|  |
| --- |
| PROJECT TITLE:**DEQUEUE using XOR** |
|  |

TEAM MEMBER’S DETAILS:

19H51A04M4 - Jeevitha - ECE

19H51A05M3 - Sreelatha - CSE

19H51A05N8 - Mounika - CSE

19H51A05P5 - Sathvika - CSE

19H51A05P7 - Udaya Sri - CSE

MENTOR:

Ashutosh K Akhouri sir

SEMESTER, YEAR:

4th semester, 2nd year

# **ABSTRACT**

The aim of this project is to develop a dequeue which act as a doubly linked list. Here we can move forward and reverse in XOR linked list. This is a memory efficient doubly linked list. Instead of sorting address of next node, it stores XOR of previous and next address.

General linked list

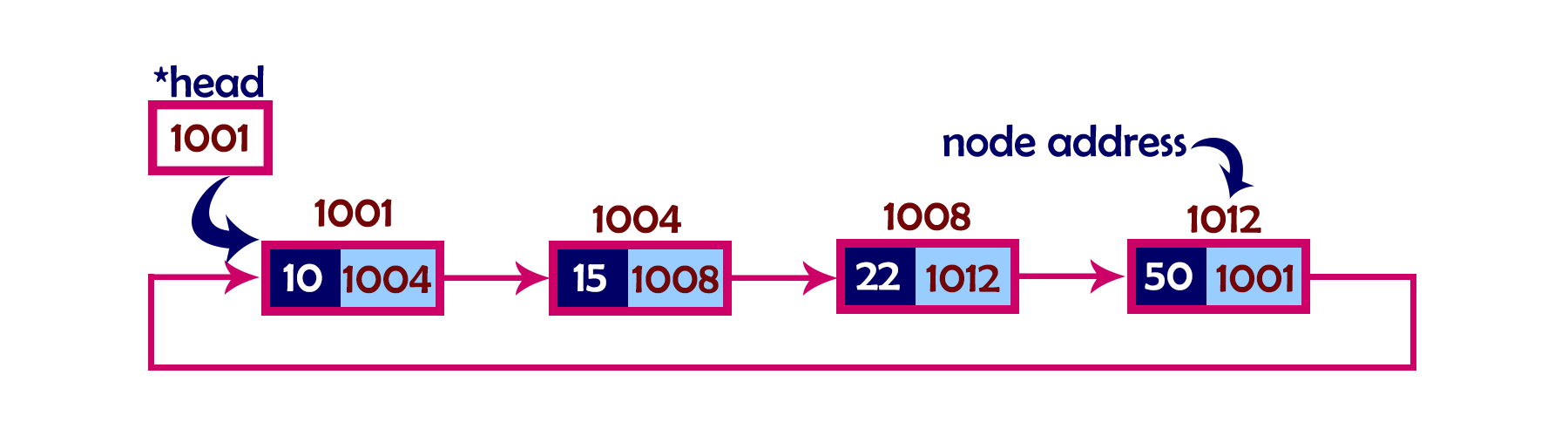
A -> B -> C -> D -> NULL

XOR linked list

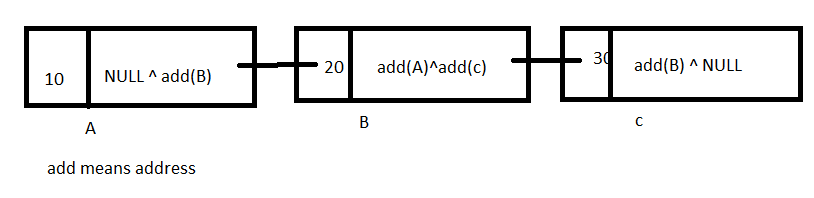
Data A B C D

next 0 XOR B A XOR C B XOR D C XOR 0

linked list:



XOR linked list:



# PURPOSE:

It takes less memory than doubly linked list. But acts as doubly linked list.Deque should be implemented using only single pointer.

# CONCLUSION:

* Garbage collection is difficult because no proper address is stored.
* Debugging becomes hard.
* Because we will not store actual address.

**Table of Contents**

**PROJECT DESCRIPTION**

Purpose of the project 5

Goals/requirements 6

**METHODOLOGY:**

Alternative approaches 7-8

Current approach chosen and Detailed Description of current approach 9-12

**Measurements to be done:**

Constraints and Assumptions: 13

**Code**  14-20

**TEST PLANS**

Approach 21

Features to be tested/ not tested 21

Pass/ fail criteria 22-23

List of test cases 24-37

Test programs listing 38-39

**MEASUREMENT AND ANALYSIS**

Tabular data for measured time-taken vs N and Graph plotting 40-41

**CONCLUSIONS**  41

**FUTURE ENHANCEMENTS AND DIFFICUTIES FACED**  42

**REFERENCE LINKS 42**

# 

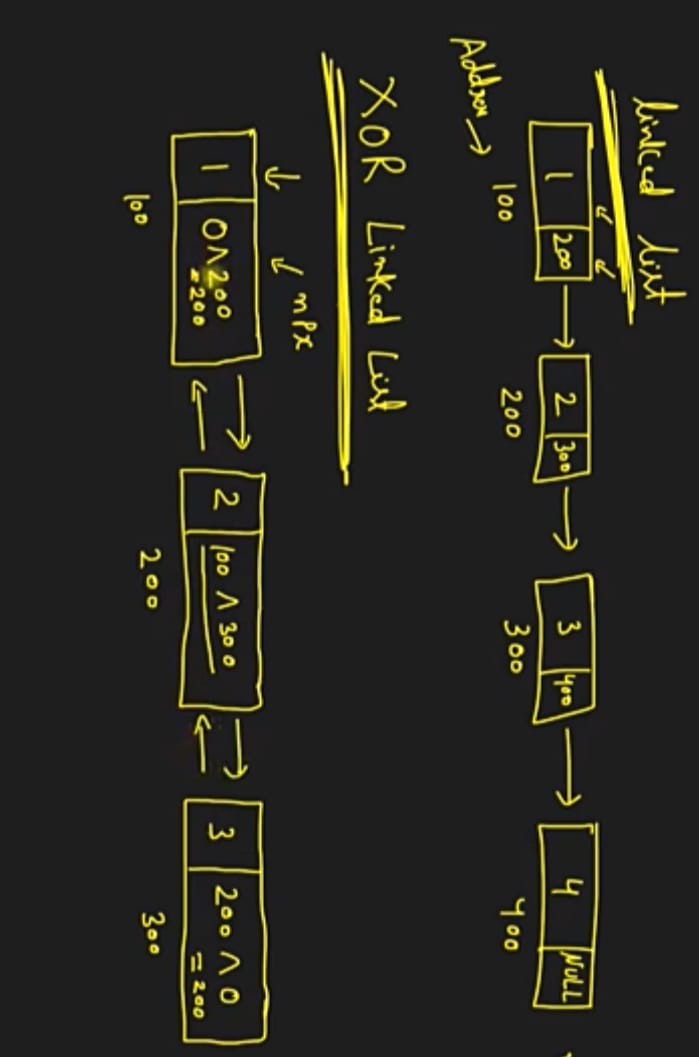
# **PROJECT DESCRIPTION**

# PURPOSE OF THE PROJECT:

* An XOR linked list is a type of data structure used in computer programming.
* It takes advantage of the bitwise XOR operation to decrease storage requirements for doubly linked list.
* Apart from saving memory, it allows for O (1) reversal, while still supporting all the other destructive update operations efficiently, like
* Concating two lists destructively in O (1)
* Insert After /Insert Before in O (1), when you only have a reference to the node and its successor/predecessor (which differs slightly from standard doubly linked list
* Remove in O (1), also with a reference to either the successor or predecessor.

# 2.GOALS/REQURIMENTS:

* Memory efficient.
* Single pointer XOR.
* Time complexity is O (1) for inserting and removing of element with XOR operation.

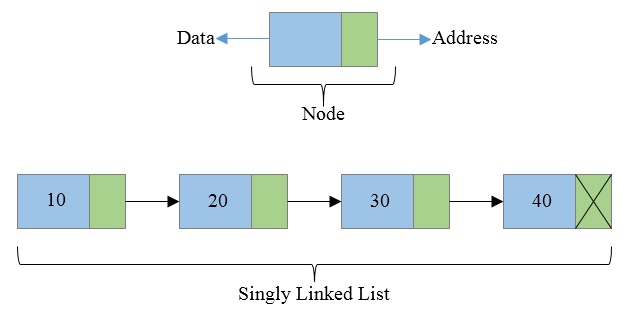


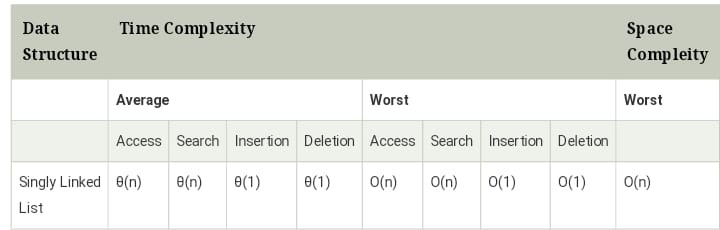
# 3.methodology:

**3.1 ALTERNATIVE APPROACHES:**

1. **Singly linked list:**

* SLL can be defined as collection of ordered set of elements.
* Here reverse traversal is not possible.
* Here popping an element from the back of the linked list is difficult.



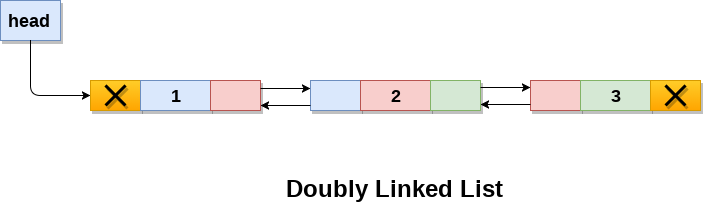


2.Deque using 2 Stacks

3.Deque using Circular Array

**4)Doubly linked list:**

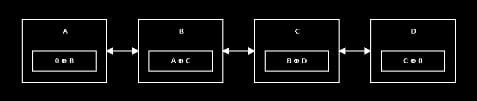
* DLL is a complexity type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.
* Previous part of the first node and the next part of the last node will always contain NULL indicating end in each direction.
* It uses extra memory when compared to other Data structure.since elements in memory are stored randomly, therefore the elements are accessed sequentially.
* No direct access is allowed.



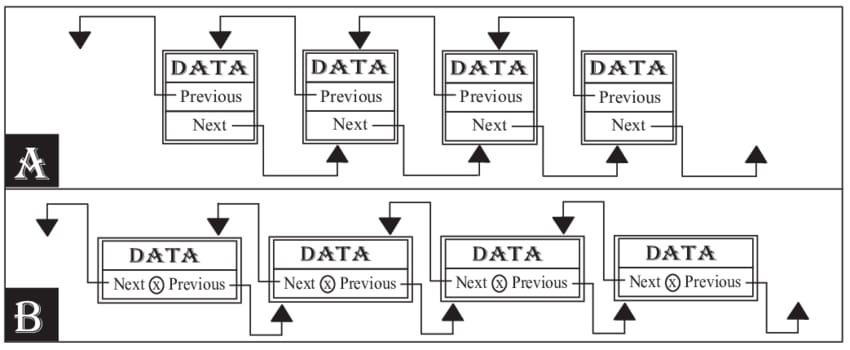
**3.2 CURRENT APPROACH:**

**XOR Linked List**

* An ordinary Doubly Linked List requires space for two address fields to store the addresses of previous and next nodes.
* 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.



A – Doubly linked list

B – XOR linked list

**3.3 DETAILED DESCRIPTION OF CURRENT APPROACH:**

Here implementation of memory-efficient doubly linked list has two simple functions.

1. A function to insert a new node at the beginning.
2. A function to traverse the list in forward direction.

**XOR List Representation:**

Here we 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

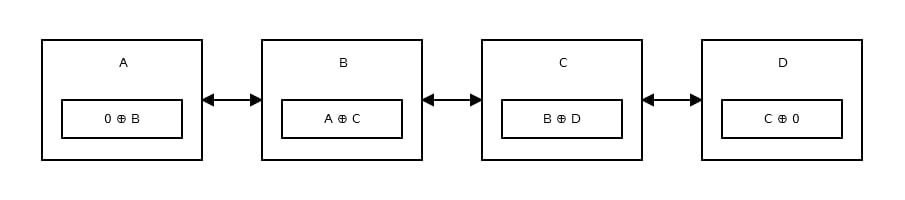
npx(C) XOR add(B)

=> (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



# MEASURMENTS TO BE DONE:

* Here all operations are done in O (1).
* Here each block of the code is done by bitwise XOR operator to save memory.
* Here execution time of various functions are done by std::chrono library.

# CONSTRAINTS:

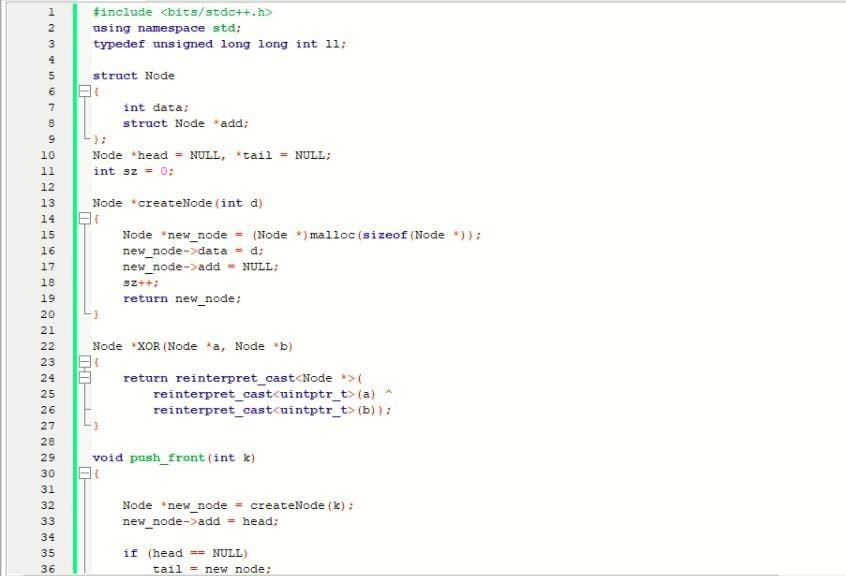
The values to be inserted in the XOR dequeue is from

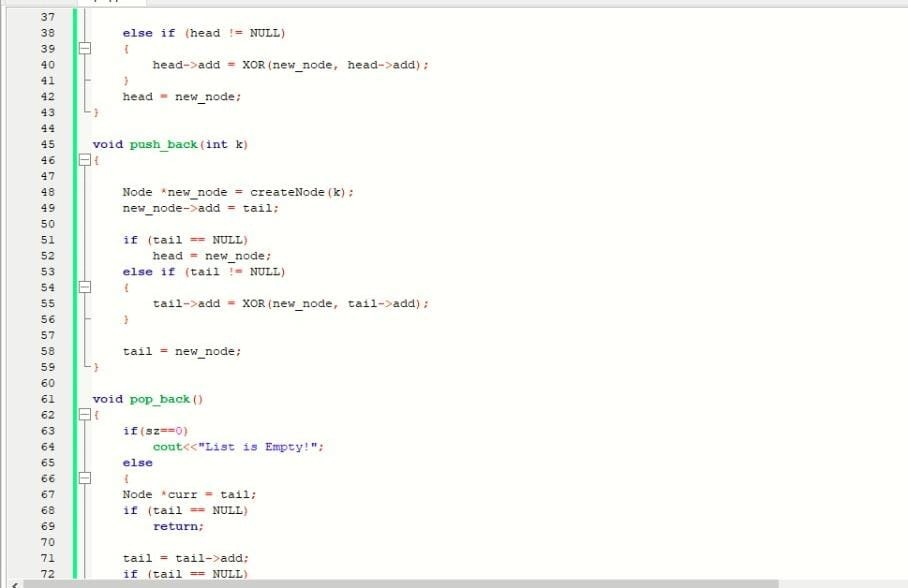
* -2147483648<=data<=2147838648
* 10^-8 <= No. of Elements <= 10^8

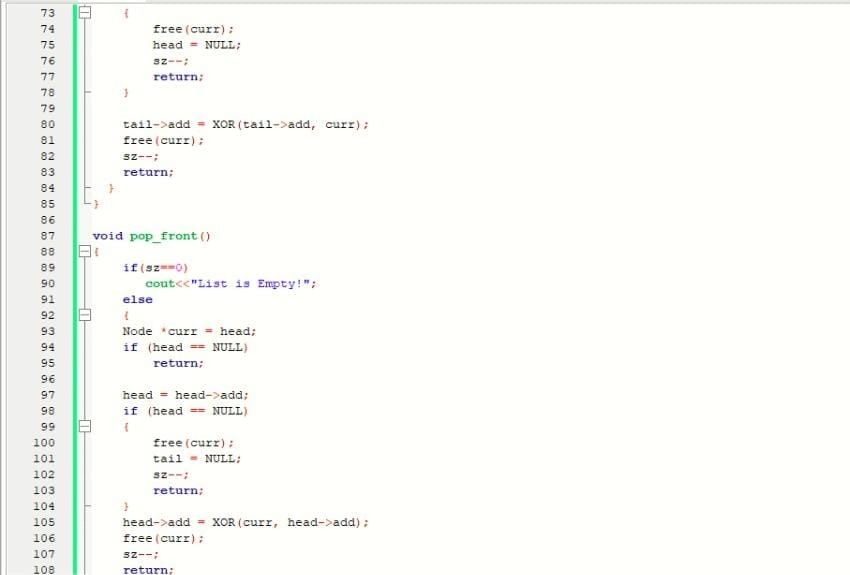
# ASSUMPTIONS:

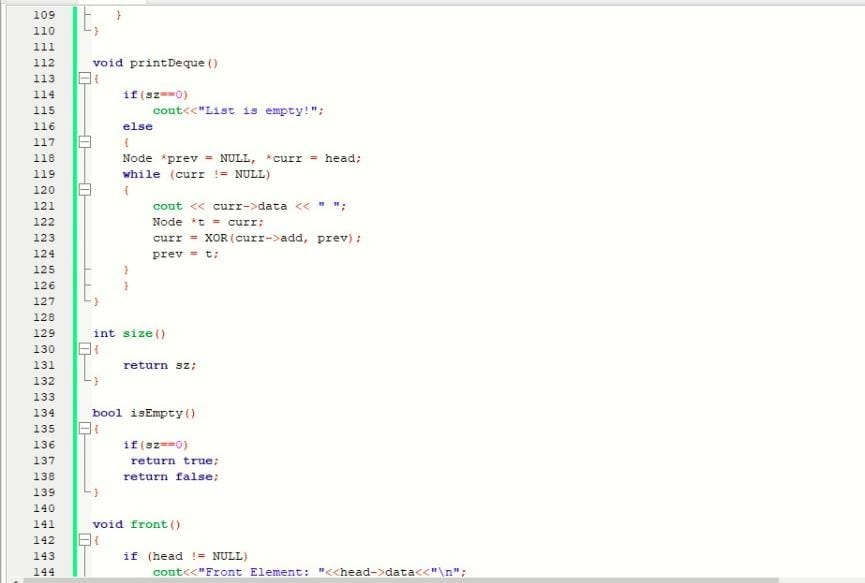
The deque doesn’t allow strings, characters, double or long integers and also doesn’t support for user defined data types as union and structure.

# CODE:



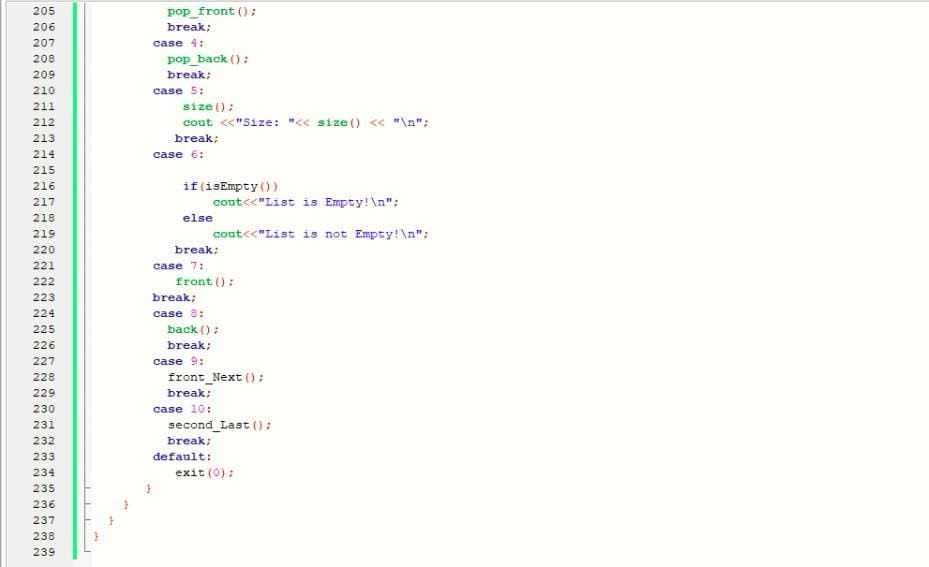












**TEST PLANS:**

APPROACH:

For saving space we use single pointer approach by using XOR operation. Hence, every node has the XOR of address of next and previous nodes

**FEATURES TO BE TESTED/NOT TESTED:**

TESTED:

When the dequeue is empty:

* Pop\_front ()
* Pop\_back ()
* Size ()
* Front ()
* Back ()

When the dequeue has single element:

* Pop\_front () applied twice
* Pop\_back () applied twice
* Pop\_front () and pop\_back () applied together.

When dequeue is filled using only push\_front ():

* Back ()

When dequeue is filled using only push\_back():

* Front ()

NOT TESTED: We have not tested when the dequeue limits till memory size.

**PASS FAIL CRITERIA:**

|  |  |
| --- | --- |
| Test cases | **Pass/fail criteria(output)** |
| **1.when dequeue is empty** | |
| * Output with pop\_front() * Output with pop\_back() | No elements in the queue so nothing to be popped out |
| * Output with front() * Output with back() | No element in the queue |
| * Output with size() | 0 |
| * Output with front\_next() * Output with second\_last() | Queue is empty! |
| **2.when dequeue has single element** | |
| * Output with front\_next() * Output with second\_last() | Less than two elements |
| * Output with pop\_front() apply twice * Output with pop\_back() apply twice * Output with pop\_front() and pop\_back() apply twice | The queue is already empty |
| 3.when the dequeue is filled using only push\_back() | |
| * Output with back() | Prints the first element entered using push\_front() |
| **4.when the dequeue is filled using only push\_back()** | |
| * Output with front() | Prints the first element entered using push\_back() |
| **5.with random entries ex: 1 2 3 4 5** | |
| * Output with front() | 5 |
| * Output with back() | 1 |
| * Output with size() | 5 |

**LIST OF TEST CASES:**

* **TC1:**

All cases for list is empty

[**https://www.ideone.com/IlzZeh**](https://www.ideone.com/IlzZeh)

Input

5 6 7 8 9 10 0

Output:

Size: 0

List is Empty!

List is Empty!

List is Empty!

list is Empty!

list is empty!

* **TC2:**

Checking Max number of elements can be inserted.

Logic :

for(i=0;i<10000000;i++){

push\_back(5);}

cout << front() << " ";

cout << back() << " ";

cout << size() << "**\n**";

Output :

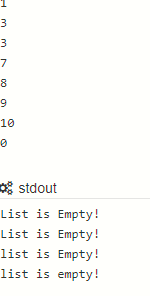
5 5 10000000

[**https://ideone.com/q2LTxQ**](https://ideone.com/q2LTxQ)

* **TC3:**

List is empty when we pop\_front() for single element & try to call front, back, front next & second last.

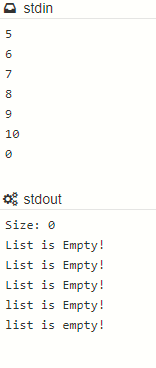
[**https://www.ideone.com/RFuD3W**](https://www.ideone.com/RFuD3W)

****

* **TC4:**

List is empty when pop\_back() for single element & try to call front, back, front next & second last.

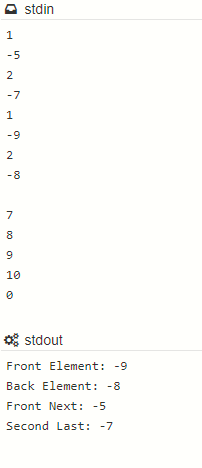
[**https://www.ideone.com/IlzZeh**](https://www.ideone.com/IlzZeh)



* **TC5:**

When we insert Negative numbers using push\_front & push\_back.

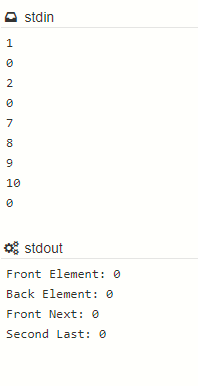
[**https://www.ideone.com/5FR5aa**](https://www.ideone.com/5FR5aa)



* **TC6:**

When instert 0 element into the list & size increases.

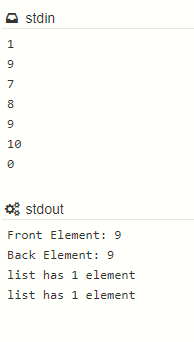
[**https://www.ideone.com/D3wl8h**](https://www.ideone.com/D3wl8h)



* **TC7:**

When one element inserted front back, front next, second last

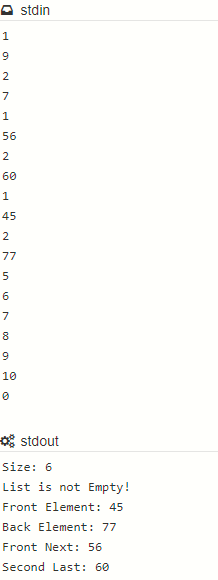
[**https://www.ideone.com/geBD5K**](https://www.ideone.com/geBD5K)



* **TC8:**

When two elements inserted front back, front next, second last.

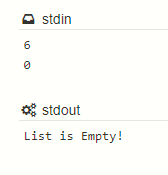
[**https://www.ideone.com/sZizMR**](https://www.ideone.com/sZizMR)



* **TC9:**

List is empty when no elements are inserted.

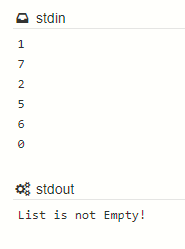
[**https://www.ideone.com/OzXEuc**](https://www.ideone.com/OzXEuc)



* **TC10:**

List is not empty when elements are inserted.

<https://www.ideone.com/TMYOl9>



* **TC11:**

Overflow – when the maximum elements are inserted & still trying to insert elements.

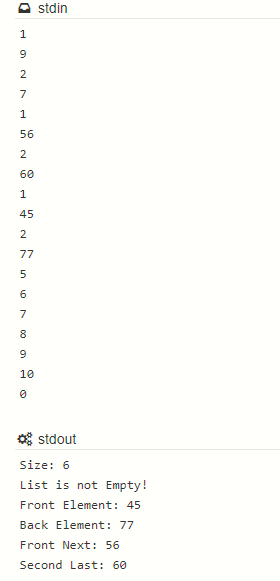
[**https://ideone.com/yWSgBd**](https://ideone.com/yWSgBd)



* **TC12:**

All cases when list has elements

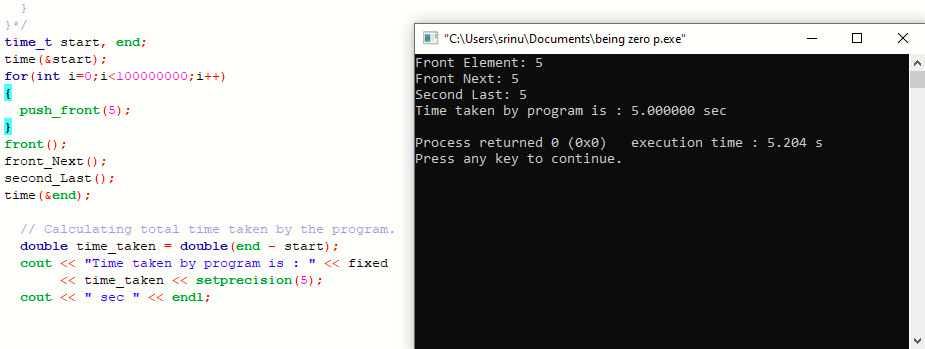
[**https://www.ideone.com/sZizMR**](https://www.ideone.com/sZizMR)



* **TC13:**

Front next and second last for max elements

[**https://www.ideone.com/7qLoX9**](https://www.ideone.com/7qLoX9)



* **TC14:**

Operations performing pop front, size, front, back and pop back.

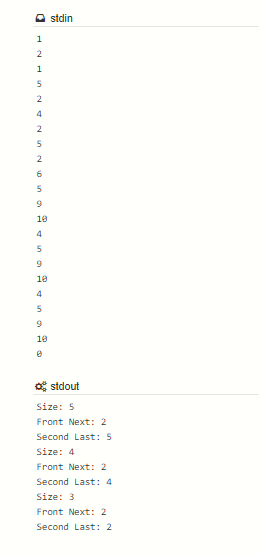
[**https://www.ideone.com/uBwaPY**](https://www.ideone.com/uBwaPY)



* **TC15:**

Operations performing pop front, pop back, front next and second last

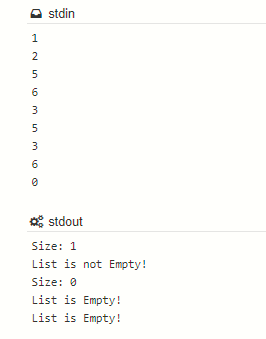
[**https://www.ideone.com/CyCv7Z**](https://www.ideone.com/CyCv7Z)



* **TC16:**

Single element pop front twice

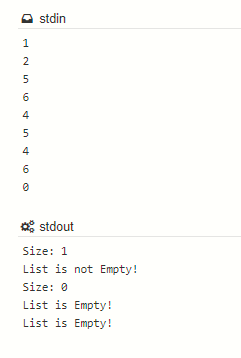
<https://www.ideone.com/GufS5i>



* **TC17:**

Single element pop back twice

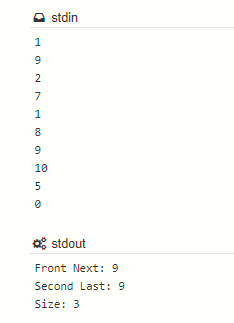
<https://www.ideone.com/0Tjq3l>



* **TC18:**

when three elements are inserted size, front next, and second last is same

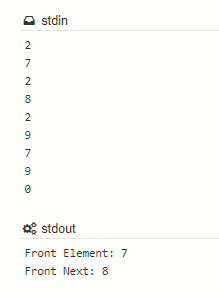
<https://www.ideone.com/3ymeCn>



* **TC19:**

Checking front and front next by filling the queue using only push back

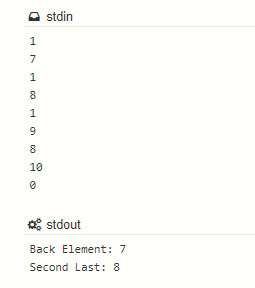
[**https://www.ideone.com/quezBX**](https://www.ideone.com/quezBX)



* **TC20:**

Checking back and second last by filling the queue using only push front

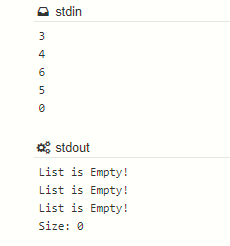
<https://www.ideone.com/8b4158>



* **TC21:**

Underflow. Still trying to pop by using pop front and pop back and size gives output as list is empty and s=0

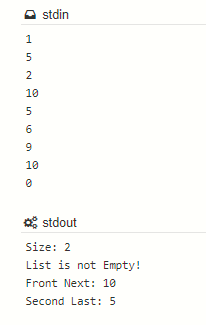
<https://www.ideone.com/ORqdWp>



* **TC22:**

Front next and second last

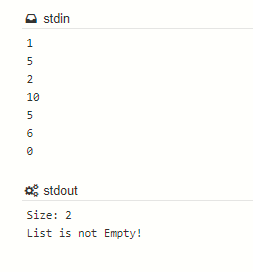
<https://www.ideone.com/5XP2Ht>



* **TC23:**

Size and empty function

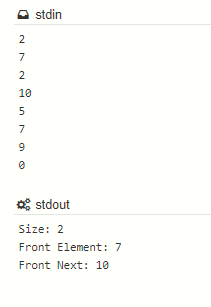
[**https://www.ideone.com/ySjAn0**](https://www.ideone.com/ySjAn0)



* **TC24:**

Normal push back function

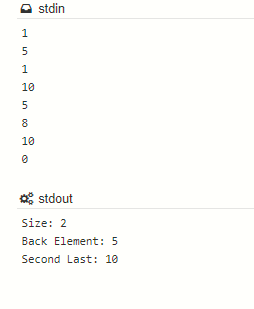
[**https://www.ideone.com/tLRUUF**](https://www.ideone.com/tLRUUF)



* **TC25:**

Normal push front funtion

[**https://www.ideone.com/zGKtj6**](https://www.ideone.com/zGKtj6)



# **TEST PROGRAMS LISTING:**

**TC1:** [**https://www.ideone.com/IlzZeh**](https://www.ideone.com/IlzZeh)

**TC2:** [**https://ideone.com/q2LTxQ**](https://ideone.com/q2LTxQ)

**TC3:** [**https://www.ideone.com/RFuD3W**](https://www.ideone.com/RFuD3W)

**TC4:** [**https://www.ideone.com/IlzZeh**](https://www.ideone.com/IlzZeh)

**TC5:** [**https://www.ideone.com/5FR5aa**](https://www.ideone.com/5FR5aa)

**TC6:** [**https://www.ideone.com/D3wl8h**](https://www.ideone.com/D3wl8h)

**TC7:** [**https://www.ideone.com/geBD5K**](https://www.ideone.com/geBD5K)

**TC8:** [**https://www.ideone.com/sZizMR**](https://www.ideone.com/sZizMR)

**TC9:** [**https://www.ideone.com/OzXEuc**](https://www.ideone.com/OzXEuc)

**TC10:** [**https://www.ideone.com/TMYOl9**](https://www.ideone.com/TMYOl9)

**TC11:** [**https://ideone.com/yWSgBd**](https://ideone.com/yWSgBd)

**TC12:** [**https://www.ideone.com/sZizMR**](https://www.ideone.com/sZizMR)

**TC13:** [**https://www.ideone.com/7qLoX9**](https://www.ideone.com/7qLoX9)

**TC14:** [**https://www.ideone.com/uBwaPY**](https://www.ideone.com/uBwaPY)

**TC15:** [**https://www.ideone.com/CyCv7Z**](https://www.ideone.com/CyCv7Z)

**TC16:** [**https://www.ideone.com/GufS5i**](https://www.ideone.com/GufS5i)

**TC17:** [**https://www.ideone.com/0Tjq3l**](https://www.ideone.com/0Tjq3l)

**TC18:** [**https://www.ideone.com/3ymeCn**](https://www.ideone.com/3ymeCn)

**TC19:** [**https://www.ideone.com/quezBX**](https://www.ideone.com/quezBX)

**TC20:** [**https://www.ideone.com/8b4158**](https://www.ideone.com/8b4158)

**TC21:** [**https://www.ideone.com/ORqdWp**](https://www.ideone.com/ORqdWp)

**TC22:** [**https://www.ideone.com/5XP2Ht**](https://www.ideone.com/5XP2Ht)

**TC23:** [**https://www.ideone.com/ySjAn0**](https://www.ideone.com/ySjAn0)

**TC24:** [**https://www.ideone.com/tLRUUF**](https://www.ideone.com/tLRUUF)

**TC25:** [**https://www.ideone.com/zGKtj6**](https://www.ideone.com/zGKtj6)

# MEASURMENT AND ANALYSIS:

***Time taken for push front:***

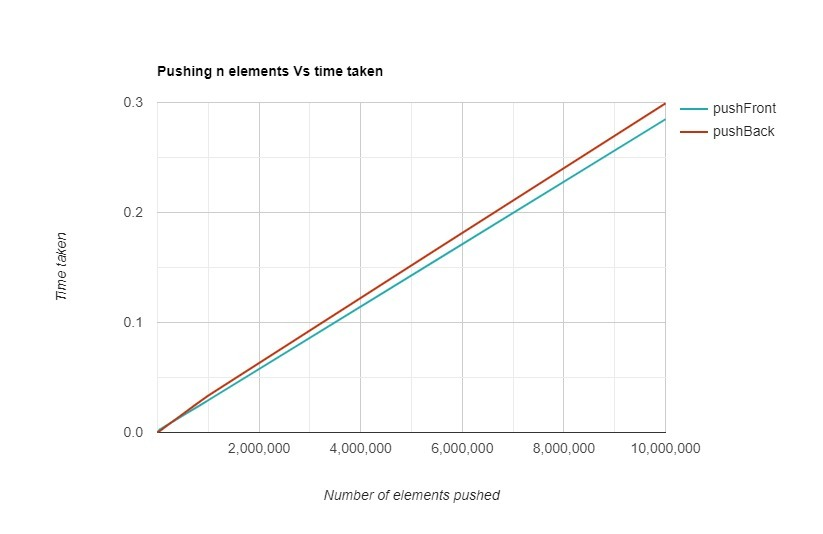
|  |  |
| --- | --- |
| **Nth – push front ()** | **Time taken** |
| 10 | 0.000002 |
| 500 | 0.000023 |
| 1000 | 0.000030 |
| 50000 | 0.002872 |
| 1000000 | 0.029336 |
| 10000000 | 0.284778 |

**Time taken to push back:**

|  |  |
| --- | --- |
| **Nth - push back()** | **Time taken** |
| 10 | 0.000003 |
| 500 | 0.000017 |
| 1000 | 0.000033 |
| 50000 | 0.001531 |
| 1000000 | 0.033469 |
| 10000000 | 0.299245 |

**Graph plotting:**

**Graph between time taken vs N:**



# 

# **CONCLUSION:**

* By using XOR linked list it decreases the storage.
* Traversing in both the directions using single operator “XOR”.
* It only requires single pointer to store both XOR address of next and previous nodes

# FUTURE ENHANCEMENT:

Not only for integer data type, we shall be able to modify the code for other datatypes also…

# DIFFICULTIES FACED:

* In calculating time complexities and drawing graphs.
* In finding maximum number of elements which are to be inserted.

# REFERENCE LINKS:

[**https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/**](https://www.geeksforgeeks.org/xor-linked-list-a-memory-efficient-doubly-linked-list-set-2/)

[**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-1/)

[**https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/**](https://www.geeksforgeeks.org/measure-execution-time-with-high-precision-in-c-c/)

[**https://www.rapidtables.com/tools/line-graph.htm**](https://www.rapidtables.com/tools/line-graph.htm)