**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. It is represented as follows in the image below. From the below image, it can be depicted out that the address of the previous node is retained and carried over for computation by the previous pointer while that of the next node is after pointers similarly.



Now there is a memory-efficient version of Doubly Linked List that 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 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. 

Diagram

Description automatically generated

Now here we will be discussing out both the ways in order to perch out how XOR representation behaves differently from ordinary representation.

1. Ordinary Representation
2. XOR List Representation

**Way 1: 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

**Way 2: XOR List Representation**

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

While traversing 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).*

**Illustration:**

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

npx(C) XOR add(B)

=> (add(B) XOR add(D)) XOR add(B) // npx(C) = add(B) XOR add(D)

=> add(B) XOR add(D) XOR add(B) // a^b = b^a and (a^b)^c = a^(b^c)

=> add(D) XOR 0 // a^a = 0

=> add(D) // a^0 = a

Similarly, we can traverse the list in the backward direction. Now straightaway coming down to the implementation part in order to figure out better.

1. **Traversing the list from left to right:**

Since we are traversing the list from left to right, say we store the previous node’s address in some variable. As the previous node information is available, we can get the next node’s address by XOR ing the value in the link field with the previous node’s address.

For example, suppose we are at node , we can get the address of node D , as shown below.

addr(B) ^ nxp(C)  
= addr(B) ^ (addr(B) ^ addr(D))  
= 0 ^ addr(D)  
= addr(D)

The XOR operation cancels addr(B) appearing twice in the equation, and all we are left with is the addr(D) . Similarly, to get the address of the first node A in the list, we can XOR the value in the link field with NULL.

NULL ^ link(A)  
= NULL ^ (NULL ^ addr(B))  
= 0 ^ addr(B)  
= addr(B)

1. **Traversing the list from right to left:**

Following a similar logic, to get the address of the last node D in the list, XOR the D's link field value with NULL.

NULL ^ link(D)  
= NULL ^ (addr(C) ^ NULL)  
= 0 ^ addr(C)  
= addr(C)

**Drawbacks of XOR linked list:**

An XOR linked list is similar to a doubly-linked list but not completely equivalent to a doubly-linked list. There are several disadvantages of an XOR linked list over a doubly-linked list, which is discussed below:

1. A doubly-linked list is easy to code and maintain, but it is a little complex for an XOR linked list.
2. XOR linked list is not supported by several languages such as Java, where conversion between pointers and integers is undefined.
3. If a pointer to an existing middle node in an XOR linked list is provided, we can’t delete that node from the list or insert a new node before or after it. On the other hand, this can be done easily with a doubly-linked list.

Consider the following program, which constructs an XOR linked list and traverses it in a forward direction using bitwise XOR operator properties. To traverse the complete list, maintain three-pointers curr, prev, and next to store the current node address, the previous node address, and the next node address, respectively. Each iteration of the loop moves these pointers one position forward or backward depending upon which direction we are traversing the list.

**C Implementation**

|  |
| --- |
| #include <stdio.h>  #include <stdlib.h>  #include <stdint.h>    // Data structure to store a XOR linked list node  struct Node  {      int data;      struct Node\* link;  };    // Helper function to return XOR of `x` and `y`  struct Node\* XOR(struct Node \*x, struct Node \*y) {      return (struct Node\*)((uintptr\_t)(x) ^ (uintptr\_t)(y));  }    // Helper function to traverse the list in a forward direction  void traverse(struct Node \*head)  {      struct Node\* curr = head;      struct Node\* prev = NULL;      struct Node \*next;        while (curr != NULL)      {          printf("%d —> ", curr->data);            // `next` node would be xor of the address of the previous node          // and current node link          next = XOR(prev, curr->link);            // update `prev` and `curr` pointers for the next iteration of the loop          prev = curr;          curr = next;      }        printf("NULL");  }    // Helper function to insert a node at the beginning of the XOR linked list  void push(struct Node \*\*head, int data)  {      // allocate a new list node and set its data      struct Node\* newNode = (struct Node\*)malloc(sizeof(struct Node));      newNode->data = data;        // The link field of the new node is XOR of the current head and `NULL`      // since a new node is being inserted at the beginning      newNode->link = XOR(\*head, NULL);        // update link value of the current head node if the linked list is not empty      if (\*head)      {          // \*(head)->link is XOR of `NULL` and address of the next node.          // To get the address of the next node, XOR it with `NULL`          (\*head)->link = XOR(newNode, XOR((\*head)->link, NULL));      }        // update the head pointer      \*head = newNode;  }    int main(void)  {      // input keys      int keys[] = { 1, 2, 3, 4, 5 };      int n = sizeof(keys)/sizeof(keys[0]);        struct Node\* head = NULL;      for (int i = n - 1; i >=0; i--) {          push(&head, keys[i]);      }        traverse(head);        return 0;  } |

**Output:**  
  
1 —> 2 —> 3 —> 4 —> 5 —> NULL

**C++ Implementation**

|  |
| --- |
| #include <iostream>  #include <vector>  #include <cstdint>  using namespace std;    // Data structure to store a XOR linked list node  struct Node  {      int data;      Node\* link;  };    // Helper function to return XOR of `x` and `y`  Node\* XOR(Node\* x, Node\* y) {      return (Node\*)((uintptr\_t)(x) ^ (uintptr\_t)(y));  }    // Helper function to traverse the list in a forward direction  void traverse(Node\* head)  {      Node\* curr = head;      Node\* prev = nullptr;      Node \*next;        while (curr != nullptr)      {          cout << curr->data << " —> ";            // `next` node would be xor of the address of the previous node          // and current node link          next = XOR(prev, curr->link);            // update `prev` and `curr` pointers for the next iteration of the loop          prev = curr;          curr = next;      }        cout << "nullptr";  }    // Helper function to insert a node at the beginning of the XOR linked list  void push(Node\* &headRef, int data)  {      // allocate a new list node and set its data      Node\* newNode = new Node();      newNode->data = data;        // The link field of the new node is XOR of the current head and nullptr      // since a new node is being inserted at the beginning      newNode->link = XOR(headRef, nullptr);        // update link value of the current head node if the linked list is not empty      if (headRef)      {          // `headRef->link` is XOR of null and address of the next node.          // To get the address of the next node, XOR it with nullptr          headRef->link = XOR(newNode, XOR(headRef->link, nullptr));      }        // update the head pointer      headRef = newNode;  }    int main()  {      // input keys      vector<int> keys = { 1, 2, 3, 4, 5 };        Node\* head = nullptr;      for (int i = keys.size() - 1; i >=0; i--) {          push(head, keys[i]);      }        traverse(head);        return 0;  } |

**Output:**  
  
1 —> 2 —> 3 —> 4 —> 5 —> nullptr