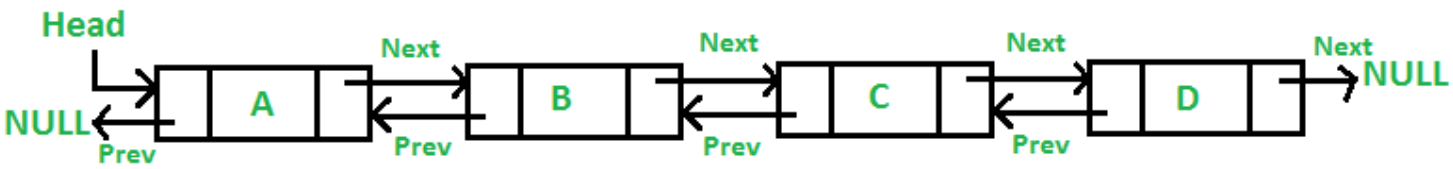
XOR Linked Lists

XOR Linked Lists are Memory Efficient implementation of *Doubly Linked Lists*. 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 the address field with every node. This memory efficient Doubly Linked List is called XOR Linked List or Memory Efficient Linked List 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 Efficient) 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 as ***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 directions. 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 the 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 the next node. Similarly, we can traverse the list in a backward direction.

C++

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

class Node {
    public:
        int data;
```

```
        struct Node* npx = 0;

        Node( int val ) {
            this -> data = val;
        }
};

// Function for insertion into Linked List
void insert( Node* &head, Node* &curr, int x ) {
    Node* newNode = new Node( x );

    if( head == NULL ) {
        head = newNode;
        curr = newNode;
    } else {
        // uintptr is used for casting pointers when we want to perform address arithmetic.
        curr -> npx = (Node*)( ( uintptr_t ) curr -> npx ^ ( uintptr_t ) newNode );
        newNode -> npx = curr;
        curr = newNode;
    }
    return;
}

// Function to pront the linked list
void print( Node* head ) {
  Node* temp = head;
  Node* prev = NULL;
    while( temp  ) {
        cout << temp -> data << " ";
```

```
        temp = ( Node* ) ( (uintptr_t) prev ^ ( uintptr_t ) temp -> npx );
        prev = current;

    }
    return;
}

int main() {
    Node* head = NULL, *curr = NULL;

    insert( head, curr, 1 );
    insert( head, curr, 2 );
    insert( head, curr, 3 );
    insert( head, curr, 4 );
    insert( head, curr, 5 );

    print( head ); // Print the linked list
    return 0;
}
```

Output

```
1 2 3 4 5
```

Mark as Read

Report An Issue

If you are facing any issue on this page. Please let us know.