```
void DeleteFrontNodeFromCLL (struct CLLNode **head) {
        struct CLLNode *temp = *head;
        struct CLLNode *current = *head;
        if(*head == NULL) {
                printf("List Empty"); return;
        }
        while (current→next != *head)
                current = current→next;
        current→ next = *head→next;
        *head = *head→next;
        free(temp);
        return;
}
```

Time Complexity: $O(n)$, for scanning the complete list of size n. Space Complexity: $O(1)$, for creating one temporary variable.

Applications of Circular List

Circular linked lists are used in managing the computing resources of a computer. We can use circular lists for implementing stacks and queues.

3.9 A Memory-Efficient Doubly Linked List

In conventional implementation, we need to keep a forward pointer to the next item on the list and a backward pointer to the previous item. That means, elements in doubly linked list implementations consists of data, a pointer to the next node and a pointer to the previous node in the list as shown below.

Conventional Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * prev;
        struct ListNode * next;
};
```

Recently a journal (Sinha) presented an alternative implementation of the doubly linked list ADT, with insertion, traversal and deletion operations. This implementation is based on pointer difference. Each node uses only one pointer field to traverse the list back and forth.

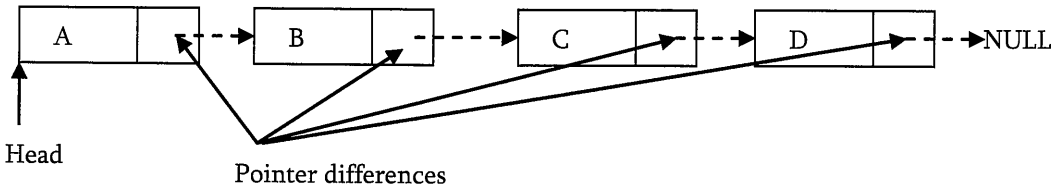
New Node Definition

```
typedef struct ListNode {
        int data;
        struct ListNode * ptrdiff;
};
```

The *ptrdiff* pointer field contains the difference between the pointer to the next node and the pointer to the previous node. Pointer difference is calculated by using exclusive-or ($\oplus$) operation.

$$ptrdiff = pointer\ to\ previous\ node \oplus pointer\ to\ next\ node.$$

The *ptrdiff* of the start node (head node) is the $\oplus$ of NULL and *next* node (next node to head). Similarly, the *ptrdiff* of end node is the $\oplus$ of *previous* node (previous to end node) and NULL. As an example, consider the following linked list.



In the above example,

- The next pointer of A is: NULL ⊕ B
- The next pointer of B is: A ⊕ C
- The next pointer of C is: B ⊕ D
- The next pointer of D is: C ⊕ NULL

Why does it work?

To have answer for this question let us consider the properties of ⊕:

X ⊕ X = 0
X ⊕ 0 = X
X ⊕ Y = Y ⊕ X (symmetric)
(X ⊕ Y) ⊕ Z = X ⊕ (Y ⊕ Z) (transitive)

For the above example, let us assume that we are at C node and want to move to B. We know that Cs *ptrdiff* is defined as B ⊕ D. If we want to move to B, performing ⊕ on Cs *ptrdiff* with D would give B. This is due to fact that,

(B ⊕ D) ⊕ D = B (since, D ⊕ D=0)

Similarly, if we want to move to D, then we have to applying ⊕ to Cs *ptrdiff* with B would give D.

(B ⊕ D) ⊕ B = D (since, B ⊕ B=0)

From the above discussion we can see that just by using single pointer, we are able to move back and forth. A memory-efficient implementation of a doubly linked list is possible to have without compromising much timing efficiency.

3.10 Problems on Linked Lists

Problem-1 Implement Stack using Linked List

Solution: Refer *Stacks* chapter.

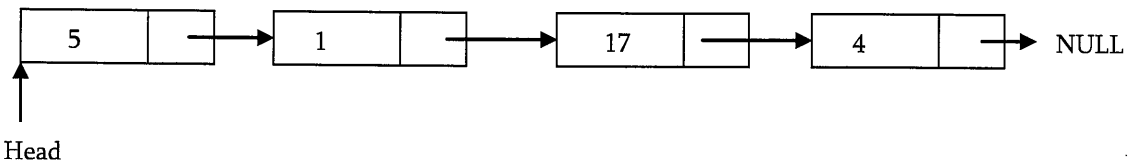
Problem-2 Find n^{th} node from the end of a Linked List.

Solution: Brute-Force Method: Start with the first node and count how many nodes are there after that node. If the number of nodes are $< n-1$ then return saying "fewer number of nodes in the list". If the number of nodes are $> n-1$ then go to next node. Continue this until the numbers of nodes after current node are $n-1$.

Time Complexity: $O(n^2)$, for scanning the remaining list (from current node) for each node. Space Complexity: O(1).

Problem-3 Can we improve the complexity of Problem-2?

Solution: Yes, using hash table. As an example consider the following list.



In this approach, create a hash table whose entries are < *position of node, node address* >. That means, key is the position of the node in the list and value is the address of that node.