // free(ptr);

        printf("Node Deleted!\n");
    }
}

void display()
{
    struct node *ptr;
    ptr = head;
    if (head == NULL)
    {
        printf("\nnothing to print");
    }
    else
    {
        printf("\n printing values ... \n");

        while (ptr->next != head)
        {

            printf("%d\n", ptr->data);
            ptr = ptr->next;
        }
        printf("%d\n", ptr->data);
```

```c
    }
}

// Insertion at Middle of Linked List
void Insert_Middle()
{
    struct node *ptr, *temp;

    int i, pos;

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

    if (temp == NULL)
    {
        printf("No Memory Space on Device!\n");
        return;
    }

    printf("Enter the Position for the New Node to be Inserted : ");
    scanf("%d", &pos);

    // pos = 1 -> Insertion at Beginning of LL
    // pos = len + 1 -> Insertion at Ending of LL

    // Length of the Linked Lst
    int len = LENGTH_CLL();

    if (pos <= 0 || pos > len + 1)
    {
        printf("Enter Valid Postion for Insertion!\n");
        return;
    }

    // Creation of New Node { [?]->NULL }
    printf("Enter the Data to be stored in Node : ");
    scanf("%d", &temp->data);
    temp->next = NULL;

    if (pos == 1)
    {
        // At the Beginning of Linked List
        temp->next = head;
        head = temp;
    }
    else
    {
        for (i = 1, ptr = head; i < pos - 1; i++)
        {
            ptr = ptr->next;
        }
```

```
        // temp is also pointing to next of "pos" to be inserted
        temp->next = ptr->next;
        // Make ptr Point to Temp
        ptr->next = temp;
    }
}

// Delete in the Middle Of the Circular Linked List
void Delete_Middle()
{
    if (head == NULL)
    {
        printf("List is Empty! No Deletion Possible!!\n");
        exit(0);
    }
    else
    {
        int ch = 0;
        printf("Delete A Node By : \n");
        printf(" 1 -> Position\n");
        printf(" 2 -> Value\n");
        printf("Enter Your Choice : ");
        scanf("%d", &ch);

        switch (ch)
        {
        case 1:
            Delete_Position();
            break;
        case 2:
            Delete_Value();
            break;
        default:
            printf("Enter a Valid Choice!\n");
            break;
        }
    }
}

// 1 -> Deletes Node at Particular Position
void Delete_Position()
{
    int i, pos;
    struct node *temp, *ptr;

    printf("Enter the Position of the Node to be Deleted : ");
    scanf("%d", &pos);

    // pos = 1 -> Deletion at Beginning of LL
    // pos = len -> Deletion at Ending of LL
```

```c
    // Length of the Circular Linked Lst
    int len = LENGTH_CLL();
    // printf("LENGTH : %d\n", len);

    if (pos <= 0 || pos > len)
    {
        printf("Enter Valid Postion for Deletion!\n");
        return;
    }

    if (pos == 1)
    {
        Delete_Begin();
        return;
    }
    else
    {
        if (pos == len)
        {
            Delete_Last();
            return;
        }

        ptr = head;
        for (i = 1; i < pos; i++)
        {
            temp = ptr;
            ptr = ptr->next;
        }
        // point the prev of {element to be deleted} to "next of deleted"
        // []     []     []
        //temp    ptr
        temp->next = ptr->next;

        printf("The Deleted Element is : %d\n", ptr->data);
        free(ptr);
    }
}

// 2 -> Deletes all Nodes with Particular Value
void Delete_Value()
{
    int value;
    struct node *temp, *ptr;

    printf("Enter the Value of the Node to be Deleted : ");
    scanf("%d", &value);

    int flag = 0;
```

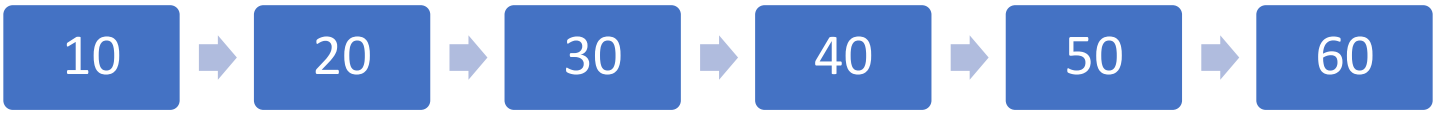
```
if (head == NULL)
{
    printf("List is Empty!No Deletions Possible\n");
    return;
}
else
{
    // Head Pointer
    ptr = head;

    while (ptr->next != head)
    {

        // If the Value of Node = Value of Node to be Deleted
        if (ptr->data == value)
        {
            if (ptr == head)
            {
                Delete_Begin();
                flag = 1;
            }
            else
            {
                if (ptr->next == head)
                {
                    Delete_Last();
                    flag = 1;
                }
                else
                {
                    temp->next = ptr->next;
                    flag = 1;
                }
                // printf("The Deleted Element is : %d\n", ptr->data);
            }
        }
        // temp stored old node's address
        temp = ptr;
        // ptr now points to next node
        ptr = ptr->next;
    }

    if (flag == 0)
    {
        printf("Node with Given Value Does Not Exist! OR Deleted Earlier!\n");
    }
    else
    {
        printf("Node with Given Value Found and Deleted Succesfully!\n");
```

```
        }
    }
}

// Exchange the First and Last Node
struct node *Exchange_First_And_Last(struct node *head)
{
    // TASK 1 : Find pointer to previous of last node
    // Declare a tmp Pointer
    struct node *tmp = head;
    // Iterate till it Points to the Previous of the Last Node
    while (tmp->next->next != head)
        tmp = tmp->next;

    //  Exchange first and last nodes using head and tmp
    // Link Allocation
    tmp->next->next = head->next;
    head->next = tmp->next;
    tmp->next = head;
    head = head->next;

    return head;
}

//Delete Alternating Node
void Delete_Alternating_Node(struct node *head)
{
    if (head == NULL)
        return;

    int length = LENGTH_CLL();

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

    while (prev->next != head && tmpnode->next != head)
    {
        // Change next link of previous node by Skipping its Next Node
        prev->next = tmpnode->next;

        // Free memory
        free(tmpnode);

        // Update prev and node
        prev = prev->next;
        tmpnode = prev->next;
    }

    if (length % 2 == 0)
```

```
    {
        // Last Node Needs to deleted in Even Length CLL
        Delete_Last();
    }
    else
    {
        printf("Nodes Deleted!\n");
    }
}

// Split List into Two Halves
void Split_CLL(struct node *head)
{
    struct node *tortoise = head;
    struct node *hare = head;

    if (head == NULL)
        return;

    while (hare->next != head && hare->next->next != head)
    {
        hare = hare->next->next;
        tortoise = tortoise->next;
    }

    // Even Elements in the Linked List
    if (hare->next->next == head)
        hare = hare->next;

    // Set the head pointer of first half
    head1 = head;

    // Set the head pointer of second half
    if (head->next != head)
        head2 = tortoise->next;

    // Make second half circular
    hare->next = tortoise->next;

    // Make first half circular
    tortoise->next = head;
}
```

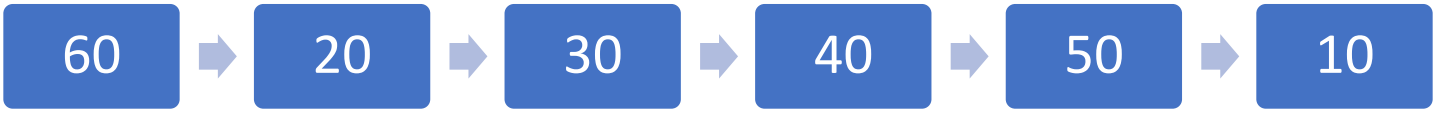
Test Cases:

A.) Creation of Circular Linked List



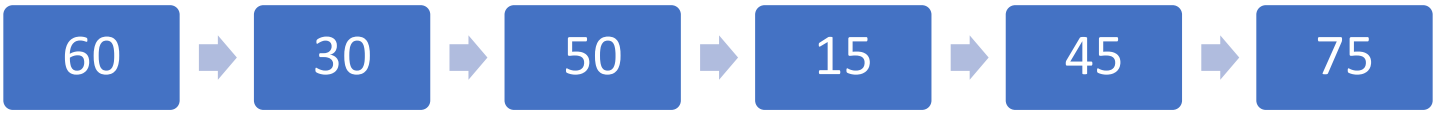

```
CIRCULAR LINKED LIST
 1 -> Create a Circular Linked List
 2 -> Display the Circular Linked List
 3 -> Insert at the Beginning of Circular Linked List
 4 -> Insert at the End of Circular Linked List
 5 -> Insert at Middle of Circular Linked List
 6 -> Delete from Beginning
 7 -> Delete from the End
 8 -> Delete at Middle of Circular Linked List
 9 -> Exchange First and Last Node of List
 10 -> Delete Alternating Nodes of Circular Linked List
 11 -> Split Linked List into Two Halves
 12 -> Exit
Enter your choice : 1
Enter the Data to be stored in Node : 10
Enter your choice : 4
Enter the Data to be stored in Node : 20
Enter your choice : 4
Enter the Data to be stored in Node : 30
Enter your choice : 4
Enter the Data to be stored in Node : 40
Enter your choice : 4
Enter the Data to be stored in Node : 50
Enter your choice : 4
Enter the Data to be stored in Node : 60
Enter your choice : 2
Elements of List : 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> HEAD
```

B.) Exchange First and Last Node of Circular Linked List

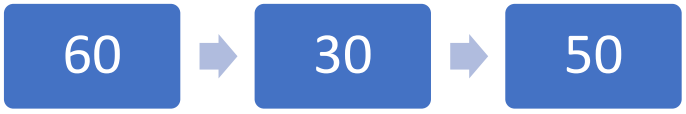


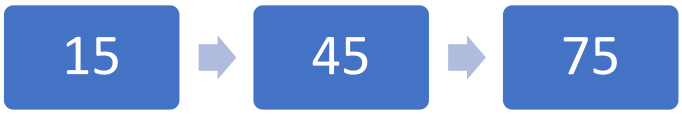
C.) Delete Alternating Nodes of Circular Linked List

D.) Exchange First and Last Node of Circular Linked List
[Added 3 More Elements in Circular Linked List]



Splitted Circular Linked List into Two Halves:





```
Enter your choice : 2
Elements of List : 10 -> 20 -> 30 -> 40 -> 50 -> 60 -> HEAD
Enter your choice : 9
Enter your choice : 2
Elements of List : 60 -> 20 -> 30 -> 40 -> 50 -> 10 -> HEAD
Enter your choice : 10
Node Deleted!
Enter your choice : 2
Elements of List : 60 -> 30 -> 50 -> HEAD
Enter your choice : 4
Enter the Data to be stored in Node : 15
Enter your choice : 4
Enter the Data to be stored in Node : 45
Enter your choice : 4
Enter the Data to be stored in Node : 75
Enter your choice : 2
Elements of List : 60 -> 30 -> 50 -> 15 -> 45 -> 75 -> HEAD
Enter your choice : 11
The Splitted Lists Are :
Elements of List : 60 -> 30 -> 50 -> HEAD
Elements of List : 15 -> 45 -> 75 -> HEAD
Enter your choice : 12
```