DEQUE USING MEMORY EFFICIENT DOUBLE LINKED LIST

➢ Team Members Details:

1. 19H51A0437-Ravali
   Branch: ECE
   Email-id: 19h51a0437@cmrcet.ac.in
2. 19H51A04F1-Anusha Yadav
   Branch: ECE
   Email-id: 19h51a04f1@cmrcet.ac.in
3. 19H51A04g0-Jakata Meghana
   Branch: ECE
   Email-id: 19h51a04g0@cmrcet.ac.in
4. 19H51A04g4-Kusuma Ruchitha
   Branch: ECE
   Email-id: 19h51a04g4@cmrcet.ac.in
5. 19H51A05H5-Miryala Sathvika
   Branch: CSE
   Email-id: 19h51a05h5@cmrcet.ac.in

➢ Mentors:

Mr Ashutosh Akhouri Sir

➢ Semester/Year:

IV Sem- II Year

Abstract-

An ordinary double linked list requires space for two address fields to store previous and next pointers. Instead of using two pointers, we are implementing deque using single pointer of the double linked list. This process is also referred to as Memory Efficient Doubly linked list or XOR linked list. It uses bitwise XOR operation. In XOR linked list instead of storing actual memory addresses, every node stores the XOR of address of previous and next nodes. The purpose of XOR linked list is to save space for one address. In XOR linked list we can traverse in both forward and reverse directions.

Advantages:

1. Reduces memory usage
2. Another advantage which might be useful is the ease of reversability.
3. Reversing a doubly linked list invloves updating of all the prev and next values in all nodes. But we don't have these pointers anymore as they are XOR-ed into one value. That means that we do not have to change anything, only the head pointer need to be changed to point to last node.

Applications:

1. XOR/Memory Efficient linked list have same functionality as double linked list, they just need one one cell for both next and previous node link, so what ever real life usage of double linked list also applied to XORed linked list too. Some of them are 1) Used for implementing the backward and forward navigation of visited web pages i.e. back and forward button.
2. Used in applications that have a Most Recently Used

Table of Contents

Project Description

1.1 Purpose of the project

The purpose of this project is to implement a deque(which can be used as stack and queue) using memory efficient double linked list.

1.2 Goals/requirements

The requirements in this project is we implement using a single pointer so that the space complexity reduces. With the help of XOR concept i.e, xor of next and prev pointer addresses we get the current node to npx's address through double linked list. The goal is to get a memory efficient code with less space complexity.

1.3 Methodology

Deque can be implemented by circular array and double linked list.

1.3.1 Alternative approaches:

Circular array in which the last element of the array is connected to the first element of the array.

Double linked list: for implementing deque, we need to keep track of two pointers, **front** and **rear**. We **enqueue (push)** an item at the rear or the front end of deque
and **dequeue(pop)** an item from both rear and front end.

Memory efficient double linked list: In this we use single pointer instead of next and prev addresses using XOR concept .

1.3.2 Current approach:

Implementing deque using memory efficient double linked list with the help of XOR concept.

1.3.3 Description of current approach:

Implementing deque using single pointer of the double linked list. This process is also referred to as Memory Efficient Doubly linked list or XOR linked list. It uses bitwise XOR operation. In XOR linked list instead of storing actual memory addresses, every node stores the XOR of address of previous and next nodes. The purpose of XOR linked list is to save space for one address. In XOR linked list we can work out the operations are done in deque.

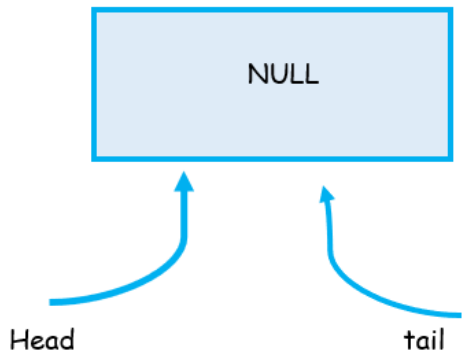


Fig-1:representation of head and tail when no nodes are present

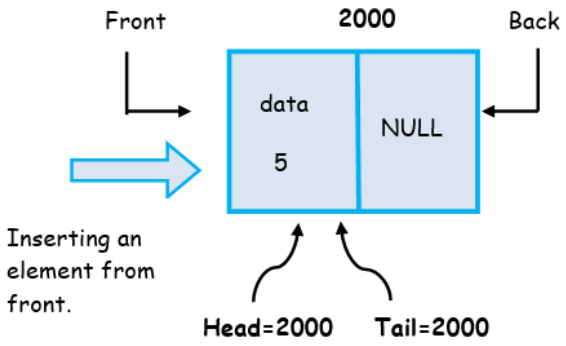


Fig-2: representation of deque when only one element is present

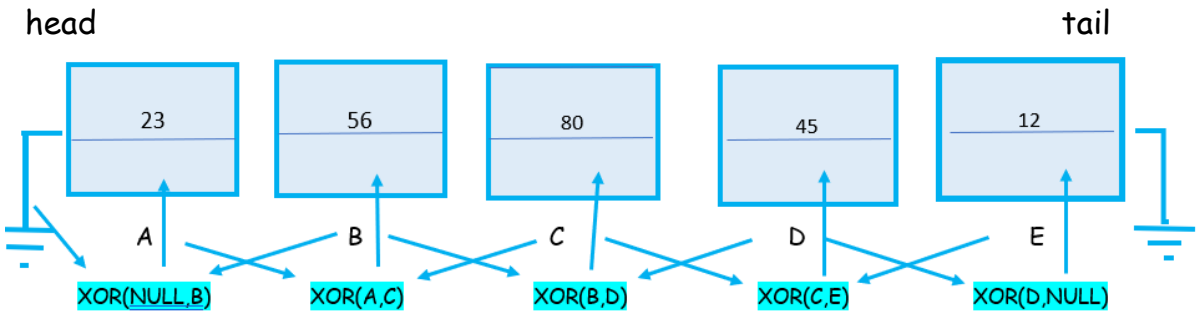


Fig-3: representation of deque through efficient double linked list contains n elements.

To implement deque we build certain funtions .They are:

- void push_back(int)
- void push_front(int)
- void pop_back()
- void pop_front()
- int get_back()
- int get_front()
- int get_secondback()
- int get_secondfront()
- int size()
- bool empty()

1. void push_back(int x);

Tail pointer changes from last node to the new node by making tail->npx to xor(prev,new node) and moving tail to new node. Head pointer changes only if the inserted node is the first node.

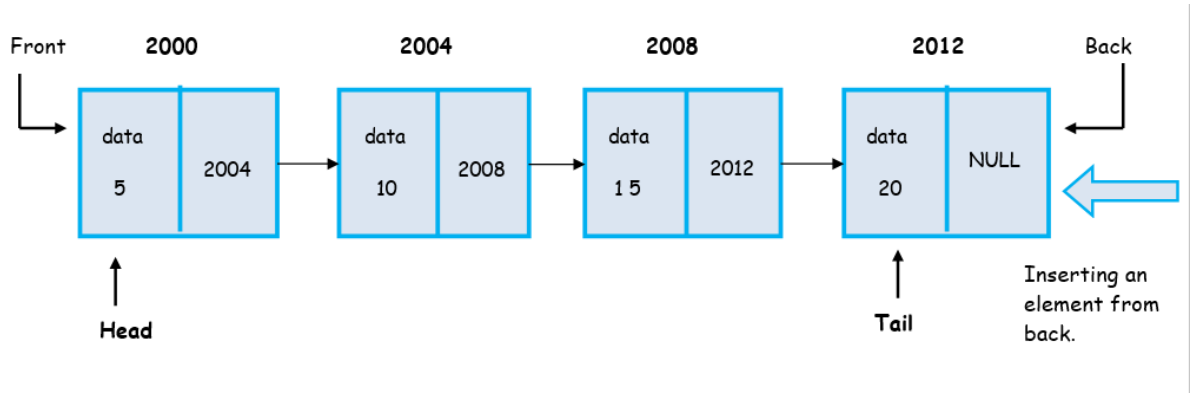


fig-4:representation of inserting node at end

2. void push_front(int x)

Head pointer changes from first node to new node by making head->npx to xor(prev,new node) and moving head to new node.Tail pointer changes ony if the inserted element is the first one.

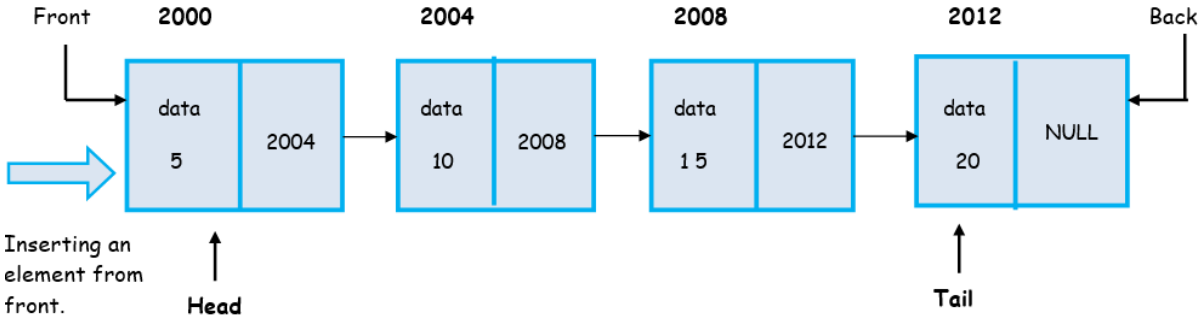


Fig 5: representation of inserting node at beginning

3. void pop_back()

In this operation we move tail pointer to its previous node by erasing the last node .It is done as follows,we consider prev node by tail->npx then we change the prev->npx as xor(prev->npx,tail) and then we change tail to prev and erase the last node. Here ,we check

condition that if after the operation tail becomes null ,we make head to null.

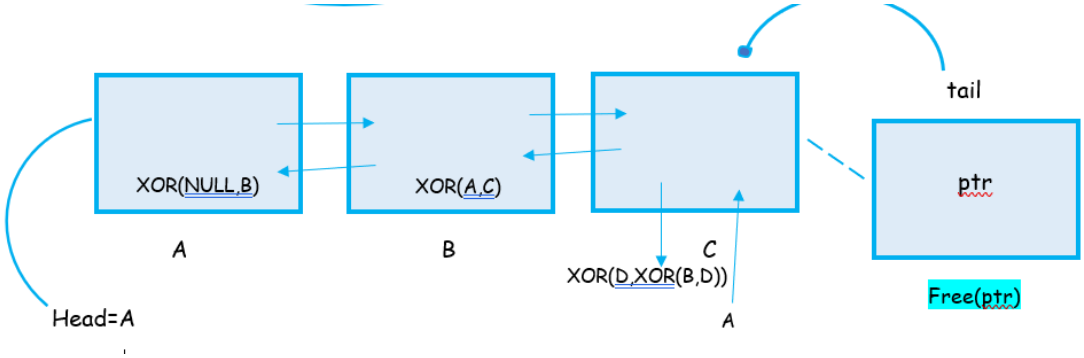


fig 6: representation of removing element from the end

4. void pop_front()

In this operation we move head pointer to its next node by erasing the first node .It is done as follows,we consider next node by head->npx then we change the next->npx as xor(next->npx,head) and then we change head to next and erase the first node. Here ,we check condition that if after the operation head becomes null we make tail to null.

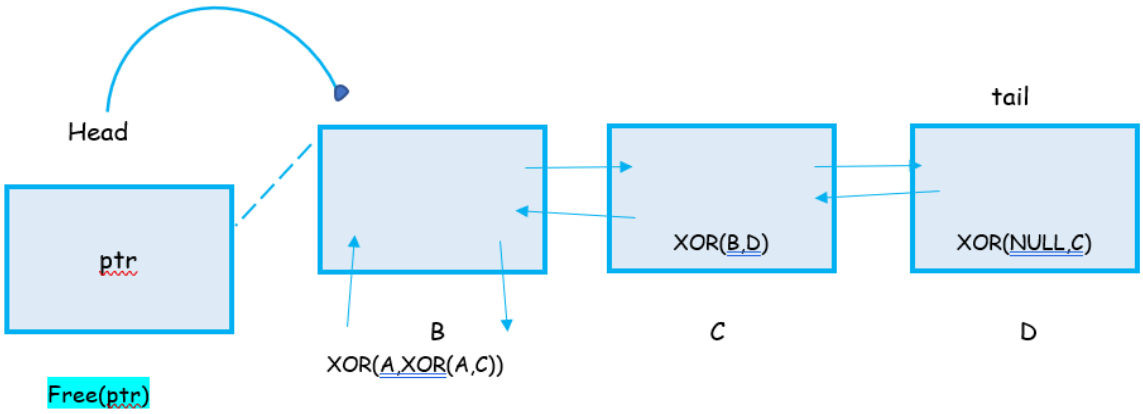


Fig 7: representation of removing element from the beginning

5. back() and front()

In these operations we get element at the respective end pointer like for back() we get element at tail and for front() we get element at head;

6. size()

This function returns that are present in the dequeue

7. isEmpty()

This function returns bool whether linked list is empty or not if linked list is empty it returns true and false if elements are present

1.4 Measurements to be done:

We can find out the time taken by different parts of a program by using the **std::chrono** library.

std::chrono has two distinct objects–timepoint and duration. A timepoint as the name suggests represents a point in time whereas a duration represents an interval or span of time. The C++ library allows us to subtract two timepoints to get the interval of time passed in between. Using provided methods we can also convert this duration to appropriate units.

The **std::chrono** provides us with three clocks with varying accuracy. The **high_resolution_clock** is the most accurate and hence it is used to measure execution time.

1.5 Constraints

- it will confuse the compiler, debugging and static analysis tools as your XOR of two pointers will not be correctly recognised by a Pointer by anything except your code.

- It also slows down pointer access to have to do the XOR operation to recover the true pointer first.
- It also can't be used in managed code — XOR obfuscated pointers won't be recognised by the garbage collector.
- This only works if we start our XOR linked list traversal from the start or end - as if we just jump into a random node in the middle, we do not have the information necessary to start traversing.

1.6 Assumtions

In XOR linked list, **only one address field is maintained** whose value is determined by the XOR of the previous node address and the next node address:

- **current_node(link) = addr(previous_node) addr(next_node)**
- $X \oplus X = 0$
- $X \oplus 0 = X$
- $X \oplus Y = Y \oplus X$
- $(X \oplus Y) \oplus Z = X \oplus (Y \oplus Z)$

Source Code:

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

struct Node{
    int data;
    struct Node*npx;
};
struct Node *head=NULL,*tail=NULL;
int size=0;


struct Node* Xor(struct Node *prev,  struct Node *next)
{
   return ( struct Node*)((uintptr_t)(prev) ^ (uintptr_t)(next));
}
```

```
void push_back(int x){
    struct Node *nn=(struct Node*)malloc(sizeof (struct Node*));
    nn->data=x;
    if(head==NULL)
    head=tail=nn;
    else{
        tail->npx=Xor(tail->npx,nn);
        nn->npx=tail;
        tail=nn;
    }
    size++;
}

void push_front(int x){
    struct Node *nn=(struct Node*)malloc(sizeof (struct Node*));
    nn->data=x;
    if(head==NULL)
    head=tail=nn;
    else{
        head->npx=Xor(head->npx,nn);
        nn->npx=head;
        head=nn;
    }
    size++;
}
```

```
void pop_back(){
    struct Node*ptr;
    if(head==NULL|| tail==NULL){
        printf("list is empty");
        return;
    }
    else{
        ptr=tail;
        struct Node *prev=Xor(ptr->npx,NULL);
        if(prev==NULL){
            head=NULL;
        }
        else{
            prev->npx=Xor(ptr,Xor(prev->npx,NULL));
        }
        tail=prev;
        if(tail==NULL)head=NULL;
        free(ptr);
    }
    size--;
}
```

```
void pop_front(){
    struct Node *ptr=head;
    if(head==NULL || tail==NULL){
        printf("list is empty");
        return;
    }
    else{
        struct Node *next=Xor(NULL,ptr->npx);
        if(next==NULL)
        tail=NULL;
        else{
            ptr=head;
            next->npx=Xor(ptr,Xor(NULL,next->npx));
        }
        head=next;
        if(tail==NULL)head=NULL;
        free(ptr);
    }
    size--;
}
```

```
int front(){
    if(head==NULL)
    return -1;
    int p=head->data;
    return p;
}

int back(){
    if(tail==NULL)
    return -1;
    else{
        int q=tail->data;
        return q;
    }

}
```

```
int secondBack(){
    if(tail==NULL)
    return -1;
    struct Node *prev=Xor(tail->npx,NULL);
    if(prev==NULL)
    return -1;
    else{
        int p=prev->data;
        return p;
    }
}

int secondFront(){
    if(head==NULL)
    return -1;
    struct Node *next=Xor(next->npx,NULL);
    if(next==NULL)
    return -1;
    else{
        int q=next->data;
        return q;
    }
}

long long int size(){
    return size;
}
```

```
bool isEmpty(){
    if(head==NULL)
    return true;
    return false;
}
int main() {
    while(1){
        int x;cin>>x;
        if(x==0)break;
        int ch;cin>>ch;
        switch(ch){
        case 1:cout<<isEmpty()<<"\n";break;
        case 2:int a;cin>>a;push_back(a);cout<<Size()<<"\n";break;
        case 3:int b;cin>>b;push_front(b);cout<<Size()<<"\n";break;
        case 4:pop_back();break;
        case 5:pop_front();break;
        case 6:cout<<front()<<" ";break;
        case 7:cout<<back()<<" ";break;
        case 8:cout<<secondBack()<<" ";break;
        case 9:cout<<secondFront()<<" ";break;
        case 10:cout<<Size()<<" ";break;
        default:break;
        }
    }

    return 0;
}
```

Source code link: https://ideone.com/yWDErq

Test Plans:

Testing all the possible cases:

Test case	Description	Expected output	Actual output	Test Cases Link
1.	we push the element in front and whether it pops it from back side and returning the size	0	0	https://ideone.com/1FqS3d
2.	Pushing 100 elements in the deque and popping out one element and returning the size.	99	99	https://ideone.com/904fw7
3.	Returning the empty deque with size	1 0	1 0	https://ideone.com/8Eks8i
4.	In this we push front element and we try to get 2nd back element which is not possible	-1	-1	https://ideone.com/yQhXdR
5.	We push one element in front and back side we try to get the 2nd back and front element	0 1	0 1	https://ideone.com/VIRHT8

6.	We try to push and pop the element and retrive the 2nd element	1 -1	1 -1	https://ideone.com/UE5xRo
7.	If we try to push between INT_MIN and INT_MAX we get run time error	Runtime error	Runtime error	https://ideone.com/VOIpTl
8.	If we try to push from 0 to INT_MAX we get run time error	Runtime error	Runtime error	https://ideone.com/uJxqZP
9.	We insert front and back 10000 times find the size	20000	20000	https://ideone.com/sZkLwa
10.	We check for empty and then insert the element and retrieve last second.	1 -1	1 -1	https://ideone.com/ncVBFT
11.	Trying to push 10^8 elements	Runtime error	Runtime error	https://ideone.com/G1Xwdj
12.	Trying to insert elements from INT_MIN to INT_MAX and retrieving second back and second front	Runtime error	Runtime error	https://ideone.com/IXsgZe
13.	Trying to insert elements from INT_MIN to INT_MAX and printing size and front and back	Runtime error	Runtime error	https://ideone.com/kUtvdB

14.	Trying to insert elements from 0 to INT_MAX and retrieving front back size	Runtime error	Runtime error	https://ideone.com/bjLFPi
15.	Inserting 3 times and accessing front back and size.	2 2 -1	2 2 -1	https://ideone.com/mq2k5h
16.	Trying to access size,back,front and then inserting and accessing front and back	0 -1 -1 10 10 2	0 -1 -1 10 10 2	https://ideone.com/HaQjMg
17.	Inserting and poping exactly the same times and then accessing front and size	10 1	10 1	https://ideone.com/xs3G3f
18.	Pushing element and printing second front then popping and then accessing second back	-1 -1	-1 -1	https://ideone.com/4qLkus
19.	Trying to access front back and pop in empty list	-1 -1 List is empty	-1 -1 List is empty	https://ideone.com/dmq99W
20.	Trying to get second back front size from empty list	-1 -1 0	-1 -1 0	https://ideone.com/7xBvBR

21.	Trying to insert 1000 elements and removing 1000 elements and accessing front	-1	-1	https://ideone.com/OPY1N3
22.	Trying to access front back second front back size for a single node deque	23 23 0	23 23 0	https://ideone.com/PYi94U
23.	Measuring time for 1000 elements insertion and one elemant	34 0	34 0	https://ideone.com/FkqKhk

Pass and fail criteria for different operations

Functions	Pass/Fail Criteria
1. Push_back() 2. Push_front()	The function will pass or it can take upto 10^7 values
3. Pop_front() 4. Pop_back()	It passes all the criteria but it can pop up to 5041 elements after that we get runtime error
5. get_front() 6. get_back()	Passes all the cases
7. get_secondback() 8. get_second_front()	Passes and fails in some cases
9. Empty() 10. Deque_size()	Passes all the cases

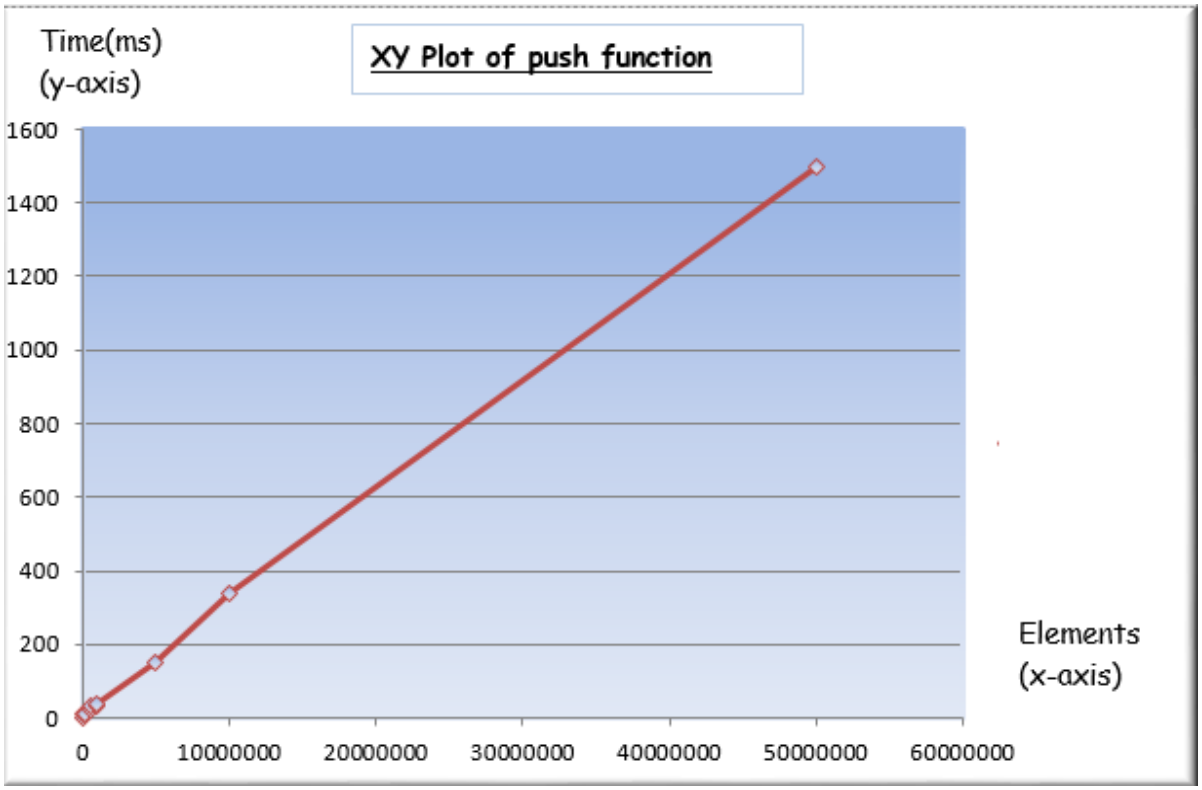
Measurements and analysis

Theoretical time complexities for all the functions/operations will be O(1)

For push operation:

No. of elements	T(micro seconds)
10	0
100	10
1000	10
10000	10
100000	10
500000	20
600000	30
900000	30
1000000	40
5000000	150
10000000	340

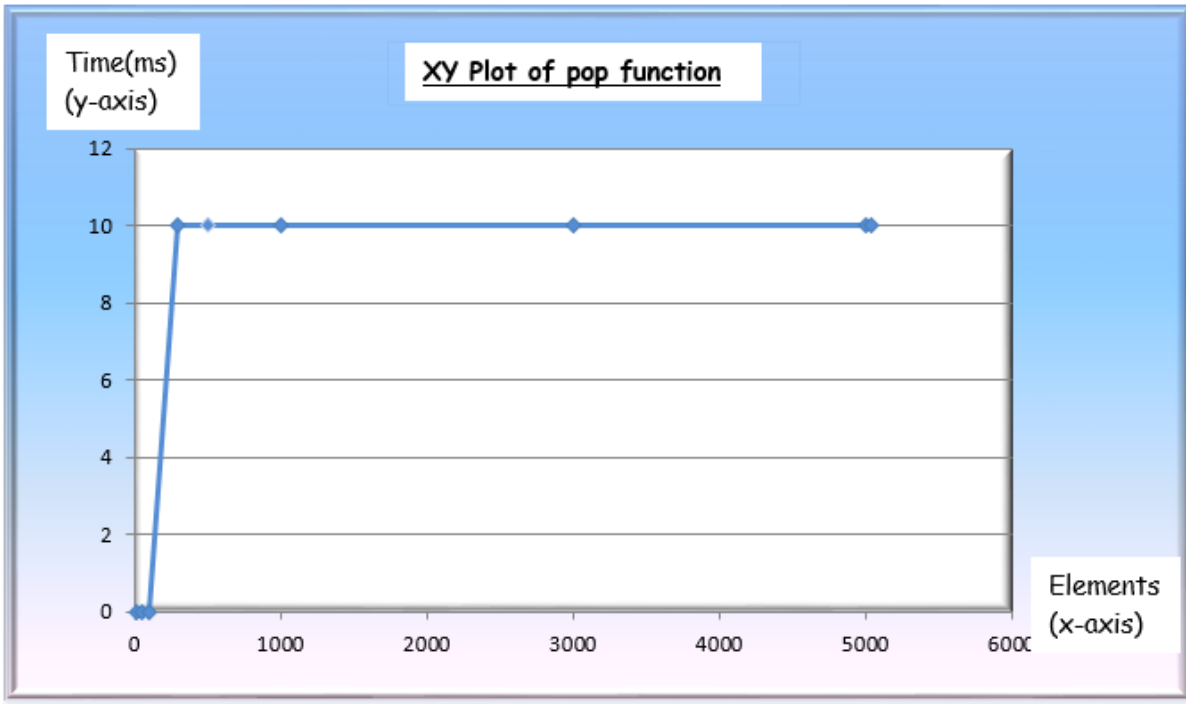
Graph for push function



For pop function:

No. of elements	Time(ms)
10	0
50	0
100	0
300	10
500	10
1000	10
3000	10
5000	10
5041	10

Graph for pop function:



Conclusions

We can conclude that XOR linked list or a memory efficient Double Linked List can be used instead of ordinary linked list because an ordinary linked list stores addresses of the previous and next list items in each list node, requiring two pointers for two address fields.

XOR linked list has only one pointer to traverse the list back and front. It uses bitwise XOR operator to store the front and rear pointer addresses. Instead of storing actual memory address, every node store the XOR address of previous and next nodes.

Future Enhancements

It has wide range of applications like web browsers history visited URL's ,storing a software applications list of undo and redo operations.

Difficulties Faced

- XOR linked list do not provide the ability to delete a node from the list by knowing only its address.
- It do not provide the ability to insert a new node before or after an existing node when knowing only the address of the existing node.

Reference Links

- https://en.m.wikipedia.org/wiki/XOR_linked_list
- https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-1/
- https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/
- https://www.techiedelight.com/xor-linked-list-overview-implementation-c-cpp/